NATIONAL OPEN UNIVERSITY OF NIGERIA

FACULTY OF SCIENCES

COURSE CODE: CIT 903

COURSE TITTLE: ADVANCED ARTIFICIAL INTELLIGENCE

Course Team

Course Code	CIT 903
Course Title	ADVANCED ARTIFICIAL INTELLIGENCE
Course Developer/Writer	Dr. Suleiman Zubair FUT, Minna
Content Editor	Prof. Francisca Nonyelum OGWUEKA NDA Kaduna
Course Material Coordination	Dr. Vivian Nwaocha, Dr. Greg Onwodi & Dr. Francis B.Osang Computer Science Department National Open University of Nigeria

NATIONAL OPEN UNIVERSITY OF NIGERIA

Headquarters

Plot 91, Cadastral Zone, Nnamdi Azikiwe Expressway, Jabi – Abuja,

Nigeria

Lagos Office

14/16 Ahmadu Bello Way

Victoria Island

Lagos

e-mail: @nou.edu.ng
URL: .nou.edu.ng

National Open University of Nigeria

First Printed --------2020

ISBN:

Printed by ……..
For
National Open University of Nigeria

Table of Contents Pages

Introduction

CIT 903: Advanced Artificial Intelligence is a 3-credit course for students studying towards acquiring a PhD in Information Technology and related disciplines.

The course is divided into 5 modules and 21 study units. It will introduce the students to Artificial Intelligence (AI), its foundations and the programing languages commonly used in AI. Basic Issues about AI, Application of AI techniques. Extensive Lab exercises in AI Language. And finally, it studies different classes of expert systems.

At the end of this course, it is expected that students should be able to understand, explain and be adequately equipped with basic issues AI and its applications in expert systems in general.

The course guide therefore gives you an overview of what the course: CIT 903 is all about, the textbooks and other course materials to be referenced, what you are expected to know in each unit, and how to work through the course material. It suggests the general strategy to be adopted and also emphasizes the need for self-assessment and tutor marked assignment. There are also tutorial classes that are linked to this course and students are advised to attend.

What you will learn in this Course

The overall aim of this course, CIT 903, is to boost the AI expertise of students to enable them develop AI based applications. This course provides extensive hands-on, case study examples, and reference materials designed to enhance your programming skills. In the course of your studies, you will be equipped with definitions of common terms, concepts and applications of AI. You will also learn about different classes of expert systems.

Course Aim

This course aims to give students an in-depth understanding of AI with a focus of its application on expert systems. It is hoped that the knowledge would enhance the AI expertise of students to enable them develop AI based applications.

Course Objectives

It is pertinent to note that each unit has precise objectives. Students should learn them carefully before proceeding to subsequent units. Therefore, it may be useful to refer to these objectives in the course of your study of the unit to assess your progress. You should always look at the unit objectives after completing a unit. In this way, you can be sure that you have done what is required of you by the end of the unit.

However, below are overall objectives of this course. On successful completion of this course, you should be able to:

- Explain what AI is all about
- Understand the foundation principles of AI
- Identify the best programing languages used for AI projects
- Understand the basic issues surrounding AI
- Outline the application areas of AI techniques

• Implement basic AI programs for search and scene analysis
• Understand how AI is used to implement various classes of expert systems

Working through this Course

To complete this course, you are required to study all the units, the recommended text books, and other relevant materials. Each unit contains some self-assessment exercises and tutor marked assignments, and at some point in this course, you are required to submit the tutor marked assignments. There is also a final examination at the end of this course. Stated below are the components of this course and what you have to do.

Course Materials

The major components of the course are:
1. Course Guide
2. Study Units
3. Text Books
4. Assignment File
5. Presentation Schedule

Study Units

There are 20 study units and 5 modules in this course. They are:

Recommended Texts

These texts will be of enormous benefit to you in learning this course:

1. Russell, S., & Norvig, P. (2005). AI a modern approach. Learning, 2(3), 4.
2. Rich E., & Knight K. (1991) Artificial Intelligence, McGraw-Hill, New York, NY
3. Shi, Z. (2011). Advanced artificial intelligence (Vol. 1). World Scientific.
4. Rothman, D. (2018). Artificial Intelligence By Example: Develop machine intelligence from scratch using real artificial intelligence use cases. Packt Publishing Ltd.
5. Coppin, B. (2004). Artificial intelligence illuminated. Jones & Bartlett Learning.
6. Nilsson, N. J., & Nilsson, N. J. (1998). Artificial intelligence: a new synthesis. Morgan Kaufmann.
7. Rothman, D., Lamons, M., Kumar, R., Nagaraja, A., Ziai, A., & Dixit, A. (2018). Python: Beginner's Guide to Artificial Intelligence: Build applications to intelligently interact with the world around you using Python. Packt Publishing Ltd.
8. Lucas, P., & Van Der Gaag, L. (1991). Principles of expert systems. Wokingham: Addison-Wesley.
9. Liebowitz, J. (Ed.). (1997). The handbook of applied expert systems. Crc Press.
10. Poole, D. L., & Mackworth, A. K. (2017). Python code for Artificial Intelligence: Foundations of Computational Agents.
11. Poole, D. L., & Mackworth, A. K. (2010). Artificial Intelligence: foundations of computational agents. Cambridge University Press.

Assignment File

The assignment file will be given to you in due course. In this file, you will find all the details of the work you must submit to your tutor for marking. The marks you obtain for these assignments will count towards the final mark for the course. Altogether, there are 40 tutor marked assignments for this course.

Presentation Schedule

The presentation schedule included in this course guide provides you with important dates for completion of each tutor marked assignment. You should therefore endeavor to meet the deadlines.

Assessment

There are two aspects to the assessment of this course. First, there are tutor marked assignments; and second, the written examination.

Therefore, you are expected to take note of the facts, information and problem solving gathered during the course. The tutor marked assignments must be submitted to your tutor for formal assessment, in accordance to the deadline given. The work submitted will count for 40% of your total course mark.

At the end of the course, you will need to sit for a final written examination. This examination will account for 60% of your total score.

Tutor Marked Assignments (TMAs)

There are 35 TMAs in this course. You need to submit all the TMAs. The best 4 will therefore be counted. When you have completed each assignment, send them to your tutor as soon as possible and make certain that it gets to your tutor on or before the stipulated deadline. If for any reason you cannot complete your assignment on time, contact your tutor before the assignment is due to discuss the possibility of extension. Extension will not be granted after the deadline, unless on extraordinary cases.

Final Examination and Grading

The final examination for CIT 903 will be of last for a period of 3 hours and have a value of 60% of the total course grade. The examination will consist of questions which reflect the self-assessment exercise and tutor marked assignments that you have previously encountered. Furthermore, all areas of the course will be examined. It would be better to use the time between finishing the last unit and sitting for the examination, to revise the entire course. You might find it useful to review your TMAs and comment on them before the examination. The final examination covers information from all parts of the course.

Course Marking Scheme

The following table includes the course marking scheme

Table 1: Course Marking Scheme

Assessment	Marks
Assignments 1-35	35 assignments, 40% for the best 4 Total = 10% X 4 = 40%
Final Examination	60% of overall course marks

Total	100% of Course Marks

Course Overview

This table indicates the units, the number of weeks required to complete them and the assignments.

Table 2: Course Organizer

Unit	Title of Work	Weeks Activity	Assessment (End of Unit)
	Course Guide	Week 1	
MODULE 1: INTRODUCTION TO AI			
Unit 1	What is Artificial Intelligence?	Week 1	Assignment 1 - 4
Unit 2	The State of Art in AI	Week 2	Assignment 5
Unit 3	AI Programming Languages	Week 3	Assignment 6
Unit 4	Types of AI	Week 4	Assignment 7 - 8
MODULE 2: BASIC AI ISSUES			
Unit 1	Attention	Week 5	Assignment 9
Unit 2	Search and Control	Week 6 - 7	Assignment 10 - 11
Unit 3	Adversarial Search (Game Tree)	Week 8	Assignment 12 - 13
Unit 4	knowledge representation	Week 9	Assignment 14 - 15
MODULE 3: APPLICATION OF AI TECHNIQUES			
Unit 1	Natural Language	Week 10	Assignment 16
Unit 2	Scene Analysis	Week 11	Assignment 17 - 18
Unit 3	Expert Systems	Week 12	Assignment 19 - 20
Unit 4	Robot Planning	Week 13	Assignment 20
MODULE 4: LAB. EXERCISES IN AI LANG			
Unit 1	Getting Started	Week 14	Assignment 21 - 22
Unit 2	Searching for Solutions	Week 15	Assignment 23 - 29
Unit 3	Multiagent Systems	Week 16	Assignment 30
Unit 4	Scene Processing	Week 17	Assignment 31
MODULE 5: STUDY OF DIFFERENT CLASSES OF EXPERT SYSTEMS			
Unit 1	Rule Based: MYCIN	Week 18	Assignment 32
Unit 2	Blackboard; HEARSAY II	Week 19	Assignment 33
Unit 3	Frame Based e.g. KEE	Week 20	Assignment 34
Unit 4	Independent study and Proposal Submission	Week 21	Assignment 35

How to get the most from this course

In distance learning, the study units replace the university lecturer. This is one of the huge advantages of distance learning mode; you can read and work through specially designed study materials at your own pace and at a time and place that is most convenient. Think of it as reading from the teacher, the study guide indicates what you ought to study, how to study it and the relevant texts to consult. You are provided with exercises at appropriate points, just as a lecturer might give you an exercise in class.

Each of the study units follows a common format. The first item is an introduction to the subject matter of the unit and how a particular unit is integrated with the other units and the course as a whole. Next to this is a set of learning objectives. These learning objectives are meant to guide your studies. The moment a unit is finished, you must go back and check whether you have achieved the objectives. If this is made a habit, then you will increase your chances of passing the course. The main body of the units also guides you through the required readings from other sources. This will usually be either from a set book or from other sources.

Self-assessment exercises are provided throughout the unit, to aid personal studies and answers are provided at the end of the unit. Working through these self-tests will help you to achieve the objectives of the unit and also prepare you for tutor marked assignments and examinations. You should attempt each self-test as you encounter them in the units.

The following are practical strategies for working through this course:

1. Read the course guide thoroughly
2. Organize a study schedule. Refer to the course overview for more details. Note the time you are expected to spend on each unit and how the assignment relates to the units. Important details, e.g. details of your tutorials and the date of the first day of the semester are available. You need to gather together all this information in one place such as a diary, a wall chart calendar or an organizer. Whatever method you choose, you should decide on and write in your own dates for working on each unit.
3. Once you have created your own study schedule, do everything you can to stick to it. The major reason that students fail is that they get behind with their course works. If you get into difficulties with your schedule, please let your tutor know before it is too late for help.
4. Turn to Unit 1 and read the introduction and the objectives for the unit.
5. Assemble the study materials. Information about what you need for a unit is given in the table of content at the beginning of each unit. You will almost always need both the study unit you are working on and one of the materials recommended for further readings, on your desk at the same time.
6. Work through the unit, the content of the unit itself has been arranged to provide a sequence for you to follow. As you work through the unit, you will be encouraged to read from your set books.
7. Keep in mind that you will learn a lot by doing all your assignments carefully. They have been designed to help you meet the objectives of the course and will help you pass the examination.

8. Review the objectives of each study unit to confirm that you have achieved them. If you are not certain about any of the objectives, review the study material and consult your tutor.
9. When you are confident that you have achieved a unit's objectives, you can start on the next unit. Proceed unit by unit through the course and try to pace your study so that you can keep yourself on schedule.
10. When you have submitted an assignment to your tutor for marking, do not wait for its return before starting on the next unit. Keep to your schedule. When the assignment is returned, pay particular attention to your tutor's comments, both on the tutor marked assignment form and also written on the assignment. Consult you tutor as soon as possible if you have any questions or problems.
11. After completing the last unit, review the course and prepare yourself for the final examination. Check that you have achieved the unit objectives (listed at the beginning of each unit) and the course objectives (listed in this course guide).

Tutors and Tutorials

There are 8 hours of tutorial provided in support of this course. You will be notified of the dates, time and location together with the name and phone number of your tutor as soon as you are allocated a tutorial group.

Your tutor will mark and comment on your assignments, keep a close watch on your progress and on any difficulties, you might encounter and provide assistance to you during the course. You must mail your tutor marked assignment to your tutor well before the due date. At least two working days are required for this purpose. They will be marked by your tutor and returned to you as soon as possible.

Do not hesitate to contact your tutor by telephone, e-mail or discussion board if you need help. The following might be circumstances in which you would find help necessary: contact your tutor if:

- You do not understand any part of the study units or the assigned readings.
- You have difficulty with the self-test or exercise.
- You have questions or problems with an assignment, with your tutor's comments on an assignment or with the grading of an assignment.

You should try your best to attend the tutorials. This is the only chance to have face-to- face contact with your tutor and ask questions which are answered instantly. You can raise any problem encountered in the course of your study. To gain the maximum benefit from the course tutorials, prepare a question list before attending them. You will learn a lot from participating in discussion actively. GOODLUCK!

Course Team

Course Code	CIT 903
Course Title	ADVANCED ARTIFICIAL INTELLIGENCE
Course Developer/Writer	Dr. Suleiman Zubair FUT, Minna
Content Editor	Prof. Francisca Nonyelum OGWUEKA NDA Kaduna
Course Material Coordination	Dr. Vivian Nwaocha, Dr. Greg Onwodi & Dr. Francis B.Osang Computer Science Department National Open University of Nigeria

NATIONAL OPEN UNIVERSITY OF NIGERIA

Headquarters

Plot 91, Cadastral Zone, Nnamdi Azikiwe Expressway, Jabi – Abuja,

Nigeria

Lagos Office

14/16 Ahmadu Bello Way

Victoria Island

Lagos

e-mail: @nou.edu.ng
URL: .nou.edu.ng

First Printed --------

ISBN:

Printed by ……..
For
National Open University of Nigeria

MODULE 1: INTRODUCTION

Unit One: What is AI
Unit Two: The State of Art in AI,
Unit Three: AI Programming Languages
Unit Four: Types of AI

Unit One: What is Artificial Intelligence?

CONTENT

1.0 Introduction

This chapter defines AI and presents Different people approach AI with different goals in mind. Two important questions to ask are: Are you concerned with thinking or behavior? Do you want to model humans or work from an ideal standard? In this study material, we adopt the view that intelligence is concerned mainly with rational action. Ideally, an intelligent agent takes the best possible action in a situation. We study the problem of building agents that are intelligent in this sense.

2.0 Objectives

At the end of this unit, you should be able to;

- Define AI
- Explain the foundation of the Turing Test approach to AI
- Identify which approach outlines the six (6) major disciplines of AI
- Explain the foundation of the cognitive modeling approach to AI
- Explain the foundation of the "laws of thought" approach to AI
- Identify the two main obstacles to the "laws of thought" approach to AI
- Understand the foundation of the rational agent approach to AI
- Outline advantages the rational-agent approach has over the other approaches

3.0 Main Content

Figure 1.1 presents eight definitions of AI, laid out along two dimensions. The definitions on top are concerned with thought processes and reasoning, whereas the ones on the bottom address behavior. The definitions on the left measure success in terms of fidelity to human performance, whereas the ones on the right measure against an ideal performance measure, called rationality. A system is rational if it does the "right thing," given what it knows. Historically, all four approaches to AI have been followed, each by different people with different methods. A human-centered approach must be in part an empirical science, involving observations and hypotheses about human behavior. A rationalist approach involves a combination of mathematics and engineering. The various group have both disparaged and helped each other. Let us look at the four approaches in more detail.

Thinking Humanly	**Thinking Rationally**
"The exciting new effort to make computers think . . . machines with minds, in the full and literal sense." (Haugeland, 1985) "[The automation of] activities that we associate with human thinking, activities such as decision-making, problem solving, learning . . ." (Bellman, 1978)	"The study of mental faculties through the use of computational models." (Charniak and McDermott, 1985) "The study of the computations that make it possible to perceive, reason, and act." (Winston, 1992)
Acting Humanly	**Acting Rationally**
"The art of creating machines that perform functions that require intelligence when performed by people." (Kurzweil, 1990) "The study of how to make computers do things at which, at the moment, people are better." (Rich and Knight, 1991)	"Computational Intelligence is the study of the design of intelligent agents." (Poole et al., 1998) "AI . . . is concerned with intelligent behavior in artifacts." (Nilsson, 1998)

Figure 3.1 Some definitions of artificial intelligence, organized into four categories.

3.1 Acting humanly: The Turing Test approach

The Turing Test, proposed by Alan Turing (1950), was designed to provide a satisfactory operational definition of intelligence. A computer passes the test if a human interrogator, after posing some written questions, cannot tell whether the written responses come from a person or from a computer. For now, we note that programming a computer to pass a rigorously applied test provides plenty to work on. The computer would need to possess the following capabilities:

- natural language processing to enable it to communicate successfully in English;
- knowledge representation to store what it knows or hears;
- automated reasoning to use the stored information to answer questions and to draw new conclusions;
- machine learning to adapt to new circumstances and to detect and extrapolate patterns.

Turing's test deliberately avoided direct physical interaction between the interrogator and the computer, because physical simulation of a person is unnecessary for intelligence. However, the so-called total Turing Test includes a video signal so that the interrogator can test the subject's perceptual abilities, as well as the opportunity for the interrogator to pass physical objects "through the hatch." To pass the total Turing Test, the computer will need

- computer vision to perceive objects, and
- robotics to manipulate objects and move about.

These six disciplines compose most of AI, and Turing deserves credit for designing a test that remains relevant 60 years later.

3.2 Thinking humanly: The cognitive modeling approach

If we are going to say that a given program thinks like a human, we must have some way of determining how humans think. We need to get inside the actual workings of human minds. There are three ways to do this: through introspection—trying to catch our own thoughts as they go by; through psychological experiments—observing a person in action; and through brain imaging—observing the brain in action. Once we have a sufficiently precise theory of the mind, it becomes possible to express the theory as a computer program. If the program's input–output behavior matches corresponding human behavior, that is evidence that some of the program's mechanisms could also be operating in humans. For example, Allen Newell and Herbert Simon, who developed GPS, the "General Problem Solver" (Newell and Simon, 1961), were not content merely to have their program solve problems correctly. They were more concerned with comparing the trace of its reasoning steps to traces of human subjects solving the same problems. The interdisciplinary field of cognitive science brings together computer models from AI and experimental techniques from psychology to construct precise and testable theories of the human mind.

In the early days of AI there was often confusion between the approaches: an author would argue that an algorithm performs well on a task and that it is therefore a good model of human performance, or vice versa. Modern authors separate the two kinds of claims; this distinction has allowed both AI and cognitive science to develop more rapidly. The two fields continue to fertilize each other, most notably in computer vision, which incorporates neurophysiological evidence into computational models.

3.3 Thinking rationally: The "laws of thought" approach

The Greek philosopher Aristotle was one of the first to attempt to codify "right thinking," that is, irrefutable reasoning processes. His syllogisms provided patterns for argument structures that always yielded correct conclusions when given correct premises—for example, "Socrates is a man; all men are mortal; therefore, Socrates is mortal." These laws of thought were supposed to govern the operation of the mind; their study initiated the field called logic.

Logicians in the 19th century developed a precise notation for statements about all kinds of objects in the world and the relations among them. (Contrast this with ordinary arithmetic notation, which provides only for statements about numbers.) By 1965, programs existed that could, in principle, solve any solvable problem described in logical notation. (Although if no solution exists, the program might loop forever.) The so-called logicist tradition within artificial intelligence hopes to build on such programs to create intelligent systems.

There are two main obstacles to this approach. First, it is not easy to take informal knowledge and state it in the formal terms required by logical notation, particularly when the knowledge is less than 100% certain. Second, there is a big difference between solving a problem "in principle" and solving it in practice. Even problems with just a few hundred facts can exhaust the computational resources of any computer unless it has some guidance as to which reasoning steps to try first. Although both of these obstacles apply to any attempt to build computational reasoning systems, they appeared first in the logicist tradition.

3.4 Acting rationally: The rational agent approach

An agent is just something that acts (agent comes from the Latin agere, to do). Of course, all computer programs do something, but computer agents are expected to do more: operate autonomously, perceive their environment, persist over a prolonged time period, adapt to change, and create and pursue goals. A rational agent is one that acts so as to achieve the best outcome or, when there is uncertainty, the best expected outcome.

In the "laws of thought" approach to AI, the emphasis was on correct inferences. Making correct inferences is sometimes part of being a rational agent, because one way to act rationally is to reason logically to the conclusion that a given action will achieve one's goals and then to act on that conclusion. On the other hand, correct inference is not all of rationality; in some situations, there is no provably correct thing to do, but something must still be done. There are also ways of acting rationally that cannot be said to involve inference. For example, recoiling from a hot stove is a reflex action that is usually more successful than a slower action taken after careful deliberation.

All the skills needed for the Turing Test also allow an agent to act rationally. Knowledge representation and reasoning enable agents to reach good decisions. We need to be able to generate comprehensible sentences in natural language to get by in a complex society. We need learning not only for erudition, but also because it improves our ability to generate effective behavior.

The rational-agent approach has two advantages over the other approaches. First, it is more general than the "laws of thought" approach because correct inference is just one of several possible mechanisms for achieving rationality. Second, it is more amenable to scientific development than are approaches based on human behavior or human thought. The standard of rationality is mathematically well defined and completely general, and can be "unpacked" to generate agent designs that provably achieve it. Human behavior, on the other hand, is well adapted for one specific environment and is defined by, well, the sum total of all the things that humans do. This book therefore concentrates on general principles of rational agents and on components for constructing them. We will see that despite the apparent simplicity with which the problem can be stated, an enormous variety of issues come up when we try to solve it.

One important point to keep in mind: We will see before too long that achieving perfect rationality—always doing the right thing—is not feasible in complicated environments. The computational demands are just too high. For most of the study material, however, we will adopt the working hypothesis that perfect rationality is a good starting point for analysis. It simplifies the problem and provides the appropriate setting for most of the foundational material in the field.

4.0 Conclusion

In this chapter, we define AI and discuss the founding principles behind the various approaches adopted towards it. We also pointed out advantages and weakness of some of the approaches.

5.0 Summary

We hope you enjoyed this unit. This unit provided definition of AI and different AI paradigms. Now, let us attempt the questions below.

6.0 Tutor Marked Assignment

i. Give a comprehensive definition of AI with respect to its various categories
ii. Define in your own words: (a) intelligence, (b) artificial intelligence, (c) agent, (d) rationality, (e) logical reasoning.
iii. Outline the six major disciplines of AI
iv. Discuss the four approaches to AI.

7.0 References/Further Readings

1.0 Russell, S., & Norvig, P. (2005). AI a modern approach. Learning, 2(3), 4.
2.0 Rich E., & Knight K. (1991) Artificial Intelligence, McGraw-Hill, New York, NY
3.0 Haugeland, J. (Ed.). (1985). Artificial Intelligence: The Very Idea. MIT Press.
4.0 Charniak, E. and McDermott, D. (1985). Introduction to Artificial Intelligence. Addison-Wesley.
5.0 Bellman, R. E. (1978). An Introduction to Artificial Intelligence: Can Computers Think? Boyd & Fraser Publishing Company.
6.0 Winston, P. H. (1992). Artificial Intelligence (Third edition). Addison-Wesley.
7.0 Kurzweil, R. (1990). The Age of Intelligent Machines. MIT Press.
8.0 Rich, E. and Knight, K. (1991). Artificial Intelligence (second edition). McGraw-Hill.
9.0 Nilsson, N. J. (1998). Artificial Intelligence: A New Synthesis. Morgan Kaufmann.
10.0 Alan Turing (1950
11.0 Newell, A. and Simon, H. A. (1961). GPS, a program that simulates human thought. In Billing, H. (Ed.), Lernende Automaten, pp. 109–124. R. Oldenbourg.
12.0 Poole, D.,Mackworth, A. K., and Goebel, R. (1998). Computational intelligence: A logical approach. Oxford University Press.

Unit Two: The State of Art in AI

CONTENT

1.0 Introduction

What can AI do today? A concise answer is difficult because there are so many activities in so many subfields. Here we sample a few applications; others appear throughout the study material.

2.0 Objectives

At the end of this unit, you should be able to;

- Outline a few examples of artificial intelligence systems that exist today.

3.0 Main Content

3.1 Robotic Vehicles

A driverless robotic car named STANLEY sped through the rough terrain of the Mojave dessert at 22 mph, finishing the 132-mile course first to win the 2005 DARPA Grand Challenge. STANLEY is a Volkswagen Touareg outfitted with cameras, radar, and laser rangefinders to sense the environment and onboard software to command the steering, braking, and acceleration (Thrun, 2006). The following year CMU's BOSS won the Urban Challenge, safely driving in traffic through the streets of a closed Air Force base, obeying traffic rules and avoiding pedestrians and other vehicles.

3.2 Speech recognition:

A traveler calling United Airlines to book a flight can have the entire conversation guided by an automated speech recognition and dialog management system.

3.3 Autonomous planning and scheduling:

A hundred million miles from Earth, NASA's Remote Agent program became the first on-board autonomous planning program to control the scheduling of operations for a spacecraft (Jonsson et al., 2000). REMOTE AGENT generated plans from high-level goals specified from the ground and monitored the execution of those plans—detecting, diagnosing, and recovering from problems as they occurred. Successor program MAPGEN (Al-Chang et al., 2004) plans the daily operations for NASA's Mars Exploration Rovers, and MEXAR2 (Cesta et al., 2007) did mission planning—both logistics and science planning—for the European Space Agency's Mars Express mission in 2008.

3.4 Game playing:

IBM's DEEP BLUE became the first computer program to defeat the world champion in a chess match when it bested Garry Kasparov by a score of 3.5 to 2.5 in an exhibition match (Goodman and Keene, 1997). Kasparov said that he felt a "new kind of intelligence" across the board from him. Newsweek magazine described the match as "The brain's last stand." The value of IBM's stock increased by $18 billion. Human champions studied Kasparov's loss and were able to draw a few matches in subsequent years, but the most recent human-computer matches have been won convincingly by the computer.

3.5 Logistics planning

During the Persian Gulf crisis of 1991, U.S. forces deployed a Dynamic Analysis and Replanning Tool, DART (Cross and Walker, 1994), to do automated logistics planning and scheduling for transportation. This involved up to 50,000 vehicles, cargo, and people at a time, and had to account for starting points, destinations, routes, and conflict resolution among all parameters. The AI planning techniques generated in hours a plan that would have taken weeks with older methods. The Defense Advanced Research Project Agency (DARPA) stated that this single application more than paid back DARPA's 30-year investment in AI.

3.6 Robotics

The iRobot Corporation has sold over two million Roomba robotic vacuum cleaners for home use. The company also deploys the more rugged PackBot to Iraq and Afghanistan, where it is used to handle hazardous materials, clear explosives, and identify the location of snipers.

3.7 Machine Translation

A computer program automatically translates from Arabic to English, allowing an English speaker to see the headline "Ardogan Confirms That Turkey Would Not Accept Any Pressure, Urging Them to Recognize Cyprus." The program uses a statistical model built from examples of Arabic-to-English translations and from examples of English text totaling two trillion words (Brants et al., 2007). None of the computer scientists on the team speak Arabic, but they do understand statistics and machine learning algorithms.

4.0 Conclusion

These are just a few examples of artificial intelligence systems that exist today. Not magic or science fiction—but rather science, engineering, and mathematics, to which this material provides an introduction.

5.0 Summary

We hope you enjoyed this unit. This unit mentions a few examples of artificial intelligence systems that exist today. Now, let us attempt the questions below.

6.0 Tutor Marked Assignment

i. Briefly discuss five (5) state of art applications of AI.

7.0 References/Further Readings

1.0 Russell, S., & Norvig, P. (2005). AI a modern approach. Learning, 2(3), 4.

2.0 Thrun, S. (2006). Stanley, the robot that won the DARPA Grand Challenge. J. Field Robotics, 23(9), 661–692.

3.0 Jonsson, A., Morris, P., Muscettola, N., Rajan, K., and Smith, B. (2000). Planning in interplanetary space: Theory and practice. In AIPS-00, pp. 177–186.

4.0 Al-Chang, M., Bresina, J., Charest, L., Chase, A., Hsu, J., Jonsson, A., Kanefsky, B.,Morris, P., Rajan, K., Yglesias, J., Chafin, B., Dias,W., and Maldague, P. (2004). MAPGEN: Mixed-Initiative planning and scheduling for the Mars Exploration Rover mission. IEEE Intelligent Systems, 19(1), 8–12.

5.0 Cesta, A., Cortellessa, G., Denis, M., Donati, A., Fratini, S., Oddi, A., Policella, N., Rabenau, E., and Schulster, J. (2007). MEXAR2: AI solves mission planner problems. IEEE Intelligent Systems, 22(4), 12–19.

6.0 Goodman, D. and Keene, R. (1997). Man versus Machine: Kasparov versus Deep Blue. H3 Publications. Cross and Walker, 1994

7.0 Brants, T., Popat, A. C., Xu, P., Och, F. J., and Dean, J. (2007). Large language models in machine translation. In EMNLP-CoNLL-2007: Proc. 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning, pp. 858–867.

Unit Three: AI Programming Languages

CONTENT

1.0 Introduction

A number of programming languages exist that are used to build Artificial Intelligence systems. General programming languages such as C++ and Java are often used because these are the languages with which most computer scientists have experience. There also exist two programming languages that have features that make them particularly useful for programming Artificial Intelligence projects—PYTHON, PROLOG and LISP.

2.0 Objectives

At the end of this unit, you should be able to;

- Outline the three most prominent Programming Languages used in AI research.
- Identify the special features of PYTHON, PROLOG and LISP that make them prominently used for programming Artificial Intelligence projects.

3.0 Main Content

We will now provide a brief overview of these two languages and explain how they are used in Artificial Intelligence research. Of course, a number of other programming languages exist that are also widely used for Artificial Intelligence, but we will focus on PYTHON, PROLOG and LISP because these are certainly the most widely used and the ones on which there is the widest range of relevant literature.

3.1 PYTHON

Python is one of the most popular programming languages used by developers today. Guido Van Rossum created it in 1991 and ever since its inception has been one of the most widely used languages along with C++, Java, etc.

3.1.1 Features and Advantages of Python

Python is an Interpreted language which in lay man's terms means that it does not need to be compiled into machine language instruction before execution and can be used by the developer directly to run the program. This makes it comprehensive enough for the language to be interpreted by an emulator or a virtual machine on top of the native machine language which is what the hardware understands.

It is a High-Level Programing language and can be used for complicated scenarios. High-level languages deal with variables, arrays, objects, complex arithmetic or Boolean expressions, and other abstract computer science concepts to make it more comprehensive thereby exponentially increasing its usability.

Python is also a General-purpose programming language which means it can be used across domains and technologies. Python also features dynamic type system and automatic memory management supporting a wide variety of programming paradigms including object-oriented, imperative, functional and procedural to name a few. Python is available for all Operating Systems and also has an open-source offering titled CPython which is garnering widespread popularity as well.

Let us now look as to how using Python for Artificial Inelegance gives us an edge over other popular programming languages.

3.1.2 AI and Python: Why?

The obvious question that we need to encounter at this point is why we should choose Python for AI over others. Python offers the least code among others and is in fact 1/5 the number compared to other OOP languages. No wonder it is one of the most popular in the market today.

- Python has Prebuilt Libraries like Numpy for scientific computation, Scipy for advanced computing and Pybrain for machine learning (Python Machine Learning) making it one of the best languages For AI.
- Python developers around the world provide comprehensive support and assistance via forums and tutorials making the job of the coder easier than any other popular languages.
- Python is platform Independent and is hence one of the most flexible and popular choiceS for use across different platforms and technologies with the least tweaks in basic coding.
- Python is the most flexible of all others with options to choose between OOPs approach and scripting. You can also use IDE itself to check for most codes and is a boon for developers struggling with different algorithms.

3.1.3 Decoding Python alongside AI

Python along with packages like NumPy, scikit-learn, iPython Notebook, and matplotlib form the basis to start your AI project.

- NumPy is used as a container for generic data comprising of an N-dimensional array object, tools for integrating C/C++ code, Fourier transform, random number capabilities, and other functions.
- Another useful library is pandas, an open source library that provides users with easy-to-use data structures and analytic tools for Python.
- Matplotlib is another service which is a 2D plotting library creating publication quality figures. You can use matplotlib to up to 6 graphical users interface toolkits, web application servers, and Python scripts.
- Your next step will be to explore k-means clustering and also gather knowledge about decision trees, continuous numeric prediction, logistic regression, etc.

Some of the most commonly used Python AI libraries are AIMA, pyDatalog, SimpleAI, EasyAi, etc. There are also Python libraries for machine learning like PyBrain, MDP, scikit, PyML. Let us look a little more in detail about the various Python libraries in AI and why this programming language is used for AI.

3.1.4 Python Libraries for General AI

- AIMA – Python implementation of algorithms from Russell and Norvig's 'Artificial Intelligence: A Modern Approach.'
- pyDatalog – Logic Programming engine in Python
- SimpleAI – Python implementation of many of the artificial intelligence algorithms described on the book "Artificial Intelligence, a Modern Approach". It focuses on providing an easy to use, well documented and tested library.
- EasyAI – Simple Python engine for two-players games with AI (Negamax, transposition tables, game solving).

3.2 PROLOG

PROLOG (PROgramming in LOGic) is a language designed to enable programmers to build a database of facts and rules, and then to have the system answer questions by a process of logical deduction using the facts and rules in the database.

Facts entered into a PROLOG database might look as follows:

tasty (cheese).
made_from (cheese, milk).
contains (milk, calcium).

These facts can be expressed as the following English statements:

Cheese is tasty.

Cheese is made from milk.

Milk contains calcium.

We can also specify rules in a similar way, which express relationships between objects and also provide the instructions that the PROLOG theorem prover will use to answer queries. The following is an example of a rule in PROLOG:

contains (X, Y) :- made_from (X, Z), contains (Z, Y).

This rule is made up of two main parts, separated by the symbol ":-".

The rule thus takes the form:

B :- A

which means "if A is true, then B is true," or "A implies B."

Hence, the rule given above can be translated as "If X is made from Z and Z contains Y then X contains Y."

Having entered the three facts and one rule given above, the user might want to ask the system a question:

?- contains (cheese, calcium).

Using a process known as resolution, the PROLOG system is able to use the rule and the facts to determine that because cheese is made from milk, and because milk contains calcium, therefore cheese does contain calcium. It thus responds:

yes

It would also be possible to ask the system to name everything that contains calcium:

?- contains (X, calcium)

The system will use the same rules and facts to deduce that milk and cheese both contain calcium, and so will respond:

X=milk.
X=cheese.

This has been a very simple example, but it should serve to illustrate how PROLOG works. Far more complex databases of facts and rules are routinely built using PROLOG, and in some cases simple databases are built that are able to solve complex mathematical problems.

PROLOG is not an efficient programming language, and so for many problems a language such as C++ would be more appropriate. In cases where logical deduction is all that is required, and the interactive nature of the PROLOG interface is suitable, then PROLOG is the clear choice. PROLOG provides a way for programmers to manipulate data in the form of rules and facts without needing to select algorithms or methodologies for handling those data.

3.3 LISP

LISP (LISt Programming) is a language that more closely resembles the imperative programming languages such as C++ and Pascal than does PROLOG. As its name suggests, LISP is based around handling of lists of data. A list in LISP is contained within brackets, such as:

[A B C]

This is a list of three items. LISP uses lists to represent data, but also to represent programs. Hence, a program in LISP can be treated as data. This introduces the possibility of writing self-modifying programs in LISP, it also allows us to use evolutionary techniques to "evolve" better LISP programs.

LISP is a far more complex language syntactically than PROLOG, and so we will not present any detail on its syntax here. It provides the usual kinds of mechanisms that other programming languages provide, such as assignment, looping, evaluating functions, and conditional control (if. . . then. . .). It also provides a great deal of list manipulation functions, such as car and cdr, which are used to return the first entry in a list and all the entries except for the first entry, respectively.

4.0 Conclusion

We have given an overview of two languages (PROLOG and LISP) and explained how they are used in Artificial Intelligence research.

5.0 Summary

We hope you enjoyed this unit. This unit mentions the unique attributes of two programming languages that are commonly used for artificial intelligence systems. Now, let us attempt the questions below.

6.0 Tutor Marked Assignment

i. Why are PROLOG and LISP so well suited to Artificial Intelligence research? Do you think languages such as C++ and Java could also be used for such research?

7.0 References/Further Readings

1) Coppin, B. (2004). Artificial intelligence illuminated. Jones & Bartlett Learning.Python Tutorial for Beginners | How to Quickly Learn Python? https://data-flair.training/blogs/python-tutorial/ Retrieved 07/29/2019
2) Lucas, P., & Van Der Gaag, L. (1991). Principles of expert systems. Wokingham: Addison-Wesley.

Unit Four: Types of AI

CONTENT

1.0 Introduction

Artificial Intelligence can be divided in various types, there are mainly two types of main categorization which are based on capabilities and based on functionally of AI.

2.0 Objectives

At the end of this unit, you should be able to;

- Outline the two categories of AI
- Understand the reason and concept behind each category

3.0 Main Content

Figure 3.1 presents a flow diagram which explain the types of AI.

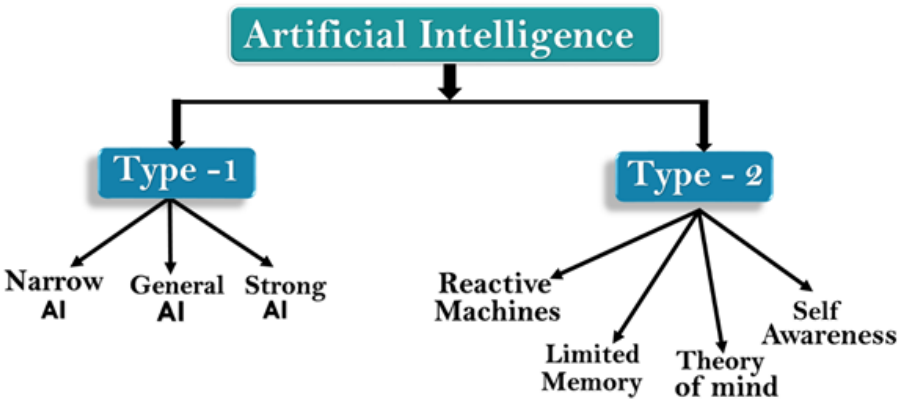


Figure 3.1: Categories of AI

3.1 AI type-1: Based on Capabilities

3.1.1 Narrow AI

Narrow AI is a type of AI which is able to perform a dedicated task with intelligence. The most common and currently available AI is Narrow AI in the world of Artificial Intelligence. Narrow AI cannot perform beyond its field or limitations, as it is only trained for one specific task. Hence it is also termed as weak AI. Narrow AI can fail in unpredictable ways if it goes beyond its limits.

Apple Siriis a good example of Narrow AI, but it operates with a limited pre-defined range of functions. IBM's Watson supercomputer also comes under Narrow AI, as it uses an Expert system approach combined with Machine learning and natural language processing. Some Examples of Narrow AI are playing chess, purchasing suggestions on e-commerce site, self-driving cars, speech recognition, and image recognition.

3.1.2 General AI

General AI is a type of intelligence which could perform any intellectual task with efficiency like a human. The idea behind the general AI is to make such a system which could be smarter and think like a human by its own.

Currently, there is no such system that exist which could perfectly come under general AI and can perform any task as perfect as a human. The worldwide researchers are now focused on developing machines with General AI. As such, systems with general AI are still under research, and it will take lots of efforts and time to develop such systems.

3.1.3 Super AI

Super AI is a level of Intelligence of Systems at which machines could surpass human intelligence, and can perform any task better than human with cognitive properties. It is an outcome of general AI.

Some key characteristics of strong AI include capability include the ability to think, to reason, solve the puzzle, make judgments, plan, learn, and communicate by its own. Super AI is still a hypothetical concept of Artificial Intelligence. Development of such systems in real is still world changing task.

3.2 Artificial Intelligence type-2: Based on functionality

3.2.1 Reactive Machines

Purely reactive machines are the most basic types of Artificial Intelligence. Such AI systems do not store memories or past experiences for future actions. These machines only focus on current scenarios and react on it as per possible best action.

IBM's Deep Blue system is an example of reactive machines. Google's AlphaGo is also an example of reactive machines.

3.2.2 Limited Memory

Limited memory machines can store past experiences or some data for a short period of time. These machines can use stored data for a limited time period only. Self-driving cars are one of the best examples

of Limited Memory systems. These cars can store recent speed of nearby cars, the distance of other cars, speed limit, and other information to navigate the road.

3.2.3 Theory of Mind

Theory of Mind AI should understand the human emotions, people, beliefs, and be able to interact socially like humans. This type of AI machines are still not developed, but researchers are making lots of efforts and improvement for developing such AI machines.

3.2.4 Self-Awareness

Self-awareness AI is the future of Artificial Intelligence. These machines will be super intelligent, and will have their own consciousness, sentiments, and self-awareness. These machines will be smarter than human mind. Self-Awareness AI does not exist in reality still and it is a hypothetical concept.

4.0 Conclusion

This unit presents a high-level categorization of work in AI based on their capabilities and functionality. Examples were also given in cases where they exist. Students should be able to correctly identify the category of any AI system they come across.

5.0 Summary

We hope you enjoyed this unit. This unit mentions types of AI and their categories. Now, let us attempt the questions below.

6.0 Tutor Marked Assignment

i. How are AI systems categorized?
ii. Give an example for each category.

7.0 References/Further Readings

1. Types of Artificial Intelligence. At https://www.javatpoint.com/types-of-artificial-intelligence, retrieved on 7/24/19.
2. Coppin, B. (2004). Artificial intelligence illuminated. Jones & Bartlett Learning.

MODULE 2: BASIC AI ISSUES

Unit One: Attention
Unit Two: Search and Control,
Unit Three: Game Trees,
Unit Four: Knowledge Representation

Unit One: Attention,

CONTENT

1.0 Introduction

In the domain of intelligent systems, the management of system resources is typically called “attention”. Attention mechanisms exist because even environments of moderate complexity are a source of vastly more information than available cognitive resources of any known intelligence can handle. Cognitive resource management has not been of much concern in artificial intelligence (AI) work that builds relatively simple systems for particular targeted problems. For systems capable of a wide range of actions in complex environments, explicit management of time and cognitive resources is not only useful, it is a necessity.

2.0 Objectives

At the end of this unit, you should be able to;

- Understand how Attention relates to AI
- Outline the three realities every AI system needs to tackle in order to address implement attention
- Understand the importance of Attention to AI systems
- Discuss how attention is implemented in Narrow AI systems

- Outline the four (4) methods Narrow AI combines to implement Attention
- Discuss how attention is implemented in AGI systems
- Outline the four types of environments that must be targeted for AGI systems
- Discuss how attention is implemented in some Artificial Attention Systems
- Discuss how Attention is implemented in some selected Natural Attention Systems

3.0 Main Content

The field of AI has a long history of targeting isolated, well-defined problems to demonstrate intelligent capabilities. While useful, many of these problems (and especially their task environments, as perceived by the system) are extremely simple compared to the problem of learning how to solve novel tasks and adapting to changes in real-world environments - a problem which must be addressed and solved in order for AI systems to approach human-level intelligence. Given the nature of this prior work, it is not surprising that limited focus has been given to real-time processing and resource management. However, the design of any AI system expected to learn and perform a range of tasks in everyday environments needs to face these realities:

- The real world is highly dynamic and complex and can provide an abundance of information at any given moment.
- Resources of any intelligent system are not only limited, but insufficient in light of the massive amount of information available from the environment.
- A range of time constraints, many of which are dictated by the environment, must be satisfied in order to ensure safe and successful operation of the system.

Much of existing work in the field of AI is also based on greatly simplified operating assumptions - a case in point being the practically impossible (but surprisingly common) assumption of infinite resources, often in terms of storage but particularly in terms of processing: A system based on this assumption will fail to perform and potentially crash in real world operation when fed with information at a greater rate than it is capable of processing. To find inspiration for implementing intelligent resource management we need not look far, nature has provided us with a prime example in human attention; a cognitive function that enables us to focus our limited resources selectively on information that is most important to us at any given moment as we perform various tasks while remaining reactive to unexpected but important events in the environment. Consider that while reading this unit, you have effectively ignored more than 99.9% of the numerous things that your mind could have spent time and resources on doing.

Perhaps not surprisingly, it turns out that this is exactly the kind of resource management that is required to enable AI systems to approach human-level intelligence in real-world environments. Thus, it makes perfect sense to investigate how AI systems can be endowed with this cognitive function for the purpose of improving their operation and making them applicable to more open-ended and complex tasks and environments. The goal need not be to replicate any biological function in detail, but rather to extract useful concepts and methods from the biological side while leaving undesirable limitations behind in order to facilitate the creation of AI systems that can successfully operate in real-world environments in real-time using limited resources.

3.1 Narrow AI and Attention

For decades now, narrow AI systems have been successfully deployed in industry without being designed to have any special attention capabilities. How can these systems solve real problems in complex environments, many of which generate more information than said systems could ever hope to process in real-time, yet are necessary for them to perform their tasks, when their design does not take attentional functions into account?

To answer this question, let us consider fundamentally what a narrow AI system is. Such a system is purpose-built for certain specified tasks and environments that are not expected to vary significantly, hence the term “narrow”. An implication of this is that once the tasks and environments the system has to deal with are specified, a great deal is known about what kind of information will be useful for the system to process in order to make decisions and what kind of information can be safely ignored.

Consider the following case:

A chess-playing system is designed for an environment consisting of a discrete 8-by-8 grid, each cell being in one of a finite set of states at any given time. Such a system can effectively ignore its surrounding real-world environment as nothing outside of the chessboard is relevant; there is no need to process information from any human-like modalities (vision, hearing, etc.). As the task of playing chess is fully pre-specified by the rules of the game and the structure of the game board, there is no chance for these modalities or information coming from other sources to ever become relevant to the system. Furthermore, any possibility that new states will at some point be added to the set of possible states is precluded, as the rules of the game (and thus the operational requirements of the system) are fully pre-specified and static. A new type of chess piece is never expected to appear on the board and new ways to move chess pieces will never be allowed for. The end result is that the chess-playing system operates in a closed world; it is never required to learn about new entities or new fundamental ways of perceiving or acting in the environment. Any learning performed by such a system targets ways to effect and react to this closed deterministic environment with the goal of improving performance, measured for example by the ratio of games won. The chess-playing task is likely to include time constraints, but these are also specified in advance as part of the rules of the game and are static in nature. The environment will not change while the system is taking its turn in the game; any reaction to the environment beyond taking turn within some pre-specified time limit is precluded.

In the chess-playing example, the environment provides a very small amount of information (the minimum for encoding the state of the board is 192 bits). As the game proceeds, the environment changes only when each player takes turn and the each change is small; with each move no more than 6 bits (the state of two squares) of information can change. While the state-space of the game is huge (upper-bounded by 647), perception and action processing for this task are simple and do not require information filtering or prioritization. Resource management may be required to determine the next move of the system, but this only applies to internal processing and is solely controlled by the amount time allowed for when deciding the next turn. During each move, the maximum amount of time available for action decision is known in advance, greatly simplifying internal resource management as opposed to an interruptible resource management scheme.

While the chess environment has low complexity by any measure, many existing narrow AI systems deal directly with real world environments. The following presents an example of such a system.

In a video security surveillance system, the task at hand is to detect humans and attempt to identify them. Sensory input to the system consists of video streams from several cameras, each targeting different parts of the target real-world environment that the system is meant to monitor. Let us assume that the system has to monitor 20 such video feeds where each video frame is a 720p image and each feed provides 24 such frames per second. This results in a sensory stream of roughly 1.3 GB of information per second, clearly a substantial amount of information to apply complex processing to in real-time. However, as the operational requirements of the system are static and known at design time, it is possible to greatly reduce incoming information very early in the sensory pipeline by immediately searching every new frame for features that indicate the presence of a human, for example, using well-known computer vision techniques (e.g. Haar cascade classifiers (Viola 2001)).

These features, once detected and extracted, could then form a basis for identifying the particular individual. At no time will such a system be expected to recognize novel features, such as finding a new type of garment worn and classify it in the context of previously seen garments, unless explicitly programmed to do so. In any case, any and all information that does not imply the presence of a human is irrelevant to the system and may be immediately discarded after initial processing as it will have no impact on the operation of the system. Assuming that there is a 0.1 probability that there is a human in each frame of video, and that when detected, the features necessary to identify the individual are roughly 1/8 the amount of information contained in a single frame, the sensory stream of the entire system amounts to a mere 16,5 MB per second.

The effects of designing static attention into the system, made possible by detailed specifications at design time and implemented by focusing the resources of the system towards information known to be relevant, results in an 80-fold decrease in the input stream of the system, making its task significantly easier to accomplish without any form of, dynamic or otherwise, advanced resource management. The resource requirements of the system are highly constant and predictable at design time. It is worth reiterating that this kind of reduction in complexity could not have been achieved without the existence of detailed pre-specified operational requirements of the system. Any time constraints that the system shall meet (e.g. performing recognition of a newly appeared individual within 2 seconds) can be addressed by optimizing code or adjusting the hardware resources of the system to fit expected resource requirements.

This example demonstrates how a narrow AI system can superficially appear to be dealing with real world environments, while they are in fact dealing with greatly simplified and filtered representations of such environments, with the representations being narrowly dictated by the operating requirements and limited, pre-determined tasks. It is left to the reader to extend this idea to other examples of narrow AI, such as:

- Routing emails and cell phone calls
- Automated image-based medical diagnosis
- Guidance for cruise missiles and weapon systems
- Automatically landing airplanes
- Financial pattern recognition

- Detection of credit card fraud

When a complete specification of tasks and environment exists, the operating environment of the system becomes a closed world consisting only of task-relevant information.

Narrow AI systems have – in a sense – a tunnel vision view on the environment, with static fixation points. A complete specification of task-relevant information can be derived from a complete operating specification without much effort. As a result, the attention of the system can be manually implemented at design and implementation time (as seen in the example above), with the concrete implementation being that the system processes particular information coming from particular types of physical or artificial sensors, while ignoring others known to be irrelevant – all dictated by the operating specification and pre-defined tasks. This results in an enormous reduction in the complexity and amount of information that the system needs to deal with, in contrast to constantly perceiving through all possible sensory channels in the target environment. Importantly, the frequency of which the environment needs to be sampled by the system (rate of incoming sensory information), and time constraints involved with the target tasks, may also be derived from the specification in the same fashion.

3.2 Artificial General Intelligence (AGI) and Attention

Moving beyond narrow AI systems to AGI systems requires some fundamental thought be given to the meaning of intelligence. It is no longer sufficient to work from a vague definition of the phenomenon. While there is no single widely accepted definition of intelligence, anyone doing research in the field of AI needs to choose his or her definition in order to specify research goals, engineering requirements, and to evaluate progress.

Various Researchers have made in-depth discussions of competing definitions for intelligence, some well-known (but not universally accepted) examples include:

- Passing the Turing test. (Turing, 1948)
- Behavior in real situations that is appropriate and adaptive to the needs of the system and demands of the environment. (Newell & Simon, 1976)
- The ability to solve hard problems, without any explicit consideration of time. (Minsky, 1985)
- Achieving goals in situations where available information has complex characteristics. (McCarthy, 1988)

However, one of the acceptable and relevant definitions is:

“Intelligence, as the experience-driven form of adaptation, is the ability of an information system to achieve its goals with insufficient knowledge and resources.” (Wang 2013: p. 16)

The distinction between narrow AI and AGI is very important with regards to attention. In the case of narrow AI systems, the task and operating environment are known (or mostly known) at design time. In such systems the world is mostly closed, in the sense that everything the system will ever need to know about is known at design time (in an ontological sense). While the operation of the system may involve learning, exactly what is to be learned is also specified in detail at design time. Using the specification of the task, narrow AI systems can implement attention by combining the following four (4) methods:

- Completely ignoring modalities (in a general sense, i.e. data streams) that are available yet irrelevant to the task as specified.
- Filtering data for characteristics that are known, at design time, to be task relevant.
- Sampling the environment at appropriate frequencies (typically the minimum frequency that still allows for acceptable performance).
- Making decisions to act at predetermined frequencies that fit the task as specified.

A combination of these methods could allow narrow AI systems to effectively filter incoming information to deal with information overload, as well as being alert to predefined interrupts. As the task and environment are known, operational boundaries are also known to some extent, including boundaries with regards to how much information the system will be exposed to. A fixed type of attention based on the methods described above, along with proper allocation of hardware resources, would be sufficient for most narrow AI systems.

The previous section discussed examples of narrow AI tasks. In contrast, in AGI systems the luxury of knowing these things beforehand is out of question – by design and requirement. To illustrate, the following is an example of an AGI-level task in a real-world environment:

Let us imagine an exploration robot that can be deployed, without special preparation, into virtually any environment, and move between them without serious problems. The various environments the robot may encounter can vary significantly in dynamics and complexity; they can be highly invariable like the surface of Mars or the Sahara Desert and dynamic like the Amazon jungle and the vast depths of the ocean. We assume the robot is equipped with a number of actuators and sensors and is designed to physically withstand the ambient environmental conditions of these environments.

It has some general pre-programmed knowledge, but is not given mission-specific knowledge prior to deployment, only high-level goals related to exploration, and neither it nor its creators know beforehand which environment(s) may be chosen or how they may change after deployment. For the purposes of this example, missions are assumed to be time constrained but otherwise open-ended. The robot has the goal of exploration, which translates into learning about the environment, through observation and action.

Immediately upon deployment, the robot thus finds itself in unfamiliar situations in which it has little or no knowledge of how to operate. Abilities of adaption and reactiveness are critical requirements as the environment may contain numerous threats which must be handled in light of the robot's persistent goal of survival. Specific actuators may function better than others in certain environments, for example when moving around or manipulating objects, and this must be learned by the robot as quickly as possible. Resource management is a core problem, as the robot's resources are limited.

Resources include energy, processing capacity, and time: Time is not only a resource in terms of the fixed mission duration, but at lower levels as well since certain situations, especially ones involving threats, have inherent deadlines on action. The resource management scheme must be highly dynamic as unexpected events that require action (or inaction) can occur at any time. (Thórisson & Helgason 2012, p. 4)

This example represents a case where the benefits of having a detailed operational specification at design time are not available. The goals of the AI system's design are expressed at a high level of abstraction, precluding such a specification. Here the methods for reducing information and complexity

for narrow AI systems, discussed above, do not help. For the exploration robot to accomplish its high-level goals, any of its sensory information may be relevant. At the same time, its resources are limited; giving equal treatment to all information is not practically possible. Goals specified at a high level of abstraction are not unique to this example; they are a unifying feature of all AGI systems.

Such systems must learn to accomplish their own (high-level) goals by relating them to their sensory experience as collected in complex, real-world environments. Already several references to "real-world" environments have been made. Some clarification is in order to disambiguate this concept. Researchers have built upon the work of Russell & Norvig (2003) in classifying environments for AI agents. The following discusses each of the environmental properties proposed by them in the case of the target and real-world environments.

1) Fully observable / Partially observable

This property is not critical to what is considered a real-world environment, but does raise an important issue. It is undesirable to limit the focus to the three-dimensional environments that people live their lives in and sense in a very particular way, a result of the biological sensory system of humans. Such environments can be abstracted to environments where the agent/human must perform proactive, goal-directed sensing, meaning that not all aspects of the environment are observable simultaneously at any given time. If particular aspects of the environment are not observable, reorienting sensors (as allowed for by the mobility of the system) can make other aspects of the environment observable. However, in the process of making new things observable the scope of what was observable before may change. Additionally, a partially observable environment does not imply that the environment is fully observable if all possible agent positions and sensor orientations were somehow simultaneously possible, as there may be aspects of the environment that are relevant to the agent but can never be observed directly.

Environments where all information is visible at any time would be called "fully observable" by Russell & Norvig. But this definition becomes less clear when we consider systems that perform active sensing where the system decides what senses to sample, and at what temporal frequency. One reason active sensing may be desirable is that real world environments contain such enormous amounts of information, that while in theory a system could observe the entire environment, practical issues such as available resources would make this completely impossible, as perception – even of just a small aspect of the environment – may demand significant processing resources. Consider also that time may be so fine-grained in the operating environments that no system will attempt to, or be able to, sense it at the lowest theoretical level of temporal granularity, inevitably causing it to miss some information. This is not to say that such extremely fine-grained temporal processing would be useful for the system, but rather to point out that any practical system is virtually guaranteed to miss some high-speed events that occur in the environment.

In a practical sense, our conclusion from all of this can only be that an AGI system must be expected to operate in partially-observable environments and that fully-observable environments are likely to be exceptions.

2) Deterministic / Stochastic

In a deterministic environment, as defined by Russell & Norvig, any changes to the state of the environment are dictated only by the current state of the environment and the actions of the system. This implies that no other entities can make changes to affect the environment, and also that the behavior of the environment is fully predictable to the system.

In stochastic environments, there is uncertainty and unpredictability with regards to future states of the environment and many different outcomes are possible. An AGI system will in all but the most trivial cases be dealing with stochastic environments because, whether the environment is truly stochastic in nature or not, there will be causal chains not immediately accessible or obvious to the AGI system that affect it.

Some aspects of the environment may be truly stochastic while others appear stochastic to the system because it does not have necessary knowledge to predict their behavior. Based on this, an AGI system must be expected to operate in stochastic environments.

3) Static / Dynamic

Static environments are not governed by the passage of time. When dealing with such environments, the system can take an arbitrary amount of time to decide the next action; the environment will not change meanwhile. This is clearly not the case for real world environments, where changes are driven by the clock of the environment regardless of the actions of the system. The present focus on real-world environments dictates that an AGI system must be expected to operate in dynamic environments.

4) Discrete / Continuous

Discrete environments offer a finite number of perceptions and actions that can be taken by the system. A chessboard is a good example of a discrete environment, where there are limited ways to change and perceive the environment. Environments that do not have discrete actions and perceptions are called continuous; typically, this involves real valued action parameters and sensory information. Hence, we must assume continuous environments for AGI systems, while noting that continuous aspects can be approximated with fine-grained discrete functionality.

5) Single agent / Multi-agent

Choosing between these properties is not necessary for AGI systems. Many conceivable operating scenarios involve some type of interaction with other intelligent entities (e.g. humans) while there are perfectly valid and challenging scenarios that are of the single agent variety (e.g. space exploration).

The conclusion from the above analysis is that the types of environments that must be targeted for AGI systems are four (4):

- Partially observable
- Stochastic
- Dynamic
- Continuous

From this, an attempt can be made to define more formally the types of environment that AGI systems target.

A real-world environment is a partially observable, stochastic, dynamic and continuous environment that is governed by its own temporal rhythm and contains vast amounts of continuously changing information.

As AGI systems are by definition unable to use the kind of techniques previously described for narrow AI systems, which rely on design-time domain-dependent knowledge, a fundamentally different approach must be adopted that involves making complex resource management decisions at run-time rather than design-time and gradually learning to adapt such decisions to actual tasks and environments that the system is faced with. Implementing such attention mechanisms is thus a key research problem that must be solved in order to realize practical AGI systems operating in real-world environments.

3.3 Artificial Attention Systems

This section surveys selected AGI architectures and other related work that has attempted to implement some form of attention functionality.

3.3.1 Ymir

The Ymir cognitive architecture was created with the goal of endowing artificial agents with human-like interaction capabilities in the form of embodied multimodal dialog skills that are task oriented and function in real-time (Thórisson 1996, 1999).

Ymir based agents are intended for face-to-face scenarios where users communicate with the agent in a natural fashion without artificial protocols, i.e. as if communicating with another human. A complete perception-action control loop is implemented, with higher level cognitive functions effecting low level perception, and vice versa, in a layered feedback-loop model. Lower layers deal directly with perceptual information and operate at faster time scales than higher layers, in which more advanced cognitive functions occur. Time is handled in an explicit fashion within the system, with every piece of data received and produced being time stamped.

The architecture contains three layers: Reactive (RL), Process Control (PCL), Content (CL) and a resource control system running across these called Action Scheduler (AS). Each layer contains a set of processing elements, including perceptual modules, with unimodal perceptors focusing each on a specific modality, while multimodal integrators are responsible for fusing data from different modalities. Deciders are another type of module that makes decisions based on available data. Sharing of information between modules and layers is accomplished using blackboards, eliminating the need for direct connections between modules.

The RL performs initial processing of perceptual data and produces low-level reactions. The PCL handles the flow of dialog and performs various processing relevant to turn-taking and task-level actions. The CL contains knowledge bases, with one being dedicated to general dialog knowledge and others being topic-specific. It controls the production of topic-relevant actions, based on available perceptual data, in conjunction with its knowledge bases. The process control and content layers have the ability to influence processing in lower layers by turning modules on and off, enabling mixed bottomup/top-down control within the system. The AS accepts behavior requests from these three layers and is responsible for translating those to low level motor movements. To this end, a behavior lexicon is used that contains

specifications of supported behavior, allowing for run-time composition of actions, and also provides a clear separation between behavioral intent and behavior execution.

Action scheduling is a complex problem, highly dependent on time and context, as execution of simultaneous behaviors is allowed and some of them may be conflicting. A scheduling scheme that provides fast response versus optimality is adopted. Long, incremental behavior sequences are a regular part of operation but interruption of these can also be expected at any time.

The way in which control in Ymir is simultaneously bottom-up and top-down can be said to give rise to an attention mechanism in which irrelevant things are ignored by turning off specific modules that produce or consume the irrelevant data. The layered architecture of Ymir, with layers operating at different time scales, is inspired by cognitive psychology in the sense that human cognition is known to have different time scales for different processes. The architecture has been shown to give rise to some human-like qualities as well in implemented systems. Like many of its predecessors, Maes' task network (Maes 1991) and Brook's subsumption architecture (Brooks 1991), Ymir-based systems are completely static at the module level, as modules and their connection potential in the architecture is manually specified a priori; they do not themselves change during operation. This has been referred to as a constructionist approach to building AI architectures (Thórisson et al. 2004). Constructionist architectures rely exclusively on the limits of human programmers, and thus represent a limitation for the complexity such architectures can reach, as interactions and side effects of run-time operation can get exponentially more complex with greater number of modules.

Ymir implements a primitive-top-down controlled filtering attention through its mechanism of enabling turning off certain modules, resulting in certain raw data or intermediate-level perceptual computations being ignored. The system also implements primitive bottom-up attention by tracking indicators of human attention: the Gandalf (Thórisson 1996), for instance, used its interlocutor’s real-time gaze, head direction, and body stance, to infer where the other’s attention was directed, and using this information to control its own internal cognitive processing. This is e.g. how Gandalf resolved ambiguous references to external objects via gesture and speech.

3.3.2 ICARUS

ICARUS is a cognitive architecture for embodied agents that has shown promising results on a number of classic AI toy problems in terms of generality (Langley 2005, Langley 2006). The distinguishing features of the architecture are a separation of memory to conceptual and procedural parts and incremental hierarchical knowledge acquisition for concepts and skills. The conceptual memory stores Boolean concepts and their relations. Skills are composed of more primitive sub-skills that bottom out in actuator manipulation, allowing new skills to be acquired by composition of existing ones. This works similarly for concepts where new concepts can be encoded in terms of existing ones. Memory is also subdivided to long-term and short-term sections. Short-term memory holds intentions and beliefs and is composed of constructs from long-term memory allowing for correspondence that is vital to relate concepts from long-term memory to short lived goals. Symbolic processing is prevalent in the architecture and sensory input is idealized compared to real-world complexity in the examples presented.

Pattern matching is employed to determine relevant skills and knowledge for a given situation using start states and other types of constraints. The control loop of the architecture starts with a bottom-up pass from sensors generating high-level beliefs at the end. Next, a top-down pass is made from beliefs

that includes skill selection and terminates in action. In cases where skills do not exist to reach a goal mean-ends analysis is performed. Cognitive processing is controlled by an attention mechanism that is goal driven and focuses on a single goal at a time.

The attention mechanism in ICARUS is one of the simplest that can be implemented in cognitive architectures. It does not have reactive aspects and can essentially be reduced to selection from active goals. In other words, attention is only controlled in a top-down fashion.

3.3.3 CHREST

CHREST (Chunky Hierarchy and and REtrieval STructures) is a cognitive architecture with a psychological focus that has been used to simulate human cognition in specific domains, such as expert behavior and verbal learning (Gobet 2005). It is based on an earlier architecture called EPAM. The theoretical assumptions CHREST is based on include that there should be close interaction between perception, learning and memory as well as that the mind is caused by a collection of emergent properties produced by the interaction of short- and long-term memory, learning, perception and decision-making processes. In this architecture, operational experience of the system is encoded into chunks, an aggregate structure composed of concepts, schemata and production rules.

The developers of the architecture consider constraining the number of possible architectures, i.e. strong architectural limits, important in the design of cognitive architectures. The architecture contains three components: An input/output module which receives and processes sensory information and controls actuators, long-term memory where operational experience and knowledge are stored and finally short-term memory which is essentially working memory. The operation of CHREST-based systems follows a step lock cognitive cycle in which input is processed (I/O module), matched with long-term memory with matches being copied to short-term memory for further processing. Finally, the I/O module takes over again, performing any prescribed actions and the process then repeats. The long-term memory is the most complex of these components implementing a "chunking network" (discrimination network) that stores different types of items including chunks (patterns), concepts, schemata and production rules. Learning and retrieval operations are used on the chunking network to form and store chunks and retrieve existing ones.

Among other phenomena, CHREST has been used to study human attention, particularly visual attention where a region of an image is selected for detailed processing including feature extraction (Lane 2009). Detected features are used to match elements in long-term memory, with matching elements being copied to short-term memory. Contents of short-term memory, domain-specific knowledge and visual information residing outside the selected image region subsequently guide movements of the systems "eye", effectively guiding attention. The most sophisticated known domain used for evaluation of the architecture is chess playing. In this case, the input image was a crisp, noise-free diagram of a chess table with chess pieces occupying appropriate squares. While the work is interesting and potentially useful in terms of cognitive sciences, it is unclear how this would scale to noisy and highly dynamic environments, especially as real-time operation was not a requirement and the system operates in atomic cognitive cycles.

3.3.4 NARS

The Non-Axiomatic Reasoning System (NARS) is a general-purpose intelligent reasoning system designed for operation in real-time under conditions of insufficient knowledge and resources (Wang, 1995). Knowledge in a NARS system is grounded in its experience, both in terms of meaning and reliability. However, a NARS system is only embodied and situated in the sense of its actual experience, rather than the more traditional sensory-motor sense as NARS does not address sensory-motor issues.

In stark contrast to conventional reasoning systems, most of which exclusively use Boolean truth-values, beliefs in NARS are real-valued numbers based on the experience of the system. This allows a NARS-based system to manage different types of uncertainty such as randomness, fuzziness, and ignorance. NARS is based on a term-oriented formal language called Narsese, which has experience-grounded semantics and a set of inference rules. Thus, knowledge and beliefs contained within the system have associated non-Boolean truth-values that are shaped by operational experience. Learning is achieved by reasoning upon this experience, generating beliefs that grow stronger as they are repeatedly confirmed or weaker if they are contradicted.

Unlike most cognitive architectures, NARS was designed with real-time operation as a requirement from the start. The logic of Narsese is embedded with time, making truth values of appropriate statements time-dependent, in contrast with traditional logic languages that are completely timeless. Time is represented primarily in a relative fashion, with the timing of one event being defined in terms of the timing of another. Temporal logical relations and operators are present in the language as well, providing some necessary tools for temporal reasoning and inference. Core mechanisms in NARS, such as learning – and meta-learning by extension – are fixed.

The control strategy for computation in NARS systems is called controlled concurrency: the execution of tasks is controlled by two special prioritization parameters, urgency and durability. The urgency value gradually decays over time, with the strength of the decay being determined by the durability value. The values depend on both the environment and internal state of the system. These parameters are used to implement dynamic resource management, allowing the system to spend most of its time on what is most important, giving rise to a type of attention mechanism.

Effectively, tasks constantly compete for processing within the system, with losers being eventually removed from the task pool. An interesting property of this mechanism is that resource allocation is context-dependent, (i.e. the same task with the same urgency and durability values will vary in execution time depending on other active tasks at any given time).

Wang (1996) examines the implications of real-time operation under insufficient computational resources, concluding that Turing machines and traditional models of computation are not applicable for such scenarios. The author makes a convincing case that deadline-based task management is not appropriate for intelligent, reactive systems. Instead, he suggests using problem solving algorithms that generate solutions or answers after each iteration with solutions improving as the number of iterations increases, iterations being the atomic processing unit of the system or what has been called an anytime algorithm (Boddy & Dean 1989). Resource management needs to be highly dynamic in these scenarios, influenced by, among others, the intermediate progress of problem-solving processes and exploration of multiple solution paths concurrently and at different speeds, although not necessarily at the hardware level.

Space is also addressed, with bag-based memories being suggested, as memory is finite and items will need to be added and removed frequently during operation.

As for attention, NARS views tasks and goals in a fairly traditional way: A distinction is made between original goals, being input tasks originating outside the system, and derived goals, being created within the system in response to original goals. While urgency and durability parameters are assigned by the system to derived goals, this is not the case for original goals which are supplied externally (e.g. by the system designers).

However, the system can modify some task parameters at runtime according to its experience. As NARS is a reasoning system, and has not focused explicitly on perception and action up to the current implementation, it is intended to accept queries and tasks from an external entity. In this scenario, having priority values dictated by an external entity are not problematic, but a different approach must be used if the system is to control an embodied agent, which includes perception and action functionality. In that setting, the frequency of system tasks will likely to be much greater and reliance on an external entity to provide priority values for each task is problematic as it results in a loss of autonomy.

3.3.5 LIDA

The LIDA architecture (Franklin 2007 & 2012) is intended for intelligent and autonomous software agents and is based upon IDA (Intelligent Distribution Agent), which is an earlier architecture used in an autonomous US Navy software system that negotiates assignments for personnel based on US Navy policies, sailor preferences, and other factors (Franklin 2006). The architecture is an implementation of the Global Workspace Theory of consciousness (Baars, 1988).

LIDA features several types of specialized memory: sensory, sensory-motor, perceptual (implemented as a slip net), episodic, declarative and procedural (implemented as a scheme net). The operation of LIDA-based systems is a series of cognitive cycles, each consisting of sense, attend and action selection phases. In the sensing phase, the current representation of the internal and external environment of the system are updated.

Incoming sensory data activates low-level feature detectors as output from these are sent to perceptual memory, where higher-level feature detectors process the information further. Final processed sensory data is then sent to the local workspace and exposed to declarative memory and episodic memory to generate associations which are also copied to the workspace. This combined data constitutes the system's current understanding of its operating situation. In the attending phase, Attentional Codelets (essentially a collection of small programs) form coalitions of data from the Local Workspace and move these to the Global Workspace. A coalition may be viewed as a collection of functionally related data. In the Global Workspace, the most urgent coalition (only one is selected in each cycle) is selected by a competitive process, and broadcast throughout the system. The broadcast reaches several components of the architecture that are related to learning, memory and decision-making (Action Selection, Perceptual Memory, Procedural Memory, Episodic Memory, Local Workspace and Attentional Codelets) and triggers different types of learning that are performed in parallel: Procedural learning occurs as the data reaches Procedural Memory, attentional learning occurs as the data reaches the Attention Codelets, perceptual learning occurs as the data reaches Perceptual Memory and episodic learning occurs as the data reaches Episodic memory. Following the broadcast, possible actions given the current

situation (encoded by the broadcast) are selected in Procedural Memory and sent to the Action Selection module where one action is selected for execution by a competitive process.

The LIDA architecture does not address time in an explicit fashion, tasks can be scheduled in terms of “ticks” (operating cycles) but not in real-time. However, some promising steps are taken in real-time direction, such as learnable alarm structures, which are reflex-like mechanisms for reacting quickly (faster than the average operating cycle) to certain events. While nothing prevents the system from performing temporal reasoning, there are no provisions for dealing with real-time operation in the control mechanisms of the system. An integrated approach to attention is followed by the architecture where attention is one of three central processes in the operating cycle.

Attention is implemented as filtering/selection which potentially allows the architecture to gracefully handle situations of information overload. Availability of time and resources is taken into account when priorty of available information is evaluated. It should be noted that LIDA implements attentional learning, giving it the capability to improve its own resource management in terms of data filtering. This is significant especially in light of the many different types of learning supported by the architecture.

The core learning mechanisms of the architecture are fixed but as internal data is handled identically to external (environmental) data, the architecture is well suited for introspection and self-improvement at the content level while the architectural level remains fixed.

3.3.6 OSCAR

OSCAR is an implemented architecture for generally intelligent agents operating under uncertainty and incomplete knowledge (Pollock, 2008). The work is inspired by the fact that any human's knowledge of individuals, in the epistemological sense (e.g. individual grains of sand, individual apples on the trees on the planet, etc.), as well as general knowledge, is very sparse. Yet we manage to form beliefs and make decisions with relative ease in our daily lives. According to Pollock, the prevalence of operating under uncertainty strongly suggests some form of statistical probability processing. For this to work, a mechanism is needed to resolve conflicting conclusions, as the introduction of probability into the reasoning process implies that incorrect and contradicting conclusions will occur. This type of reasoning is called defeasible reasoning, and forms the basis of the OSCAR architecture.

Beliefs are encoded in OSCAR as first-order representations, and first-order logic is the basis of reasoning. Inference schemes supplied a priori are used for the reasoning process, such as statistical syllogism. The correctness of inference schemes is evaluated over time; if a particular scheme has been found unreliable under specific circumstances this will be reflected in the reasoning process and conclusions based on that scheme will therefore less likely to be made in the future. The mechanisms for invalidating inferences based on experience are called undercutting defeaters; they are processed in a distinct phase of the reasoning process called defeat status computation. For the sake of practicality, argument construction and defeat status computation are interleaved; otherwise all knowledge that could possibly be relevant to present processing would need to be considered in the argument construction phase before defeat status computation could occur. However, the construction of new arguments can affect defeat status computation with the side effect that not only the argument construction is defeasible but also the defeat status computation itself; reasoning in OSCAR is said to be doubly defeasible.

This produces important properties for generally-intelligent agents, as reasoning can be interrupted at any time, yielding the best conclusions available at that particular point in time and essentially implementing an anytime reasoning algorithm.

The main modules of the architecture are called Practical Cognition and Epistemic Cognition. Practical Cognition has the responsibility of posing planning problems, evaluating and selecting plans as well as directing plan execution. Epistemic Cognition is responsible for constructing plans, generating and revising beliefs, as well as forming epistemic goals. The connection between these two modules forms a loop where epistemic cognition can supply practical cognition with the goal of learning some new information and practical cognition will in turn issue a corresponding goal to epistemic cognition. As plan construction relies on defeasible reasoning, it is a defeasible process and constructed plans can be expected to be invalidated at any time should relevant new information be acquired. Planning and learning are interleaved as forward reasoning (prediction) from perceptual inputs is coupled with backwards reasoning (planning) from goals or interests.

The strength of the OSCAR architecture is its powerful time-bound symbolic reasoning with support for deadlines. The reasoning process, which includes planning, is interruptible at any time for the best available current information, making it suitable for real-time operation. Some introspective capabilities are present, such as dynamic construction of defeaters for inference schemes. However, work remains to be done for OSCAR to be able to control embodied agents.

Weak points of the architecture include lack of attention mechanisms, which has problematic implications for real-time processing when available information exceeds processing capacity. Furthermore, the limitations of memory in the architecture are somewhat unclear from the literature available, for example whether any form of procedural or episodic memory has been implemented.

Finally, how the architecture scales to multi-processor hardware and parallelization is an important question that relates directly to its scalability and practical use: The centralized nature of its operation may hint at problems in this regard. Nevertheless, OSCAR may offer valuable contributions for future work on cognitive architectures as it presents a practical way to implement time-bound reasoning under uncertainty.

3.4 Natural Attention Systems

This chapter surveys selected works on natural attention systems; namely human attention. While many animals besides humans are known to also possess attention mechanisms (Zentall 2004), it is well outside the scope of the present work to determine or speculate on which of Earth's life forms possess attention systems, of which kind, and to what degree. Nevertheless, the fact that attention is not a uniquely human capability is suggestive evidence that attention is a critical cognitive process for surviving in complex and dynamic environments. It seems reasonable to make the assumption that no other species on the planet has more sophisticated or complex attention mechanisms than human beings, or at the very least that human attention presents a sufficient superset of existing attention mechanisms to suffice for the present discussion. This allows discussion of biological attention to focus on human attention as the most interesting case in the theoretical, practical and technological sense, which also makes sense in light of the fact that the vast majority of attention research so far has been focused on human attention.

This section is devoted to present a review of a notable or relevant attention research. Two fields of study are the source of such work: Cognitive psychology and neuroscience.

3.4.1 Cognitive Psychology

3.4.1.1 Knudsen Attention Framework

More recent models of attention focus on the interaction between top-down and bottomup attention, such as the Knudsen attention framework (Knudsen 2007) shown in Figure 3.1 It consists of four interacting processes: working memory, top-down sensitivity control, bottom-up filtering and competitive selection. The first two processes work in a recurrent loop to control top-down attention; working memory is intimately linked to attention as its contents are determined by attention.

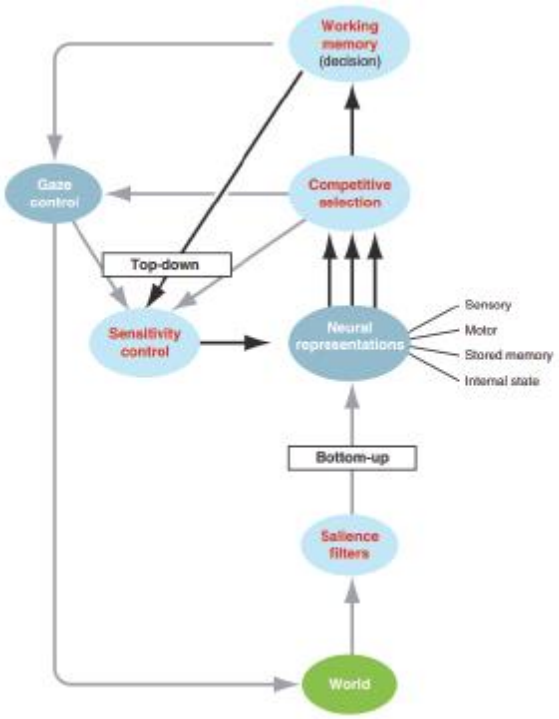


Figure 3.1: The Knudsen attention framework (reprinted from Knudsen 2007). As shown in Figure 3.2, information flows up from the environment and passes through saliency filters that detect important or unusual stimuli. Information that is passed through the filters then activates memory representations that encode knowledge.

Memory representations are also activated by top-down sensitivity control, which is a process influenced by the contents of working memory and adjusts activation thresholds of representations. Representations compete for access to working memory, with the most active ones being admitted. Overall, the flow of information from the environment into working memory is regulated by the framework. While gaze is incorporated in the framework, that component is not necessary to the fundamental operation of attention in the framework.

This framework seems to capture the major necessary parts for attention and be a promising starting point for artificial general intelligence (AGI) systems, from which some important issues for consideration can be extracted.

3.4.1.2 Early Selection vs. Late Selection

The cognitive performance characteristics discussed before imply simultaneous operation of a selective filter and deliberate steering mechanism which together perform allocation of cognitive resources. A number of psychological models for attention have been proposed that typically fall into one of two categories: Early selection models are models where selection of sensory information occurs early in the sensory pipe-line and is based on primitive physical features of the information (shallow processing) and little or no analysis of meaning. In other words, early selection models assume that attention influences perceptual processes. The Broadbent filter model (Broadbent 1958) is one of the best known early-selection (filter) models. It assumes information filtering based on primitive physical features, with information that is not selected by the filter receiving no further processing.

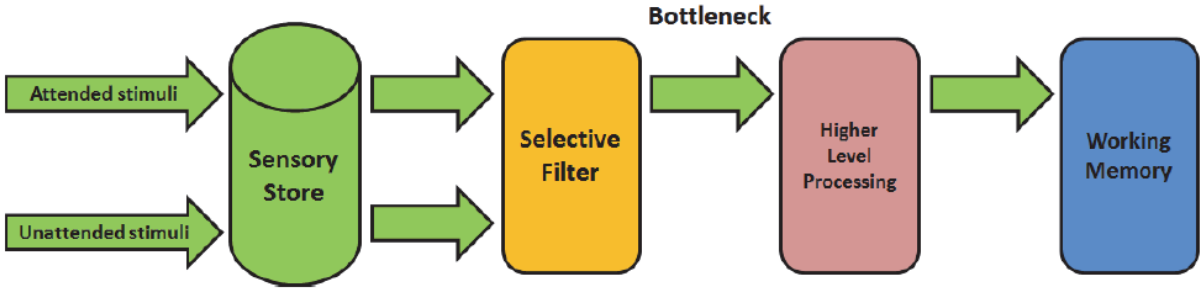


Figure 3.2: Diagram depicting the operation of the Broadbent filter model. All sensory information enters a sensory store from which a selective filter chooses information on the basis of low-level physical characteristics to receive further processing. Information not chosen by the filter is completely discarded.

Late selection models are models where selection is performed after some level of nontrivial analysis of meaning at later stages of the sensory pipeline, assuming further analysis of incoming sensory information must be performed in order to determine its relevance and carry out efficient selection. Implicit in this view is that attention operates only after perceptual processes are completed. The Deutsch-Norman model (Norman 1969) is a prime example of a late selection model. In contrast to the filter model, it proposes gradual processing of information to the point where memory representations are activated. Competitive selection is performed at the level of these representations, with the most active ones being selected for further processing. The model also assumes an attentional bottleneck at this point, where only one representation can be selected for processing at a time.

The early vs. late section issue has resulted in considerable debate in the cognitive psychology community. Some obvious problems are apparent for early selection models; they fail to account for commonly-observed human behavior such as noticing unexpected but relevant information – the

cocktail party effect. The acoustic features alone of someone calling our name from the other side of a crowded room are not likely to be sufficient to attract our attention – some analysis of meaning must be involved. Recent work in neuroscience has found evidence that further validates late selection models: "In and near low-level auditory cortices, attention modulates the representation by enhancing cortical tracking of attended speech streams, but ignored speech remains represented. In higher-order regions, the representation appears to become more selective, in that there is no detectable tracking of ignored speech." (Zion 2013: 980). This work also found evidence of simultaneous involvement of top-down and bottom-up attentional processes in the Cocktail Party Effect and selective auditory attention.

3.4.2 Neuroscience

Neuroscience refers to the scientific study of the nervous system, representing a broad interdisciplinary branch of biology that has collaborated with computer science among many other fields. This section surveys selected phenomena and prior work from neuroscience that is relevant to attention.

3.4.2.1 CODAM

The CODAM (Corollary Discharge of Attention Movement) model of attention is grounded in evidence from neuroscience (Taylor 2007). The most important feature of this model is that the control signal for orienting attention is duplicated, being sent not only to mechanisms carrying out attentional orientation but to working memory mechanisms as well in order to prime working memory for new information that is likely to be forthcoming due to the shift in orientation. The authors believe this duplication of the control signal allows for faster and more efficient access to relevant information in working memory, and furthermore that it is instrumental in giving rise to consciousness.

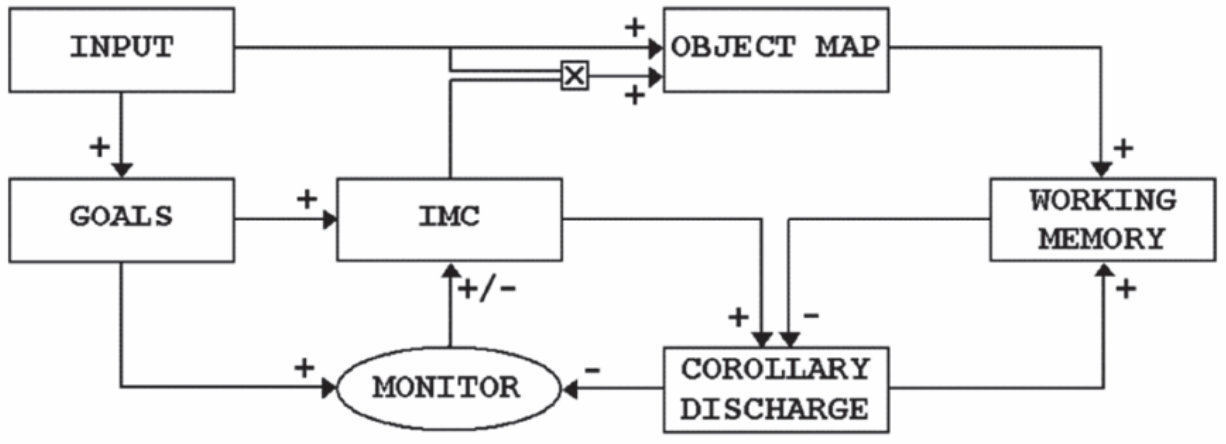


Figure 3.3: The CODAM model of attention (from Taylor 2007).

In CODAM, attention is viewed as a high-level controller for lower-level brain mechanisms. An overview of the model is shown in Figure 3.3, where IMC (Inverse Model Controller) generates a feedback control signal for orientation of attention, which is sent to both perceptual mechanisms to execute

reorientation and to the working memory for priming expected information, reducing effects from distractors and quickly activating an error signal if intended goals of the system are not realized. The Goals module transmits information with regards to intent of the system to the IMC while the Input module is the source of new information to be perceived. The copy of the attention control signal that directly affects working memory is called the corollary discharge and activates a predictive forward model in the Corollary Discharge module which in turn generates expectations with regards to what information is about to be attended. An error monitor (Monitor) generates an error signal based on the differences between what the system intended to observe versus what it actually observes.

While CODAM is a plausible model of some aspects of human attention due to its grounding in neuroscience, the level of abstraction at which the model is presented makes it somewhat difficult to relate to the present work. However, it does establish predictive functionality as an integral part of attention and the failure of predictions as triggering events for reactive behavior.

3.4.2.2 Gamma Band Activity

Recently, recording technologies and tools for analysis have been developed that allow a more detailed examination of low-amplitude cortical oscillations; in particular the 30-100 Hz range which is called the Gamma band. In Kaiser (2003), research on Gamma band activity using a combination of intracortical recordings, EEG and MEG have identified an important role of this signal in a range of cognitive processes. These include top-down and bottom-up attention in addition to learning and memory. The results are interpreted as demonstrating that rather than being mostly focused on perception, the main task of the brain is to anticipate specific requirements related to tasks and activate corresponding structures. This may be viewed as support for the important role of predictive functionality in human cognition.

4.0 Conclusion

This Unit introduced attention and explained how attention is implemented in the various branches of AI systems. What is mentioned under this unit is not in any way exhaustive. Thus, further research and reading is encouraged.

5.0 Summary

We hope you enjoyed this unit. Now, let us attempt the questions below.

6.0 Tutor Marked Assignment

i. Explain why and how implementing Attention in Narrow AI differs form implementing it in AGI systems.

7.0 References/Further Readings

1) Helgason, H. P. (2013). General attention mechanism for artificial intelligence systems (Doctoral dissertation, Ph. D. dissertation, Reykjavik University, 2013.[Online]. Available: http://skemman. is/en/item/view/1946/16163).

2) Helgason, H. P., & Thórisson, K. R. (2012). Attention Capabilities for AI Systems. In ICINCO (1) (pp. 281-286).
3) Knudsen, E. I. (2007). Fundamental components of attention. Annu. Rev. Neurosci., 30, 57-78.
4) Attention in Artificial Intelligence systems Posted by Yi-Ling Hwong on September 22, 2017 https://agi.io/2017/09/22/attention-in-artificial-intelligence-systems/
5) Helgason, H. P., Thórisson, K. R., Garrett, D., & Nivel, E. (2014). Towards a General Attention Mechanism for Embedded Intelligent Systems. International Journal of Computer Science and Artificial Intelligence, 4(1), 1.
6) Viola, P., Jones, M. (2001). Rapid object detection using a boosted cascade of simple features. Proc. IEEE Conference on Computer Vision and Pattern Recognition.
7) Turing, A. M. (1948). Computing machinery and intelligence. Mind 59, 433-460. Reprinted in: 1992, Mechanical Intelligence: Collected Works of A. M. Turing, pages 133-160.
8) Newell, A., Simon, H. (1976). Computer science as empirical inquiry: symbols and search. Communications of the ACM, 19(3), pages 113-126.
9) Minsky, M. (1985). The Society of Mind. Simon and Schuster: New York.
10) McCarthy, J. (1988). Mathematical logic in artificial intelligence. Daedalus, 117(1), pages 297-311.
11) Wang, P. (2013). A General Theory of Intelligence. Available at: https://sites.google.com/site/narswang/EBook
12) Thórisson, K. R., Helgason, H. P. (2012). Cognitive Architectures and Autonomy: A Comparative Review. Journal of Artificial General Intelligence, vol. 3, p. 1-30.
13) Russell, S., & Norvig, P. (2003). Artificial Intelligent: a Modern Approach. Prentice Hall.
14) Thórisson, K. R. (1996). Communicative humanoids: a computational model of psychosocial dialogue skills. Doctoral dissertation, Massachusetts Institute of Technology.
15) Thórisson, K. R. (1999). Mind model for multimodal communicative creatures and humanoids. Applied Artificial Intelligence, 13(4-5), 449-486.
16) Maes, P. (1991). The agent network architecture (ANA). ACM SIGART Bulletin, 2(4),
17) 115-120.
18) Thórisson, K. R., Benko, H., Abramov, D., Arnold, A., Maskey, S., & Vaseekaran, A. (2004). Constructionist design methodology for interactive intelligences. AI Magazine, 25(4), 77.
19) Thórisson, K. R. (1996). Communicative humanoids: a computational model of psychosocial dialogue skills. Doctoral dissertation, Massachusetts Institute of Technology.
20) Langley, P. (2005). An adaptive architecture for physical agents. In Web Intelligence, 2005. Proceedings. The 2005 IEEE/WIC/ACM International Conference on (pp. 18-25). IEEE.
21) Langley, P., & Choi, D. (2006). A unified cognitive architecture for physical agents. In Proceedings of the National Conference on Artificial Intelligence (Vol. 21, No. 2, p. 1469). Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999.
22) Gobet, F., & Lane, P. C. (2005). The CHREST architecture of cognition: Listening to
23) empirical data. Visions of mind: Architectures for cognition and affect, 204-224.
24) Lane, P. C., Gobet, F., & Smith, R. L. (2009). Attention mechanisms in the CHREST cognitive architecture. In Attention in Cognitive Systems (pp. 183-196). Springer Berlin Heidelberg.
25) Wang, P. (1995). Non-Axiomatic Reasoning System: Exploring the Essence of Intelligence. Ph.D. dissertation, Indiana University, Indiana.

26) Wang, P. (1996). Problem-solving under insufficient resources. In Working Notes of the Symposium on Flexible Computation, 148-155. Cambridge, Mass.: AAAI Press.
27) Boddy, M., Dean, T. (1989). Solving time-dependant planning problems. In Sridharan, N. S. (Ed.), Proceedings of the Eleventh International Joint Conference on Artificial Intelligence, 979-984, Detriot, MI, USA. Morgan Kaufmann.
28) Franklin, S. (2006). The LIDA architecture: Adding new modes of learning to an intelligent, autonomous software agent. In Proceedings of the International Conference on Integrated Design and Process Technology, PAGE NUMBERS: San Diego, CA. Society for Design and Process Science.
29) Franklin, S., Ramamurthy, U., D'Mello, S. K., McCauley, L., Negatu, A., Silva, R., & Datla, V. (2007). LIDA: A computational model of global workspace theory and developmental learning. In AAAI Fall Symposium on AI and Consciousness: Theoretical Foundations and Current Approaches (pp. 61-66).
30) Franklin, S., Strain, S., Snaider, J., McCall, R., & Faghihi, U. (2012). Global workspace theory, its LIDA model and the underlying neuroscience. Biologically Inspired Cognitive Architectures, 1(1).
31) Baars, B. J. (1988). A Cognitive Theory of Consciousness. Cambridge, UK: Cambridge University Press.
32) Zentall, T. R. (2004). Selective and divided attention in animals. Behavioral Processes 69.
33) Knudsen, E. I. (2007). Fundamental components of attention. Pages 57-78. Annu Rev Neurosci, vol. 30.
34) Broadbent, D. E. (1958). Perception and Communication. London: Pergamon.
35) Norman, D. A. (1969). Memory while shadowing. Pages 85-93. Quarterly Journal of Experimental Psychology, vol. 21.
36) Taylor, J. G. (2007) CODAM model: Through attention to consciousness. Scholarpedia, 2(11):1598.

Unit Two: Search and Control,

CONTENT

1.0 Introduction

Search is inherent to the problems and methods of artificial intelligence (AI). That is because AI problems are intrinsically complex. Efforts to solve problems with computers which humans can routinely solve by employing innate cognitive abilities, pattern recognition, perception and experience, invariably must turn to considerations of search. All search methods essentially fall into one of two categories 1) exhaustive (blind) methods and 2) heuristic or informed methods. We include other search methods which are generally classified under problem solving methods in AI.

This topic will explain all about the search algorithms in AI

2.0 Objectives

At the end of this unit, you should be able to do the following:

- Define what problem-solving agents are in AI
- Outline the three main factors a search problem can have
- Define the basic terminologies used in search algorithms
- Differentiate between the two major types of search algorithms
- Solve search problems using any of the relevant method reviewed in this unit

3.0 Main Content

Artificial Intelligence is the study of building agents that act rationally. Most of the time, these agents perform some kind of search algorithm in the background in order to achieve their tasks.

The simple reflex agents don't specifically search for best possible solution, as they are programmed to perform a particular action for a particular state. On the contrary, the artificially intelligent agents that work towards a goal, identify the action or series of actions that lead to the goal. The series of actions that lead to the goal becomes the solution for the given problem. Here, the agent has to consider the impact of the action on the future states. Such agents search through all the possible solutions to find the best possible solution for the given problem.

3.1 Problem-solving agents

In Artificial Intelligence, Search techniques are universal problem-solving methods. Rational agents or Problem-solving agents in AI mostly used these search strategies or algorithms to solve a specific problem and provide the best result. Problem-solving agents are the goal-based agents and use atomic representation. In this topic, we will learn various problem-solving search algorithms.

3.2 Search Algorithm Terminologies

Search: Searching is a step by step procedure to solve a search-problem in a given search space. A search problem can have three main factors:

a. Search Space: Search space represents a set of possible solutions, which a system may have.
b. Start State: It is a state from where agent begins the search.

c. Goal test: It is a function which observe the current state and returns whether the goal state is achieved or not.

Search tree: A tree representation of search problem is called Search tree. The root of the search tree is the root node which is corresponding to the initial state.

Actions: It gives the description of all the available actions to the agent.

Transition model: A description of what each action do, can be represented as a transition model.

Path Cost: It is a function which assigns a numeric cost to each path.

Solution: It is an action sequence which leads from the start node to the goal node.

Optimal Solution: If a solution has the lowest cost among all solutions.

The Solution to a search problem is a sequence of actions, called the plan that transforms the start state to the goal state. This plan is achieved through search algorithms.

2.2.3 Measuring problem-solving performance

Before we get into the design of specific search algorithms, we need to consider the criteria that might be used to choose among them. We can evaluate an algorithm's performance in four ways:

Completeness: Is the algorithm guaranteed to find a solution when there is one?

Optimality: Does the strategy find the optimal solution, as defined on page 68?

Time complexity: How long does it take to find a solution?

Space complexity: How much memory is needed to perform the search?

Time and space complexity are always considered with respect to some measure of the problem difficulty. In theoretical computer science, the typical measure is the size of the state space graph, |V | + |E|, where V is the set of vertices (nodes) of the graph and E is the set of edges (links). This is appropriate when the graph is an explicit data structure that is input to the search program. (The map of Romania is an example of this.) In AI, the graph is often represented implicitly by the initial state, actions, and transition model and is frequently infinite.

For these reasons, complexity is expressed in terms of three quantities: b, the branching factor or maximum number of successors of any node; d, the depth of the shallowest goal node (i.e., the number of steps along the path from the root); and m, the maximum length of any path in the state space. Time is often measured in terms of the number of nodes generated during the search, and space in terms of the maximum number of nodes stored in memory. For the most part, we describe time and space complexity for search on a tree; for a graph, the answer depends on how "redundant" the paths in the state space are.

To assess the effectiveness of a search algorithm, we can consider just the search cost—which typically depends on the time complexity but can also include a term for memory usage—or we can use the total cost, which combines the search cost and the path cost of the solution found. For the problem of finding a route from Arad to Bucharest, the search cost is the amount of time taken by the search and the

solution cost is the total length of the path in kilometers. Thus, to compute the total cost, we have to add milliseconds and kilometers. There is no "official exchange rate" between the two, but it might be reasonable in this case to convert kilometers into milliseconds by using an estimate of the car's average speed (because time is what the agent cares about). This enables the agent to find an optimal tradeoff point at which further computation to find a shorter path becomes counterproductive.

3.3 Types of search algorithms

Based on the search problems we can classify the search algorithms into uninformed (Blind search) search and informed search (Heuristic search) algorithms.

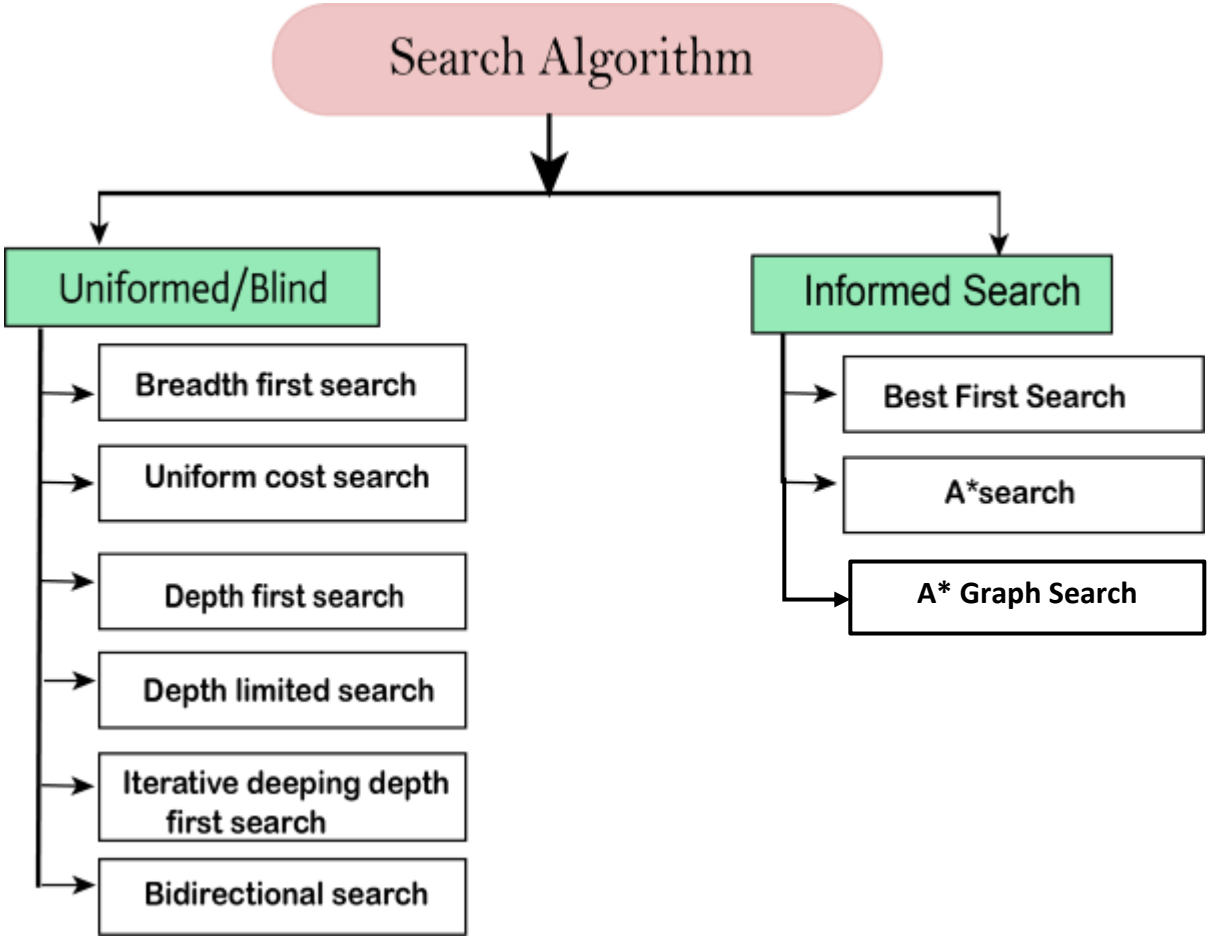


Figure 3.1 Taxonomy of search Algorithms

3.3.1 Uninformed Search Algorithms

The uninformed search does not contain any domain knowledge such as closeness, the location of the goal. It operates in a brute-force way as it only includes information about how to traverse the tree and how to identify leaf and goal nodes. Uninformed search applies a way in which search tree is searched without any information about the search space like initial state operators and test for the goal, so it is

also called blind search. It examines each node of the tree until it achieves the goal node. Following are the various types of uninformed search algorithms:

a. Breadth-first Search
b. Depth-first Search
c. Depth-limited Search
d. Iterative deepening depth-first search
e. Uniform cost search
f. Bidirectional Search

Each of these algorithms will have:

- A problem graph, containing the start node and the goal node.
- A strategy, describing the manner in which the graph will be traversed to get to goal node.
- A fringe, which is a data structure used to store all the possible states (nodes) that you can go from the current states.
- A tree, that results while traversing to the goal node.
- A solution plan, which the sequence of nodes from the start node to the goal node.

3.3.1.1 Breadth First Search

Breadth-first search (BFS) is an algorithm for traversing or searching tree or graph data structures. It starts at the tree root (or some arbitrary node of a graph, sometimes referred to as a 'search key'), and explores all of the neighbor nodes at the present depth prior to moving on to the nodes at the next depth level. The breadth-first search algorithm is an example of a general-graph search algorithm. Breadth-first search is implemented using FIFO queue data structure.

Advantages:

a. BFS will provide a solution if any solution exists.
b. If there are more than one solutions for a given problem, then BFS will provide the minimal solution which requires the least number of steps.

Disadvantages:

a. It requires lots of memory since each level of the tree must be saved into memory to expand the next level.
b. BFS needs lots of time if the solution is far away from the root node.

Example 01:

In the below tree structure, we have shown the traversing of the tree using BFS algorithm from the root node S to goal node K. BFS search algorithm traverse in layers, so it will follow the path which is shown by the dotted arrow, and the traversed path will be:

S---> A--->B---->C--->D---->G--->H--->E---->F---->I---->K

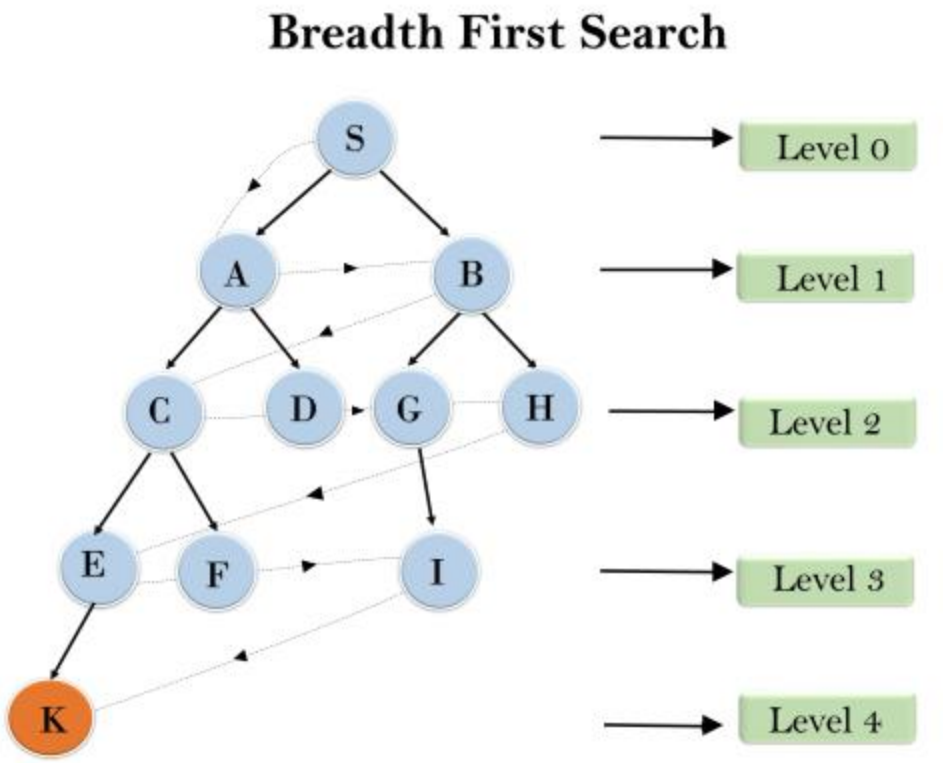


Figure 3.2: Breath First Search Technique

Time Complexity: Time Complexity of BFS algorithm can be obtained by the number of nodes traversed in BFS until the shallowest Node. Where the d= depth of shallowest solution and b is a node at every state.

$T(b) = 1+b^2+b^3+.....+ b^d= O(b^d)$

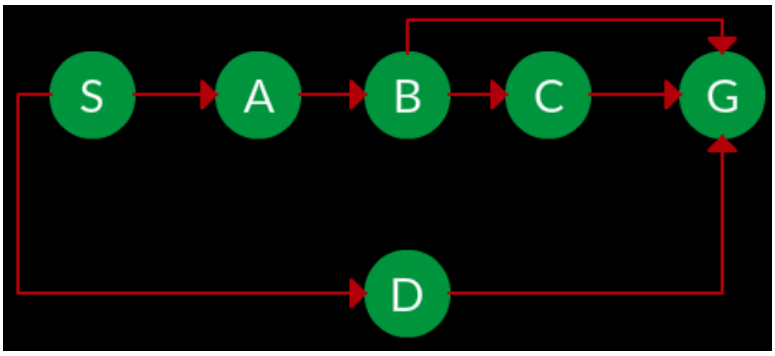
Space Complexity: Space complexity of BFS algorithm is given by the Memory size of frontier which is $O(b^d)$.

Completeness: BFS is complete, which means if the shallowest goal node is at some finite depth, then BFS will find a solution.

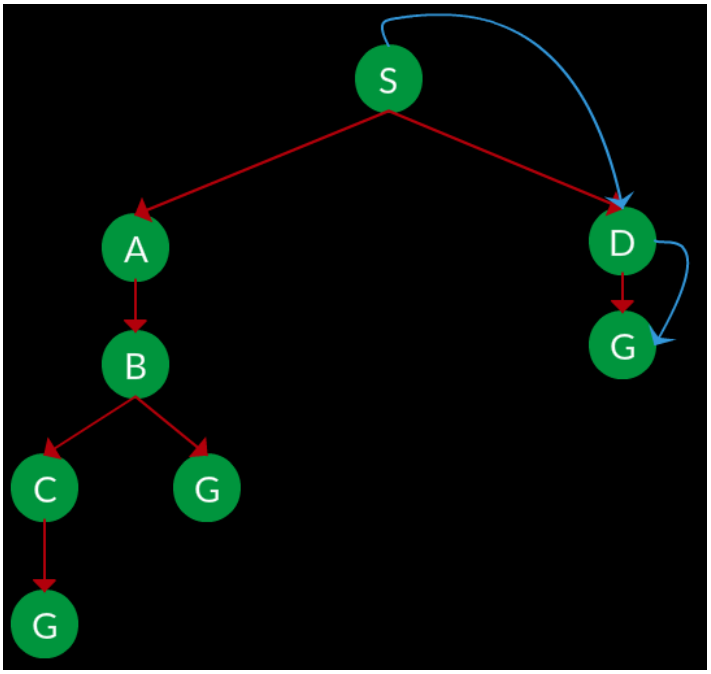
Optimality: BFS is optimal if path cost is a non-decreasing function of the depth of the node.

Example 02:

Question: Which solution would BFS find to move from node S to node G if run on the graph below?



Solution: The equivalent search tree for the above graph is as follows. As BFS traverses the tree "shallowest node first", it would always pick the shallower branch until it reaches the solution (or it runs out of nodes, and goes to the next branch). The traversal is shown in blue arrows.



Path: S -> D -> G

Let s = the depth of the shallowest solution.

n^i = number of nodes in level i.

Time complexity: Equivalent to the number of nodes traversed in BFS until the shallowest solution.

$T(n) = 1 + n^2 + n^3 + ... + n^s = O(n^s)$

Space complexity: Equivalent to how large can the fringe get. $S(n) = O(n^s)$

Completeness: BFS is complete, meaning for a given search tree, BFS will come up with a solution if it exists.

Optimality: BFS is optimal as long as the costs of all edges are equal.

3.3.1.2 Depth First Search

Depth-first search (DFS) is a recursive algorithm for traversing or searching tree or graph data structures. It is called the depth-first search because it starts from the root node and follows each path to its greatest depth node before moving to the next path. The algorithm starts at the root node (selecting some arbitrary node as the root node in the case of a graph) and explores as far as possible along each branch before backtracking. (Note: Backtracking is an algorithm technique for finding all possible solutions using recursion.). DFS uses a stack data structure for its implementation. The process of the DFS algorithm is similar to the BFS algorithm.

Advantage:

a. DFS requires very less memory as it only needs to store a stack of the nodes on the path from root node to the current node.
b. It takes less time to reach to the goal node than BFS algorithm (if it traverses in the right path).

Disadvantage:

a. There is the possibility that many states keep re-occurring, and there is no guarantee of finding the solution.
b. DFS algorithm goes for deep down searching and sometime it may go to the infinite loop.

Example 01:

In the below search tree, we have shown the flow of depth-first search, and it will follow the order as:

Root node--->Left node ----> right node.

It will start searching from root node S, and traverse A, then B, then D and E, after traversing E, it will backtrack the tree as E has no other successor and still goal node is not found. After backtracking it will traverse node C and then G, and here it will terminate as it found goal node.

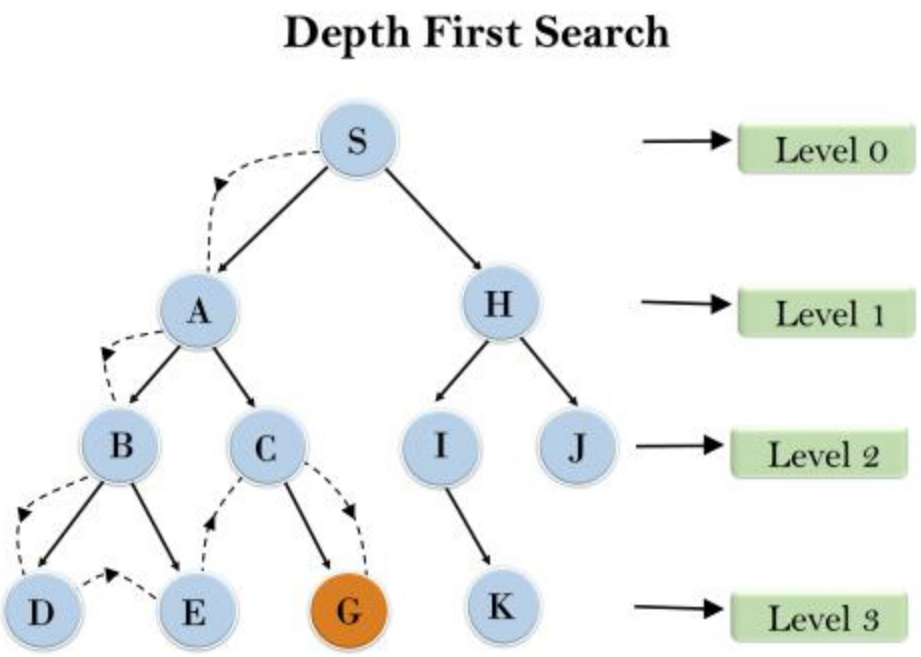


Figure 3.3: Depth First Search

Completeness: DFS search algorithm is complete within finite state space as it will expand every node within a limited search tree.

Time Complexity: Time complexity of DFS will be equivalent to the node traversed by the algorithm. It is given by:

$T(n)= 1+ n^2+ n^3 +.....+ n^m = O(n^m)$

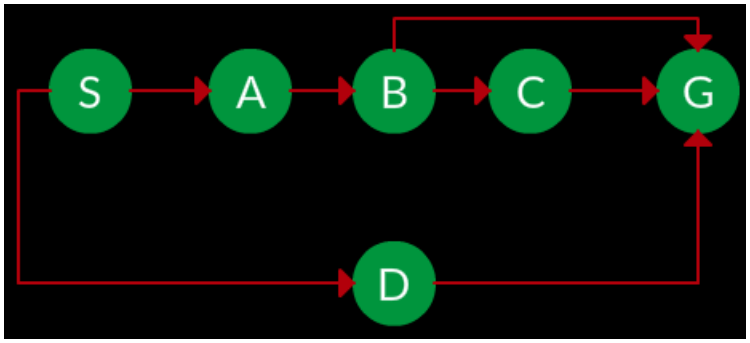
Where, m= maximum depth of any node and this can be much larger than d (Shallowest solution depth)

Space Complexity: DFS algorithm needs to store only single path from the root node, hence space complexity of DFS is equivalent to the size of the fringe set, which is $O(b^m)$.

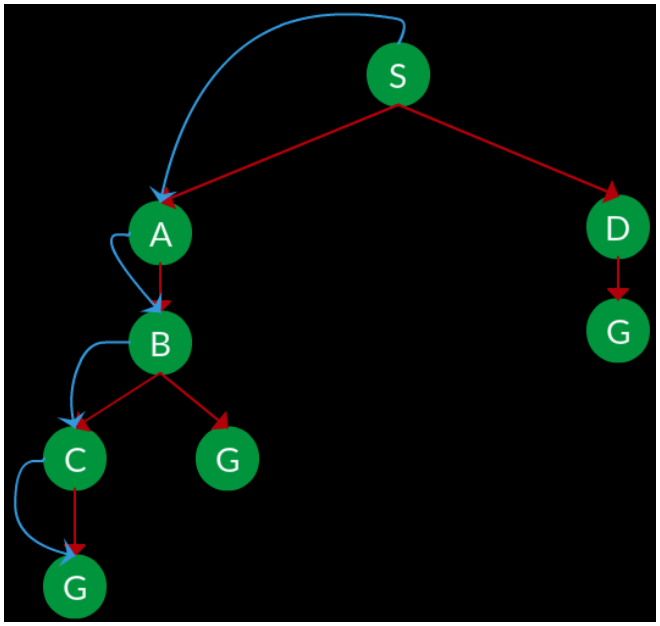
Optimal: DFS search algorithm is non-optimal, as it may generate a large number of steps or high cost to reach to the goal node.

Example 02:

Question: Which solution would DFS find to move from node S to node G if run on the graph below?



Solution: The equivalent search tree for the above graph is as follows. As DFS traverses the tree "deepest node first", it would always pick the deeper branch until it reaches the solution (or it runs out of nodes, and goes to the next branch). The traversal is shown in blue arrows.



Path: S -> A -> B -> C -> G

Let d = the depth of the search tree = number of levels of the search tree.

n^i = number of nodes in level i.

Time complexity: Equivalent to the number of nodes traversed in DFS.

$T(n) = 1 + n^2 + n^3 + ... + n^d = O(n^d)$

Space complexity: Equivalent to how large can the fringe get. S(n) = O(n X d)

Completeness: DFS is complete if the search tree is finite, meaning for a given finite search tree, DFS will come up with a solution if it exists.

Optimality: DFS is not optimal, meaning the number of steps in reaching the solution, or the cost spent in reaching it is high.

3.3.1.3 Depth-Limited Search Algorithm

A depth-limited search algorithm is similar to depth-first search with a predetermined limit. Depth-limited search can solve the drawback of the infinite path in the Depth-first search. In this algorithm, the node at the depth limit will treat as it has no successor nodes further.

Depth-limited search can be terminated with two Conditions of failure:

a. Standard failure value: It indicates that problem does not have any solution.
b. Cutoff failure value: It defines no solution for the problem within a given depth limit.

Advantages:

Depth-limited search is Memory efficient.

Disadvantages:

a. Depth-limited search also has a disadvantage of incompleteness.
b. It may not be optimal if the problem has more than one solution.

Example:

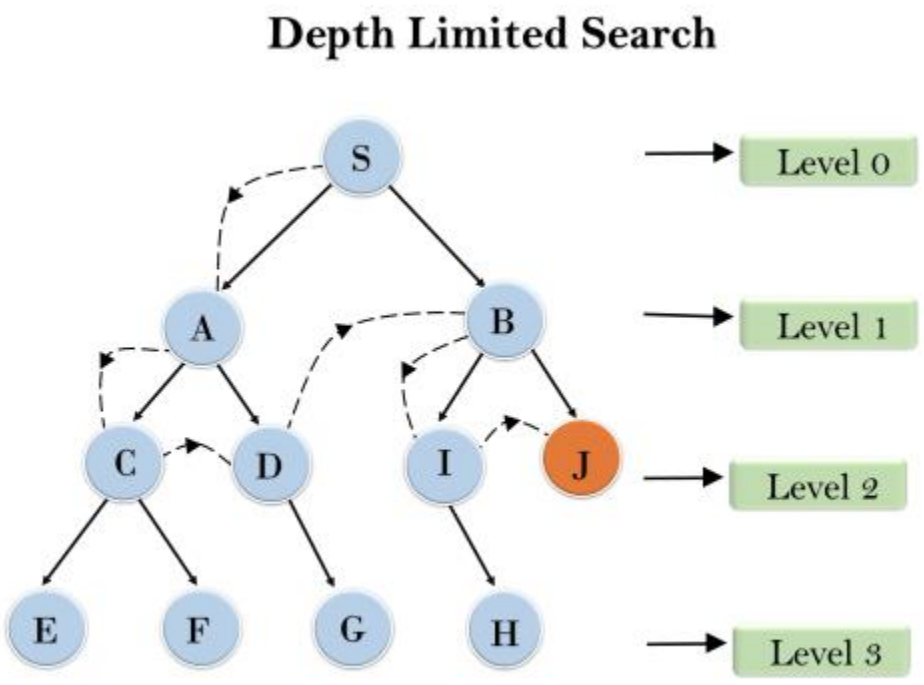


Figure 3.4: Deep Limited Search

Completeness: DLS search algorithm is complete if the solution is above the depth-limit.

Time Complexity: Time complexity of DLS algorithm is O(bℓ).

Space Complexity: Space complexity of DLS algorithm is O(b×ℓ).

Optimal: Depth-limited search can be viewed as a special case of DFS, and it is also not optimal even if ℓ>d.

3.3.1.4 Uniform Cost Search

Uniform-cost search is a searching algorithm used for traversing a weighted tree or graph. This algorithm comes into play when a different cost is available for each edge. The primary goal of the uniform-cost search is to find a path to the goal node which has the lowest cumulative cost. Uniform-cost search expands nodes according to their path costs form the root node. It can be used to solve any graph/tree where the optimal cost is in demand. A uniform-cost search algorithm is implemented by the priority queue. It gives maximum priority to the lowest cumulative cost. Uniform cost search is equivalent to BFS algorithm if the path cost of all edges is the same.

Advantages:

Uniform cost search is optimal because at every state the path with the least cost is chosen. In other word UCS is complete and optimal

Disadvantages:

It does not care about the number of steps involve in searching and only concerned about path cost (i.e Explores options in every “direction”). Due to which this algorithm may be stuck in an infinite loop (i.e. No information on goal location)

Cost of a node is defined as:

cost(node) = cumulative cost of all nodes from root

cost(root) = 0

Example 01:

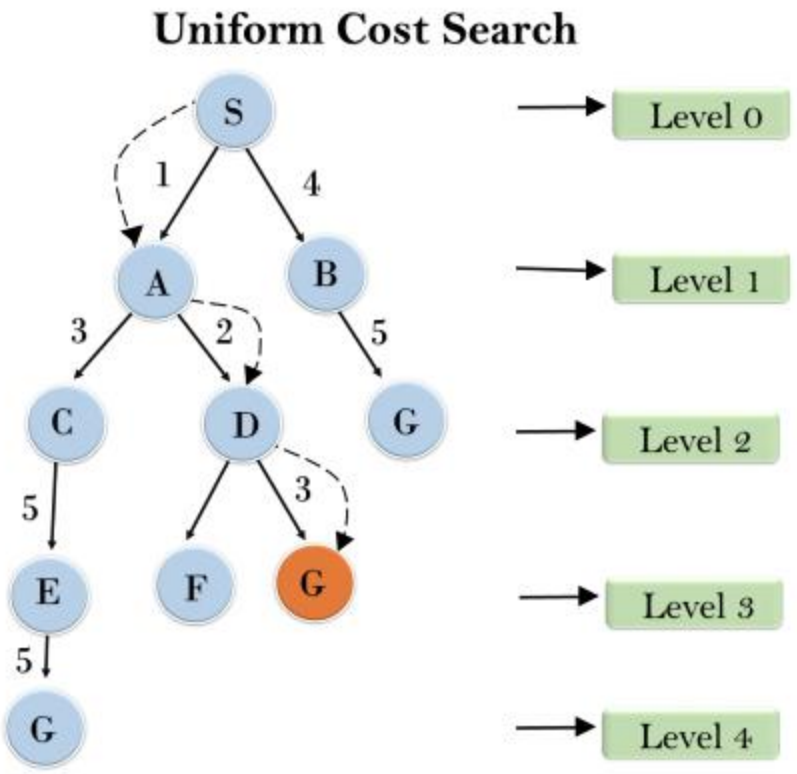


Figure 3.5: Uniform Cost Search

Completeness:

Uniform-cost search is complete, such as if there is a solution, UCS will find it.

Time Complexity:

Let C* is **Cost of the optimal solution**, and ε is each step to get closer to the goal node. Then the number of steps is = C*/ε+1. Here we have taken +1, as we start from state 0 and end to C*/ε.

Hence, the worst-case time complexity of Uniform-cost search is $O(b^{1 + [C^*/\varepsilon]})$/.

Space Complexity:

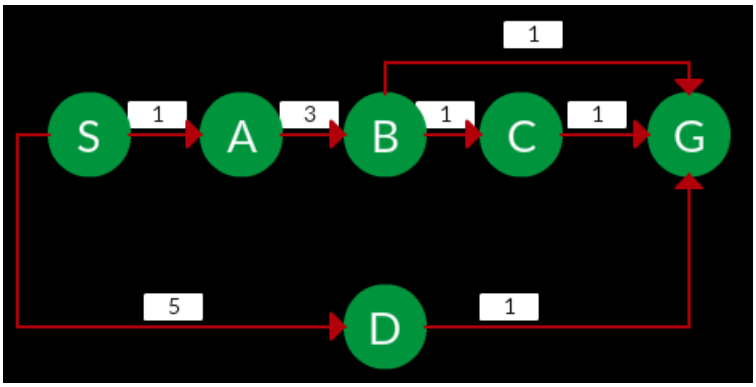
The same logic is for space complexity so, the worst-case space complexity of Uniform-cost search is $O(b^{1 + [C^*/\varepsilon]})$.

Optimal:

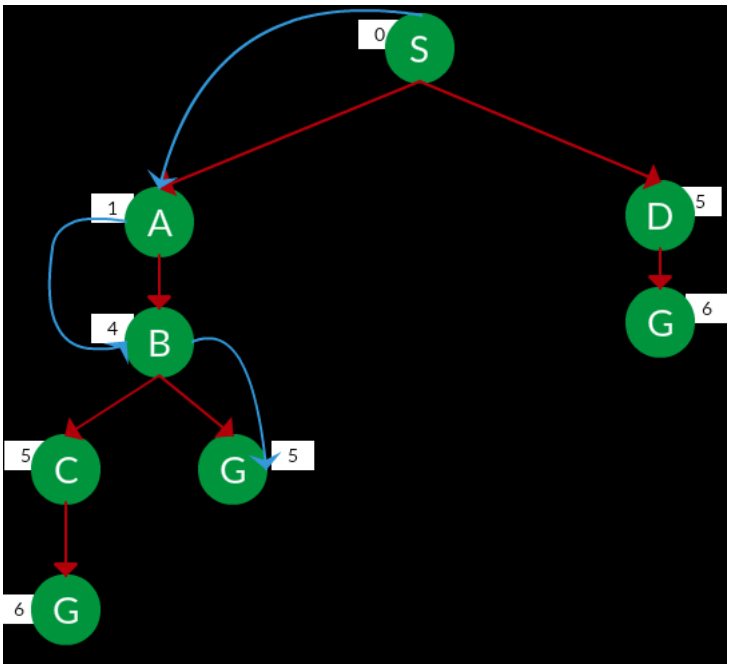
Uniform-cost search is always optimal as it only selects a path with the lowest path cost.

Example 02:

Question: Which solution would UCS find to move from node S to node G if run on the graph below?



Solution: The equivalent search tree for the above graph is as follows. Cost of each node is the cumulative cost of reaching that node from the root. Based on UCS strategy, the path with least cumulative cost is chosen. Note that due to the many options in the fringe, the algorithm explores most of them so long as their cost is low, and discards them when a lower cost path is found; these discarded traversals are not shown below. The actual traversal is shown in blue.



Path: S -> A -> B -> G

Cost: 5

Let C = cost of solution.

ϵ = arcs cost.

Then C / ϵ = effective depth

Time complexity: $T(n) = O(n^{(C/\epsilon)})$

Space complexity: $S(n) = O\ O(n^{(C/\epsilon)})$

3.3.1.5 Iterative deepening depth-first Search

The iterative deepening algorithm is a combination of DFS and BFS algorithms. This search algorithm finds out the best depth limit and does it by gradually increasing the limit until a goal is found. This algorithm performs depth-first search up to a certain "depth limit", and it keeps increasing the depth limit after each iteration until the goal node is found.

This Search algorithm combines the benefits of Breadth-first search's fast search and depth-first search's memory efficiency. The iterative search algorithm is useful uninformed search when search space is large, and depth of goal node is unknown.

Advantages:

It combines the benefits of BFS and DFS search algorithm in terms of fast search and memory efficiency.

Disadvantages:

The main drawback of IDDFS is that it repeats all the work of the previous phase.

Example:

Following tree structure is showing the iterative deepening depth-first search. IDDFS algorithm performs various iterations until it does not find the goal node. The iteration performed by the algorithm is given as:

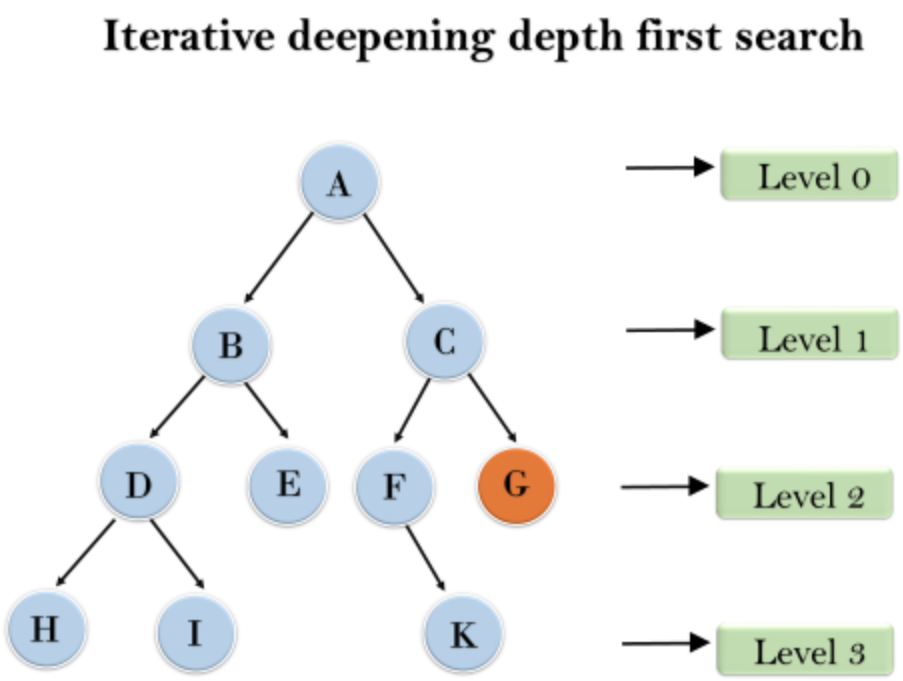


Figure 3.6: Iterative Deepening depth First Search

1'st Iteration-----> A

2'nd Iteration----> A, B, C

3'rd Iteration------>A, B, D, E, C, F, G

4'th Iteration------>A, B, D, H, I, E, C, F, K, G

In the fourth iteration, the algorithm will find the goal node.

Completeness:

This algorithm is complete is if the branching factor is finite.

Time Complexity:

Let's suppose b is the branching factor and depth is d then the worst-case time complexity is $O(b^d)$.

Space Complexity:

The space complexity of IDDFS will be $O(b^d)$.

Optimal:

IDDFS algorithm is optimal if path cost is a non- decreasing function of the depth of the node.

3.3.1.6 Bidirectional Search Algorithm

In the below search tree, bidirectional search algorithm is applied. This algorithm divides one graph/tree into two sub-graphs. It starts traversing from node 1 in the forward direction and starts from goal node 16 in the backward direction.

The algorithm terminates at node 9 where two searches meet.

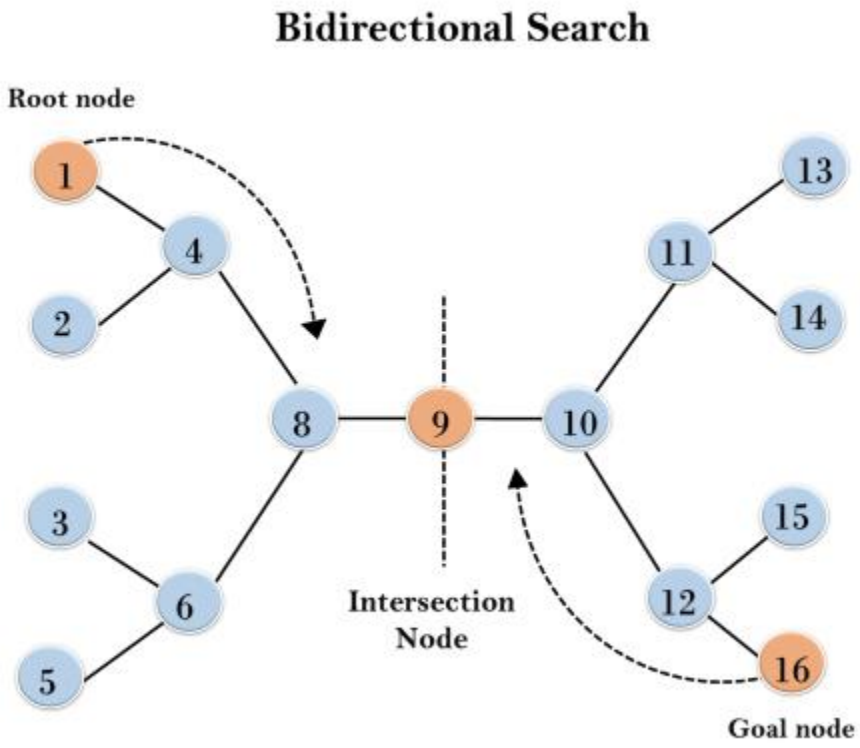


Figure 3.7: Bidirectional Search

Completeness: Bidirectional Search is complete if we use BFS in both searches.

Time Complexity: Time complexity of bidirectional search using BFS is O(bd).

Space Complexity: Space complexity of bidirectional search is O(bd).

Optimal: Bidirectional search is Optimal.

3.3.2 Informed Search Algorithms

So far we have talked about the uninformed search algorithms which looked through search space for all possible solutions of the problem without having any additional knowledge about search space. But informed search algorithm contains an array of knowledge such as how far we are from the goal, path

cost, how to reach to goal node, etc. This knowledge help agents to explore less to the search space and find more efficiently the goal node.

The informed search algorithm is more useful for large search space. Informed search algorithm uses the idea of heuristic, so it is also called Heuristic search.

Heuristics function: Heuristic is a function which is used in Informed Search, and it finds the most promising path. It takes the current state of the agent as its input and produces the estimation of how close agent is from the goal. The heuristic method, however, might not always give the best solution, but it guaranteed to find a good solution in reasonable time. Heuristic function estimates how close a state is to the goal. It is represented by h(n), and it calculates the cost of an optimal path between the pair of states. The value of the heuristic function is always positive.

Admissibility of the heuristic function is given as:

$h(n) \leq h^*(n)$

Here h(n) is heuristic cost, and h*(n) is the estimated cost. Hence heuristic cost should be less than or equal to the estimated cost.

Pure Heuristic Search:

Pure heuristic search is the simplest form of heuristic search algorithms. It expands nodes based on their heuristic value h(n). It maintains two lists, OPEN and CLOSED list. In the CLOSED list, it places those nodes which have already expanded and in the OPEN list, it places nodes which have yet not been expanded.

In an informed search, a heuristic is a function that estimates how close a state is to the goal state. For examples – Manhattan distance, Euclidean distance, etc. (Lesser the distance, closer the goal.) Thus, on each iteration, each node n with the lowest heuristic value is expanded and generates all its successors and n is placed to the closed list. The algorithm continues unit a goal state is found. Different heuristics are used in different informed algorithms discussed below.

In this section, we will discuss the following search algorithms.

I. Greedy Search
II. A* Tree Search
III. A* Graph Search

3.3.2.1 Best-first Search Algorithm (Greedy Search)

Greedy best-first search algorithm always selects the path which appears best at that moment. It is the combination of depth-first search and breadth-first search algorithms. It uses the heuristic function and search. Best-first search allows us to take the advantages of both algorithms. With the help of best-first search, at each step, we can choose the most promising node. In the best first search algorithm, we expand the node which is closest to the goal node and the closest cost is estimated by heuristic function, i.e.

f(n)= g(n).

Were, h(n)= estimated cost from node n to the goal.

The greedy best first algorithm is implemented by the priority queue.

In greedy search, we expand the node closest to the goal node. The "closeness" is estimated by a heuristic h(x) .

Best first search algorithm:

i. **Step 1:** Place the starting node into the OPEN list.
ii. **Step 2:** If the OPEN list is empty, Stop and return failure.
iii. **Step 3:** Remove the node n, from the OPEN list which has the lowest value of h(n), and places it in the CLOSED list.
iv. **Step 4:** Expand the node n, and generate the successors of node n.
v. **Step 5:** Check each successor of node n, and find whether any node is a goal node or not. If any successor node is goal node, then return success and terminate the search, else proceed to Step 6.
vi. **Step 6:** For each successor node, algorithm checks for evaluation function f(n), and then check if the node has been in either OPEN or CLOSED list. If the node has not been in both list, then add it to the OPEN list.
vii. **Step 7:** Return to Step 2.

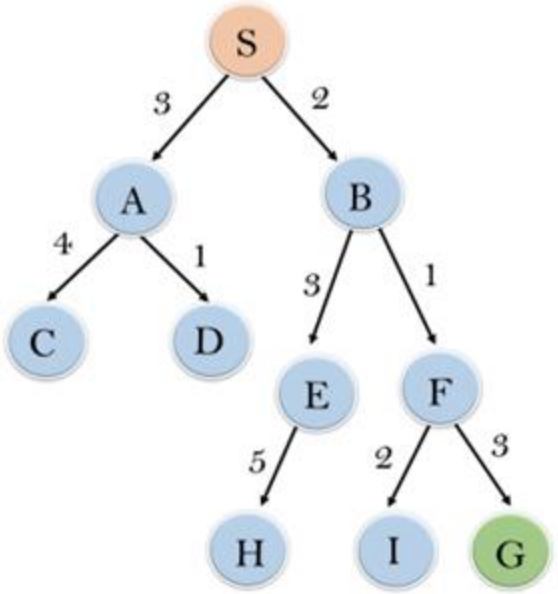
Advantages:

a. Best first search can switch between BFS and DFS by gaining the advantages of both the algorithms.
b. This algorithm is more efficient than BFS and DFS algorithms.

Disadvantages:

a. It can behave as an unguided depth-first search in the worst-case scenario.
b. It can get stuck in a loop as DFS.
c. This algorithm is not optimal.

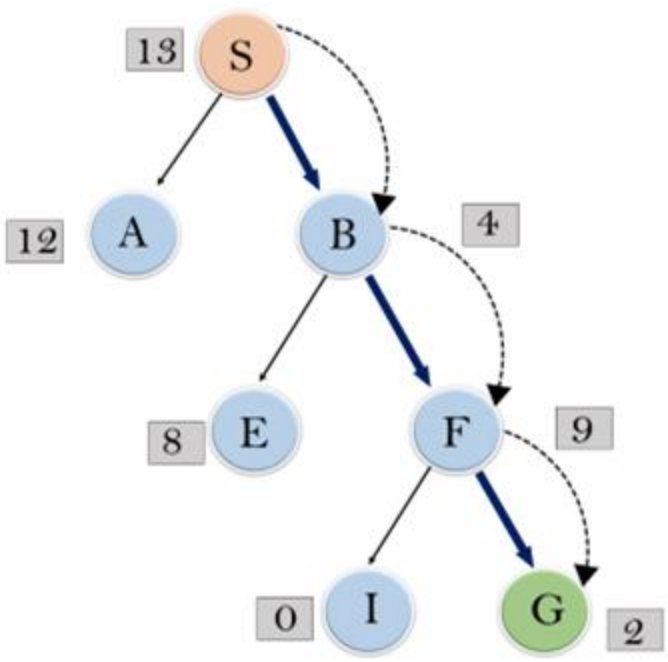
Example 01:



node	H (n)
A	12
B	4
C	7
D	3
E	8
F	2
H	4
I	9
S	13
G	0

Figure 3.8: Best-first Search Algorithm (Greedy Search)

In this search example, we are using two lists which are OPEN and CLOSED Lists. Following are the iteration for traversing the above example.



Expand the nodes of S and put in the CLOSED list

Initialization: Open [A, B], Closed [S]

Iteration 1: Open [A], Closed [S, B]

Iteration 2: Open [E, F, A], Closed [S, B]

: Open [E, A], Closed [S, B, F]

Iteration 3: Open [I, G, E, A], Closed [S, B, F]

: Open [I, E, A], Closed [S, B, F, G]

Hence the final solution path will be: **S----> B----->F----> G**

Time Complexity: The worst-case time complexity of Greedy best first search is $O(b^m)$.

Space Complexity: The worst-case space complexity of Greedy best first search is $O(b^m)$. Where, m is the maximum depth of the search space.

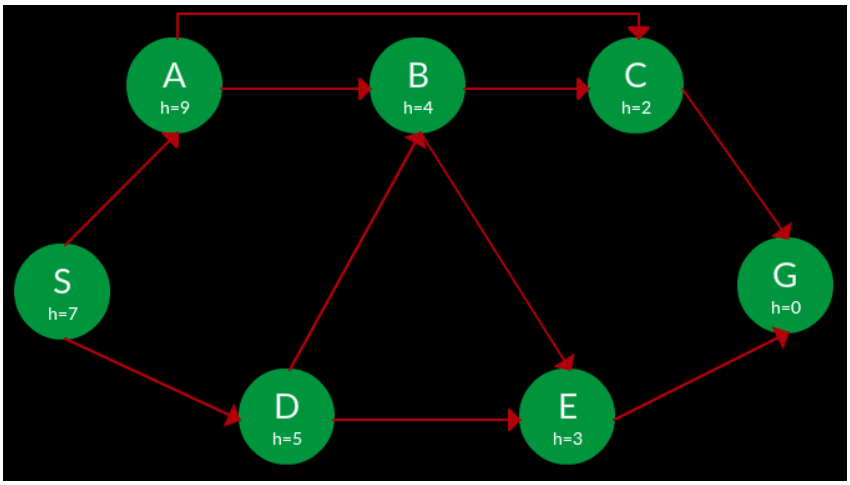
Complete: Greedy best-first search is also incomplete, even if the given state space is finite.

Optimal: Greedy best first search algorithm is not optimal.

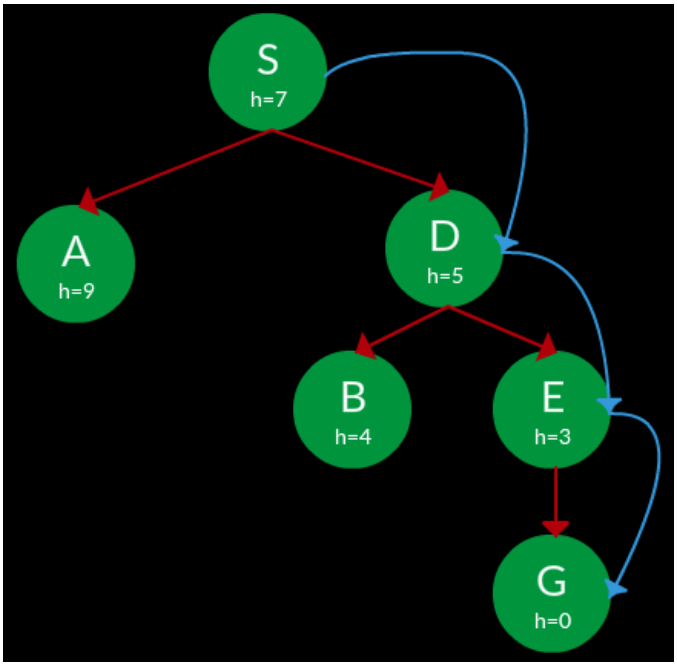
Example 02:

Consider the below search problem, and we will traverse it using greedy best-first search. At each iteration, each node is expanded using evaluation function f(n)=h(n) , which is given in the below table.

Question: Find the path from S to G using greedy search. The heuristic values h of each node below the name of the node.



Solution. Starting from S, we can traverse to A(h=9) or D(h=5). We choose D, as it has the lower heuristic cost. Now from D, we can move to B(h=4) or E(h=3). We choose E with lower heuristic cost. Finally, from E, we go to G(h=0). This entire traversal is shown in the search tree below, in blue.

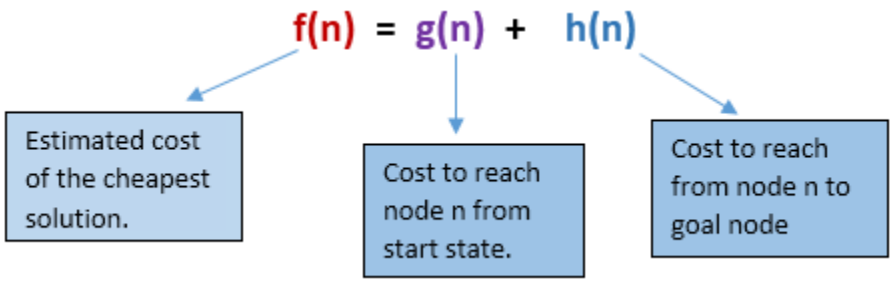


Path: S -> D -> E -> G

3.3.2.2 A* Tree Search

A* search is the most commonly known form of best-first search. It uses heuristic function h(n), and cost to reach the node n from the start state g(n). It has combined features of UCS and greedy best-first search, by which it solves the problem efficiently. A* search algorithm finds the shortest path through the search space using the heuristic function. This search algorithm expands less search tree and provides optimal result faster. A* algorithm is similar to UCS except that it uses g(n)+h(n) instead of g(n). Thus, the heuristic is the summation of the cost in UCS, denoted by g(n), and the cost in greedy search, denoted by h(n). The summed cost is denoted by f(n).

Hence, we can combine both costs as following, and this sum is called as a **fitness number**.



Heuristic: The following points should be noted with respect to heuristics in A* search. f(n) = g(n) + h(n)

a. Here, h(n) is called the forward cost, and is an estimate of the distance of the current node from the goal node.
b. And, g(n) is called the backward cost, and is the cumulative cost of a node from the root node.
c. A* search is optimal only when for all nodes, the forward cost for a node h(x) underestimates the actual cost h*(n) to reach the goal. This property of A* heuristic is called **admissibility**.

Admissibility: $0 \leq h(x) \leq h^{*}(x)$

Strategy: Choose the node with lowest f(x) value.

Algorithm of A* search:

a. **Step1:** Place the starting node in the OPEN list.
b. **Step 2:** Check if the OPEN list is empty or not, if the list is empty then return failure and stops.
c. **Step 3:** Select the node from the OPEN list which has the smallest value of evaluation function (g+h), if node n is goal node then return success and stop, otherwise
d. **Step 4:** Expand node n and generate all of its successors, and put n into the closed list. For each successor n', check whether n' is already in the OPEN or CLOSED list, if not then compute evaluation function for n' and place into Open list.
e. **Step 5:** Else if node n' is already in OPEN and CLOSED, then it should be attached to the back pointer which reflects the lowest g(n') value.
f. **Step 6:** Return to Step 2.

Advantages:

a. A* search algorithm is the best algorithm than other search algorithms.
b. A* search algorithm is optimal and complete.
c. This algorithm can solve very complex problems.

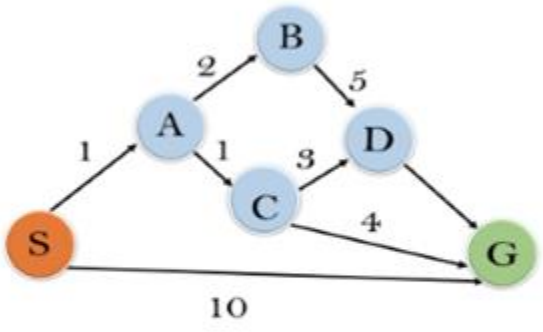
Disadvantages:

a. It does not always produce the shortest path as it mostly based on heuristics and approximation.
b. A* search algorithm has some complexity issues.
c. The main drawback of A* is memory requirement as it keeps all generated nodes in the memory, so it is not practical for various large-scale problems.

Example 01:

In this example, we will traverse the given graph using the A* algorithm. The heuristic value of all states is given in the below table so we will calculate the f(n) of each state using the formula f(n)= g(n) + h(n), where g(n) is the cost to reach any node from start state.

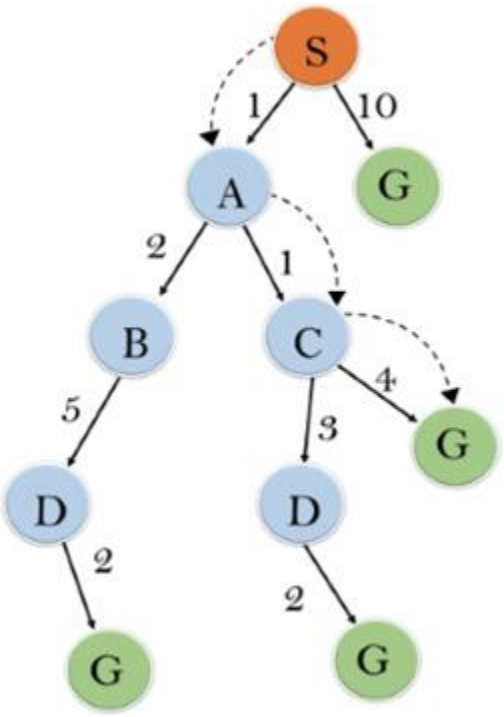
Here we will use OPEN and CLOSED list.



State	h(n)
S	5
A	3
B	4
C	2
D	6
G	0

Figure 3.9: A* Tree Search

Solution:



Initialization: {(S, 5)}

Iteration1: {(S--> A, 4), (S-->G, 10)}

Iteration2: {(S--> A-->C, 4), (S--> A-->B, 7), (S-->G, 10)}

Iteration3: {(S--> A-->C--->G, 6), (S--> A-->C--->D, 11), (S--> A-->B, 7), (S-->G, 10)}

Iteration 4 will give the final result, as S--->A--->C--->G it provides the optimal path with cost 6.

Points to remember:

a. A* algorithm returns the path which occurred first, and it does not search for all remaining paths.
b. The efficiency of A* algorithm depends on the quality of heuristic.
c. A* algorithm expands all nodes which satisfy the condition f(n)

Completeness: A* algorithm is complete as long as:

a. Branching factor is finite.
b. Cost at every action is fixed.

Optimal: A* search algorithm is optimal if it follows below two conditions:

Admissible: the first condition requires for optimality is that h(n) should be an admissible heuristic for A* tree search. An admissible heuristic is optimistic in nature.

Consistency: Second required condition is consistency for only A* graph-search.

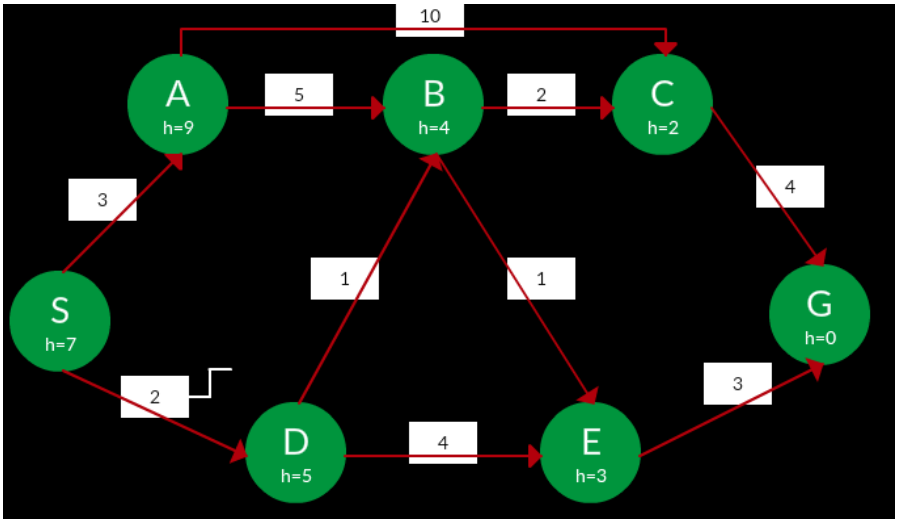
If the heuristic function is admissible, then A* tree search will always find the least cost path.

Time Complexity: The time complexity of A* search algorithm depends on heuristic function, and the number of nodes expanded is exponential to the depth of solution d. So, the time complexity is O(b^d), where b is the branching factor.

Space Complexity: The space complexity of A* search algorithm is O(b^d)

Example 02:

Question: Find the path to reach from S to G using A* search.



Solution: Starting from S, the algorithm computes g(x) + h(x) for all nodes in the fringe at each step, choosing the node with the lowest sum. The entire working is shown in the table below.

Note that in the fourth set of iteration, we get two paths with equal summed cost f(x), so we expand them both in the next set. The path with lower cost on further expansion is the chosen path.

PATH	H(X)	G(X)	F(X)
S	7	0	7
S -> A	9	3	12
S -> D √	5	2	7
S -> D -> B √	4	2 + 1 = 3	7

S -> D -> E	3	2 + 4 = 6	9
S -> D -> B -> C √	2	3 + 2 = 5	7
S -> D -> B -> E √	3	3 + 1 = 4	7
S -> D -> B -> C -> G	0	5 + 4 = 9	9
S -> D -> B -> E -> G √	0	4 + 3 = 7	7

Path: S -> D -> B -> E -> G

Cost: 7

3.3.2.3 A* Graph Search

A* tree search works well, except that it takes time re-exploring the branches it has already explored. In other words, if the same node has expanded twice in different branches of the search tree, A* search might explore both of those branches, thus wasting time.

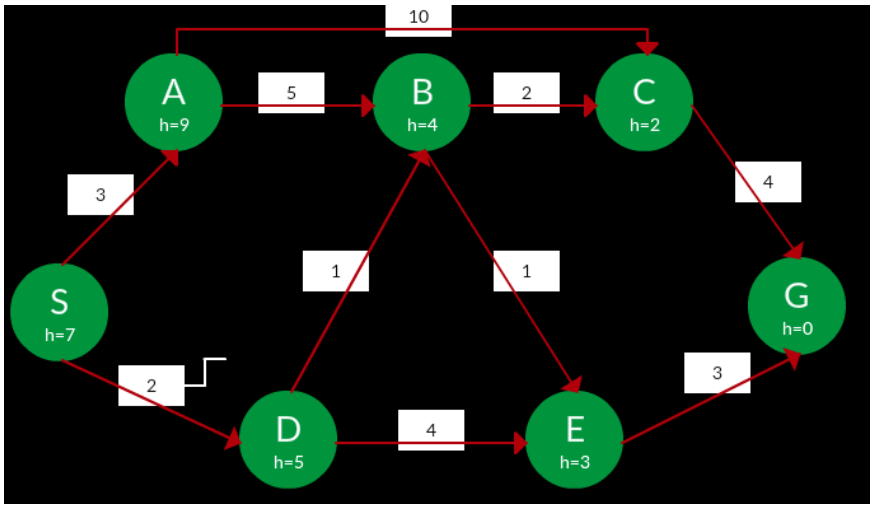
A* Graph Search, or simply Graph Search, removes this limitation by adding this rule: **do not expand the same node more than once.**

Heuristic: Graph search is optimal only when the forward cost between two successive nodes A and B, given by h(A) - h (B) , is less than or equal to the backward cost between those two nodes g(A -> B). This property of graph search heuristic is called consistency.

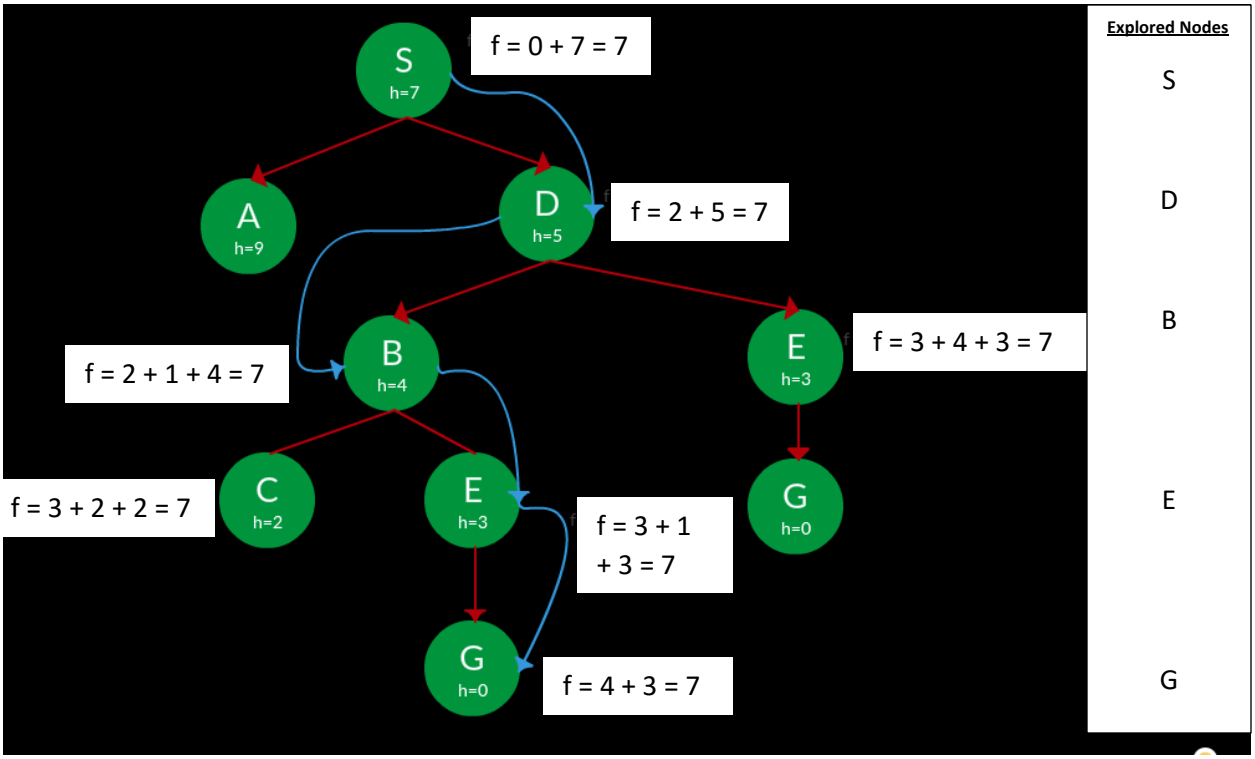
Consistency: $h(A) - h(B) \leq g(A \rightarrow B)$

Example

Question: Use graph search to find path from S to G in the following graph.



Solution: We solve this question pretty much the same way we solved last question, but in this case, we keep a track of nodes explored so that we don't re-explore them.



Path: S -> D -> B -> C -> E -> G

Cost: 7

3.3.3 Other Search Methods

3.3.3.1 Hill Climbing Algorithm

Hill climbing algorithm is a local search algorithm which continuously moves in the direction of increasing elevation/value to find the peak of the mountain or best solution to the problem. It terminates when it reaches a peak value where no neighbor has a higher value.

Hill climbing algorithm is a technique which is used for optimizing the mathematical problems. One of the widely discussed examples of Hill climbing algorithm is Traveling-salesman Problem in which we need to minimize the distance traveled by the salesman.

It is also called greedy local search as it only looks to its good immediate neighbor state and not beyond that.

A node of hill climbing algorithm has two components which are state and value. Hill Climbing is mostly used when a good heuristic is available. In this algorithm, we don't need to maintain and handle the search tree or graph as it only keeps a single current state.

Features of Hill Climbing:

Following are some main features of Hill Climbing Algorithm:

- Generate and Test variant: Hill Climbing is the variant of Generate and Test method. The Generate and Test method produce feedback which helps to decide which direction to move in the search space.
- Greedy approach: Hill-climbing algorithm search moves in the direction which optimizes the cost.
- No backtracking: It does not backtrack the search space, as it does not remember the previous states.

State-space Diagram for Hill Climbing:

The state-space landscape is a graphical representation of the hill-climbing algorithm which is showing a graph between various states of algorithm and Objective function/Cost.

On Y-axis we have taken the function which can be an objective function or cost function, and state-space on the x-axis. If the function on Y-axis is cost then, the goal of search is to find the global minimum and local minimum. If the function of Y-axis is Objective function, then the goal of the search is to find the global maximum and local maximum.

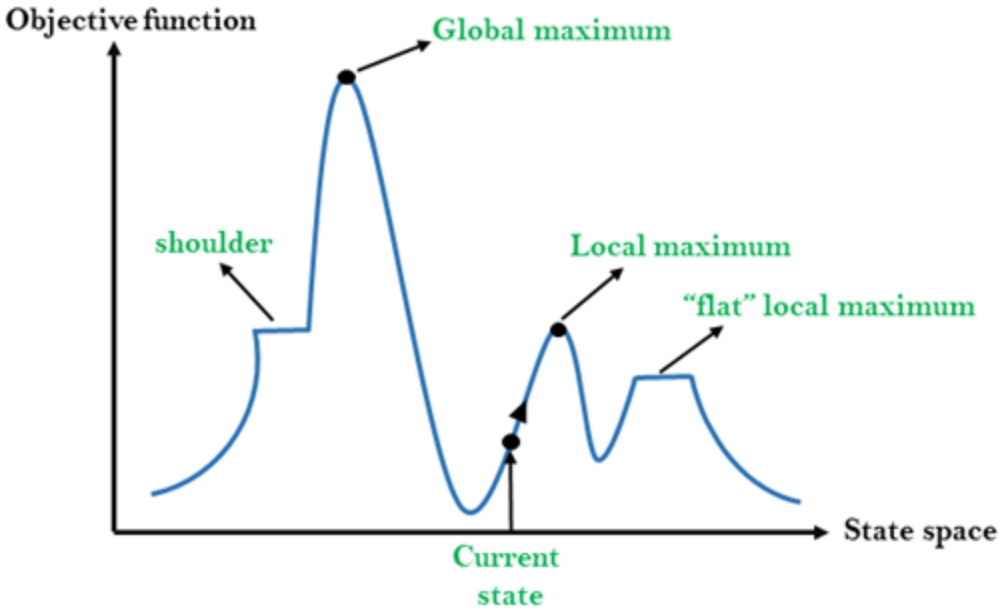


Figure 3.10: State-space Diagram for Hill Climbing

Different regions in the state space landscape:

Local Maximum: Local maximum is a state which is better than its neighbor states, but there is also another state which is higher than it.

Global Maximum: Global maximum is the best possible state of state space landscape. It has the highest value of objective function.

Current state: It is a state in a landscape diagram where an agent is currently present.

Flat local maximum: It is a flat space in the landscape where all the neighbor states of current states have the same value.

Shoulder: It is a plateau region which has an uphill edge.

Types of Hill Climbing Algorithm:

(a) Simple Hill Climbing:

Simple hill climbing is the simplest way to implement a hill climbing algorithm. It only evaluates the neighbor node state at a time and selects the first one which optimizes current cost and set it as a current state. It only checks it's one successor state, and if it finds better than the current state, then move else be in the same state. This algorithm has the following features:

- Less time consuming
- Less optimal solution and the solution is not guaranteed

Algorithm for Simple Hill Climbing:

Step 1: Evaluate the initial state, if it is goal state then return success and Stop.

Step 2: Loop Until a solution is found or there is no new operator left to apply.

Step 3: Select and apply an operator to the current state.

Step 4: Check new state:

i. If it is goal state, then return success and quit.
ii. Else if it is better than the current state then assign new state as a current state.
iii. Else if not better than the current state, then return to step2.

Step 5: Exit.

(b) Steepest-Ascent hill climbing:

The steepest-Ascent algorithm is a variation of simple hill climbing algorithm. This algorithm examines all the neighboring nodes of the current state and selects one neighbor node which is closest to the goal state. This algorithm consumes more time as it searches for multiple neighbors.

Algorithm for Steepest-Ascent hill climbing:

Step 1: Evaluate the initial state, if it is goal state then return success and stop, else make current state as initial state.

Step 2: Loop until a solution is found or the current state does not change.

i. Let SUCC be a state such that any successor of the current state will be better than it.
ii. For each operator that applies to the current state:
   a. Apply the new operator and generate a new state.
   b. Evaluate the new state.
   c. If it is goal state, then return it and quit, else compare it to the SUCC.
   d. If it is better than SUCC, then set new state as SUCC.
   e. If the SUCC is better than the current state, then set current state to SUCC.

Step 5: Exit.

(c) Stochastic hill climbing

Stochastic hill climbing does not examine for all its neighbor before moving. Rather, this search algorithm selects one neighbor node at random and decides whether to choose it as a current state or examine another state.

Problems in Hill Climbing Algorithm:

1. Local Maximum: A local maximum is a peak state in the landscape which is better than each of its neighboring states, but there is another state also present which is higher than the local maximum.

Solution: Backtracking technique can be a solution of the local maximum in state space landscape. Create a list of the promising path so that the algorithm can backtrack the search space and explore other paths as well.

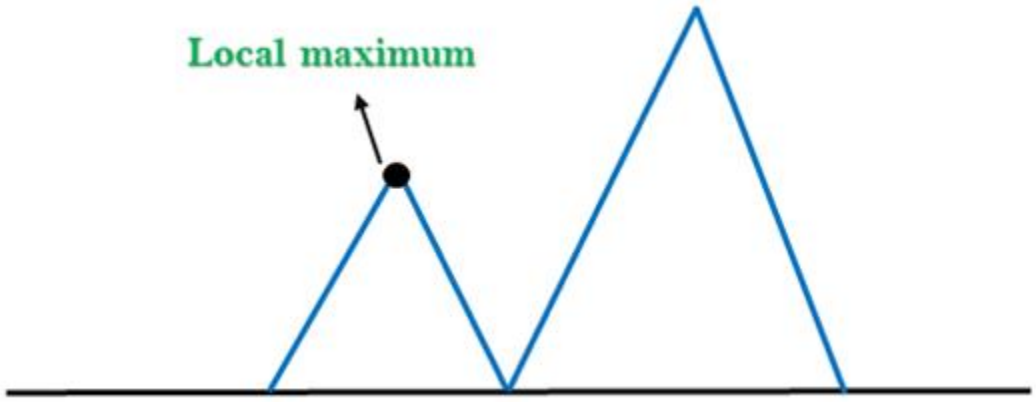


Figure 3.11: Local Maximum (Hill Climbing Algorithm)

2. Plateau: A plateau is the flat area of the search space in which all the neighbor states of the current state contains the same value, because of this algorithm does not find any best direction to move. A hill-climbing search might be lost in the plateau area.

Solution: The solution for the plateau is to take big steps or very little steps while searching, to solve the problem. Randomly select a state which is far away from the current state so it is possible that the algorith m could find non-plateau region.

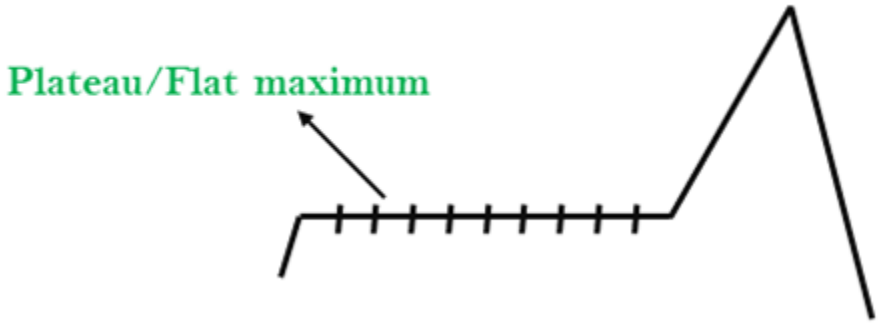


Figure 3.12: Flat Maximum (Hill Climbing Algorithm)

3. Ridges: A ridge is a special form of the local maximum. It has an area which is higher than its surrounding areas, but itself has a slope, and cannot be reached in a single move.

Solution: With the use of bidirectional search, or by moving in different directions, we can improve this problem.

Figure 3.13: Ridge (Hill Climbing Algorithm)

Simulated Annealing:

A hill-climbing algorithm which never makes a move towards a lower value guaranteed to be incomplete because it can get stuck on a local maximum. And if algorithm applies a random walk, by moving a successor, then it may complete but not efficient. Simulated Annealing is an algorithm which yields both efficiency and completeness.

In mechanical term Annealing is a process of hardening a metal or glass to a high temperature then cooling gradually, so this allows the metal to reach a low-energy crystalline state. The same process is used in simulated annealing in which the algorithm picks a random move, instead of picking the best move. If the random move improves the state, then it follows the same path. Otherwise, the algorithm follows the path which has a probability of less than 1 or it moves downhill and chooses another path.

3.3.3.2 Means-Ends Analysis

We have studied the strategies which can reason either in forward or backward, but a mixture of the two directions is appropriate for solving a complex and large problem. Such a mixed strategy, make it possible that first to solve the major part of a problem and then go back and solve the small problems arise during combining the big parts of the problem. Such a technique is called Means-Ends Analysis.

Means-Ends Analysis is problem-solving techniques used in Artificial intelligence for limiting search in AI programs. It is a mixture of Backward and forward search technique. The MEA technique was first introduced in 1961 by Allen Newell, and Herbert A. Simon in their problem-solving computer program, which was named as General Problem Solver (GPS). The MEA analysis process centered on the evaluation of the difference between the current state and goal state.

How means-ends analysis Works:

The means-ends analysis process can be applied recursively for a problem. It is a strategy to control search in problem-solving. Following are the main Steps which describes the working of MEA technique for solving a problem.

a. First, evaluate the difference between Initial State and final State.
b. Select the various operators which can be applied for each difference.
c. Apply the operator at each difference, which reduces the difference between the current state and goal state.

Operator Subgoaling

In the MEA process, we detect the differences between the current state and goal state. Once these differences occur, then we can apply an operator to reduce the differences. But sometimes it is possible that an operator cannot be applied to the current state. So we create the subproblem of the current state, in which operator can be applied, such type of backward chaining in which operators are selected, and then sub goals are set up to establish the preconditions of the operator is called Operator Subgoaling.

Algorithm for Means-Ends Analysis:

Let's we take Current state as CURRENT and Goal State as GOAL, then following are the steps for the MEA algorithm.

Step 1: Compare CURRENT to GOAL, if there are no differences between both then return Success and Exit.

Step 2: Else, select the most significant difference and reduce it by doing the following steps until the success or failure occurs.

a. Select a new operator O which is applicable for the current difference, and if there is no such operator, then signal failure.
b. Attempt to apply operator O to CURRENT. Make a description of two states.

   i) O-Start, a state in which O?s preconditions are satisfied.

   ii) O-Result, the state that would result if O were applied In O-start.

c. If

(First-Part <------ MEA (CURRENT, O-START)

And

(LAST-Part <----- MEA (O-Result, GOAL), are successful, then signal Success and return the result of combining FIRST-PART, O, and LAST-PART.

The above-discussed algorithm is more suitable for a simple problem and not adequate for solving complex problems.

Example of Mean-Ends Analysis

Let's take an example where we know the initial state and goal state as given below. In this problem, we need to get the goal state by finding differences between the initial state and goal state and applying operators.

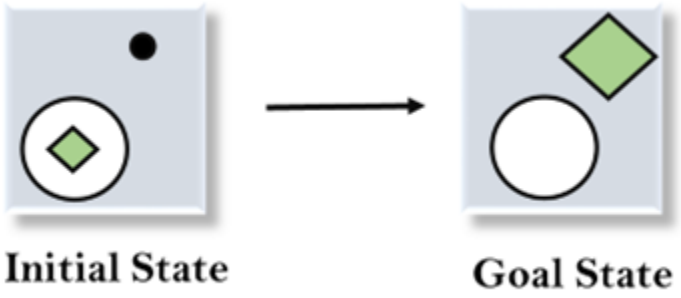


Figure 3.14: State Transition (Means-Ends Analysis)

Solution:

To solve the above problem, we will first find the differences between initial states and goal states, and for each difference, we will generate a new state and will apply the operators. The operators we have for this problem are:

a. Move
b. Delete
c. Expand

1. Evaluating the initial state: In the first step, we will evaluate the initial state and will compare the initial and Goal state to find the differences between both states.

Figure 3.15: Initial State (Means-Ends Analysis)

2. Applying Delete operator: As we can check the first difference is that in goal state there is no dot symbol which is present in the initial state, so, first we will apply the Delete operator to remove this dot.

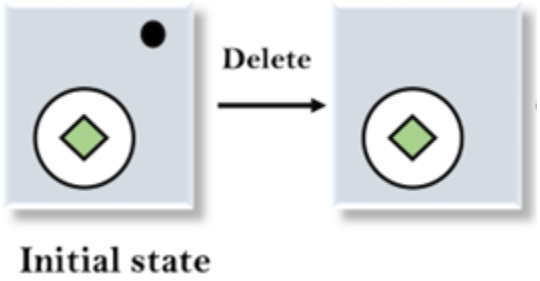


Figure 3.16: Applying Delete operator (Means-Ends Analysis)

3. Applying Move Operator: After applying the Delete operator, the new state occurs which we will again compare with goal state. After comparing these states, there is another difference that is the square is outside the circle, so, we will apply the Move Operator.

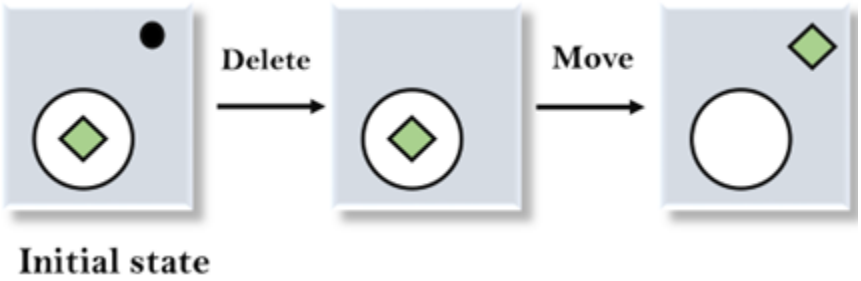


Figure 3.17: Applying Move operator (Means-Ends Analysis)

4. Applying Expand Operator: Now a new state is generated in the third step, and we will compare this state with the goal state. After comparing the states there is still one difference which is the size of the square, so, we will apply Expand operator, and finally, it will generate the goal state.

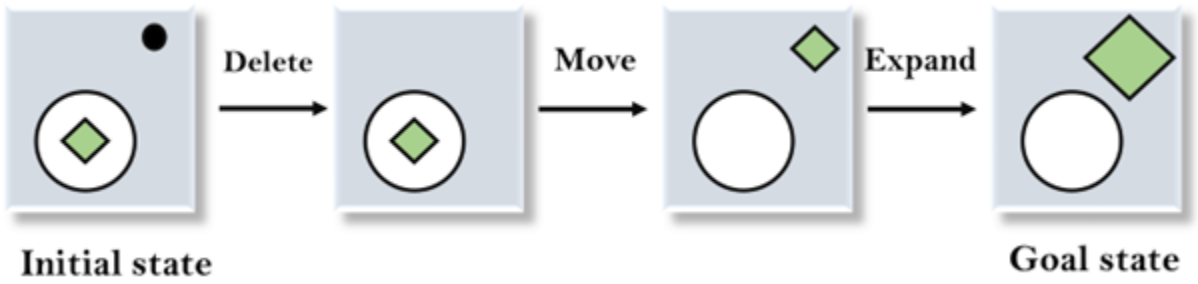


Figure 3.18: Applying Expand operator (Means-Ends Analysis)

4.0 Conclusion

We have explained all about the search algorithms in AI. The unit presented the classification of AI search algorithms into two classes (informed and uninformed search algorithms). The basic concept behind each technique presented in this unit was elaborated using simple examples. This list is not in any way an exhaustive list as this area is a very active research area. With a proper knowledge of this unit, you should be able to take on any search problems in AI.

5.0 Summary

This unit has introduced methods that an agent can use to select actions in environments that are deterministic, observable, static, and completely known. In such cases, the agent can construct sequences of actions that achieve its goals; this process is called search. We can summarize the major learning points of the unit as follows:

- Before an agent can start searching for solutions, a goal must be identified and a well-defined problem must be formulated.
- A problem consists of five parts: the initial state, a set of actions, a transition model describing the results of those actions, a goal test function, and a path cost function. The environment of the problem is represented by a state space. A path through the state space from the initial state to a goal state is a solution.
- Search algorithms treat states and actions as atomic: they do not consider any internal structure they might possess.
- A general TREE-SEARCH algorithm considers all possible paths to find a solution, whereas a GRAPH-SEARCH algorithm avoids consideration of redundant paths.
- Search algorithms are judged on the basis of completeness, optimality, time complexity, and space complexity. Complexity depends on b, the branching factor in the state space, and d, the depth of the shallowest solution.

- Uninformed search methods have access only to the problem definition. The basic algorithms are as follows:
  - Breadth-first search expands the shallowest nodes first; it is complete, optimal for unit step costs, but has exponential space complexity.
  - Uniform-cost search expands the node with lowest path cost, g(n), and is optimal for general step costs.
  - Depth-first search expands the deepest unexpanded node first. It is neither complete nor optimal, but has linear space complexity. Depth-limited search adds a depth bound.
  - Iterative deepening search calls depth-first search with increasing depth limits until a goal is found. It is complete, optimal for unit step costs, has time complexity comparable to breadth-first search, and has linear space complexity.
  - Bidirectional search can enormously reduce time complexity, but it is not always applicable and may require too much space.
- Informed search methods may have access to a heuristic function h(n) that estimates the cost of a solution from n.
  - The generic best-first search algorithm selects a node for expansion according to an evaluation function.
  - Greedy best-first search expands nodes with minimal h(n). It is not optimal but is often efficient.
  - A∗ search expands nodes with minimal f(n) = g(n) + h(n). A∗ is complete and optimal, provided that h(n) is admissible (for TREE-SEARCH) or consistent (for GRAPH-SEARCH). The space complexity of A∗ is still prohibitive.
  - RBFS (recursive best-first search) and SMA∗ (simplified memory-bounded A∗) are robust, optimal search algorithms that use limited amounts of memory; given enough time, they can solve problems that A∗ cannot solve because it runs out of memory.
- The performance of heuristic search algorithms depends on the quality of the heuristic function. One can sometimes construct good heuristics by relaxing the problem definition, by storing precomputed solution costs for subproblems in a pattern database, or by learning from experience with the problem class. We hope you enjoyed this unit. Now, let us attempt the questions below.

6.0 Tutor Marked Assignment

i. The missionaries and cannibals problem is usually stated as follows. Three missionaries and three cannibals are on one side of a river, along with a boat that can hold one or two people. Find a way to get everyone to the other side without ever leaving a group of missionaries in one place outnumbered by the cannibals in that place. This problem is famous in AI because it was the subject of the first paper that approached problem formulation from an analytical viewpoint (Amarel, 1968).
   a. Formulate the problem precisely, making only those distinctions necessary to ensure a valid solution. Draw a diagram of the complete state space.
   b. Implement and solve the problem optimally using an appropriate search algorithm. Is it a good idea to check for repeated states?

c. Why do you think people have a hard time solving this puzzle, given that the state space is so simple?

ii. Define in your own words the following terms: state, state space, search tree, search node, goal, action, transition model, and branching factor.

7.0 References/Further Readings

1) Russell, S., & Norvig, P. (2005). AI a modern approach. Learning, 2(3), 4.
2) Rich E., & Knight K. (1991) Artificial Intelligence, McGraw-Hill, New York, NY
3) Search Algorithms in AI https://www.geeksforgeeks.org/search-algorithms-in-ai/ retrieved 7/8/2019
4) Kanal, L., & Kumar, V. (Eds.). (2012). Search in artificial intelligence. Springer Science & Business Media. Search Algorithms in Artificial Intelligence https://www.javatpoint.com/search-algorithms-in-ai retrieved 7/8/2019

Unit Three: Adversarial Search (Game Tree),

CONTENT

1.0 Introduction

Adversarial search is a search, where we examine the problem which arises when we try to plan ahead of the world and other agents are planning against us.

2.0 Objectives

At the end of this unit, you should be able to do the following:

- Outline the different type of games
- Define what a game tree is
- Explain the operation of a game tree
- Explain the working operation of the mini-max algorithm
- Outline the properties of the mini-max algorithm
- Point out the limitation of the mini-max algorithm
- Explain the working operation of Alpha-beta pruning
- Outline and explain the two-move ordering in Alpha-beta pruning
- Mention the rules to finding good ordering in Alpha-beta pruning

3.0 Main Content

In the previous unit, we have studied the search strategies which are only associated with a single agent that aims to find the solution which often expressed in the form of a sequence of actions. But there might be some situations where more than one agent is searching for the solution in the same search space, and this situation usually occurs in game playing.

The environment with more than one agent is termed as multi-agent environment, in which each agent is an opponent of other agent and playing against each other. Each agent needs to consider the action of other agent and effect of that action on their performance.

So, Searches in which two or more players with conflicting goals are trying to explore the same search space for the solution, are called adversarial searches, often known as Games. Games are modeled as a Search problem and heuristic evaluation function, and these are the two main factors which help to model and solve games in AI.

3.1 Types of Games in AI

	Deterministic	Chance Moves
Perfect information	Chess, Checkers, go, Othello	Backgammon, monopoly
Imperfect information	Battleships, blind, tic-tac-toe	Bridge, poker, scrabble, nuclear war

a. Perfect information: A game with the perfect information is that in which agents can look into the complete board. Agents have all the information about the game, and they can see each other moves also. Examples are Chess, Checkers, Go, etc.
b. Imperfect information: If in a game, agents do not have all information about the game and not aware with what's going on, such type of games are called the game with imperfect information, such as tic-tac-toe, Battleship, blind, Bridge, etc.
c. Deterministic games: Deterministic games are those games which follow a strict pattern and set of rules for the games, and there is no randomness associated with them. Examples are chess, Checkers, Go, tic-tac-toe, etc.
d. Non-deterministic games: Non-deterministic are those games which have various unpredictable events and has a factor of chance or luck. This factor of chance or luck is introduced by either dice or cards. These are random, and each action response is not fixed. Such games are also called as stochastic games. Example: Backgammon, Monopoly, Poker, etc.

3.2 Zero-Sum Game

Zero-sum games are adversarial search which involves pure competition. In Zero-sum game each agent's gain or loss of utility is exactly balanced by the losses or gains of utility of another agent. One player of the game try to maximize one single value, while other player tries to minimize it. Each move by one player in the game is called as ply. Chess and tic-tac-toe are examples of a Zero-sum game.

Zero-sum game: Embedded thinking

The Zero-sum game involved embedded thinking in which one agent or player is trying to figure out:

- What to do.
- How to decide the move
- Needs to think about his opponent as well
- The opponent also thinks what to do

Each of the players is trying to find out the response of his opponent to their actions. This requires embedded thinking or backward reasoning to solve the game problems in AI.

Formalization of the problem

A game can be defined as a type of search in AI which can be formalized of the following elements:

- **Initial state:** It specifies how the game is set up at the start.
- **Player(s):** It specifies which player has moved in the state space.
- **Action(s):** It returns the set of legal moves in state space.
- **Result (s, a):** It is the transition model, which specifies the result of moves in the state space.
- **Terminal-Test(s):** Terminal test is true if the game is over, else it is false at any case. The state where the game ends is called terminal states.
- **Utility (s, p):** A utility function gives the final numeric value for a game that ends in terminal states s for player p. It is also called payoff function. For Chess, the outcomes are a win, loss, or draw and its payoff values are +1, 0, ½. And for tic-tac-toe, utility values are +1, -1, and 0.

3.3 Game tree

A game tree is a type of recursive search function that examines all possible moves of a strategy game, and their results, in an attempt to ascertain the optimal move. They are very useful for Artificial Intelligence in scenarios that do not require real-time decision making and have a relatively low number of possible choices per play. The most commonly-cited example is chess, but they are applicable to many situations.

A game tree is a tree where nodes of the tree are the game states and Edges of the tree are the moves by players. Game tree involves initial state, actions function, and result Function.

Example: Tic-Tac-Toe game tree:

The following figure is showing part of the game-tree for tic-tac-toe game. Following are some key points of the game:

- There are two players MAX and MIN.
- Players have an alternate turn and start with MAX.
- MAX maximizes the result of the game tree
- MIN minimizes the result.

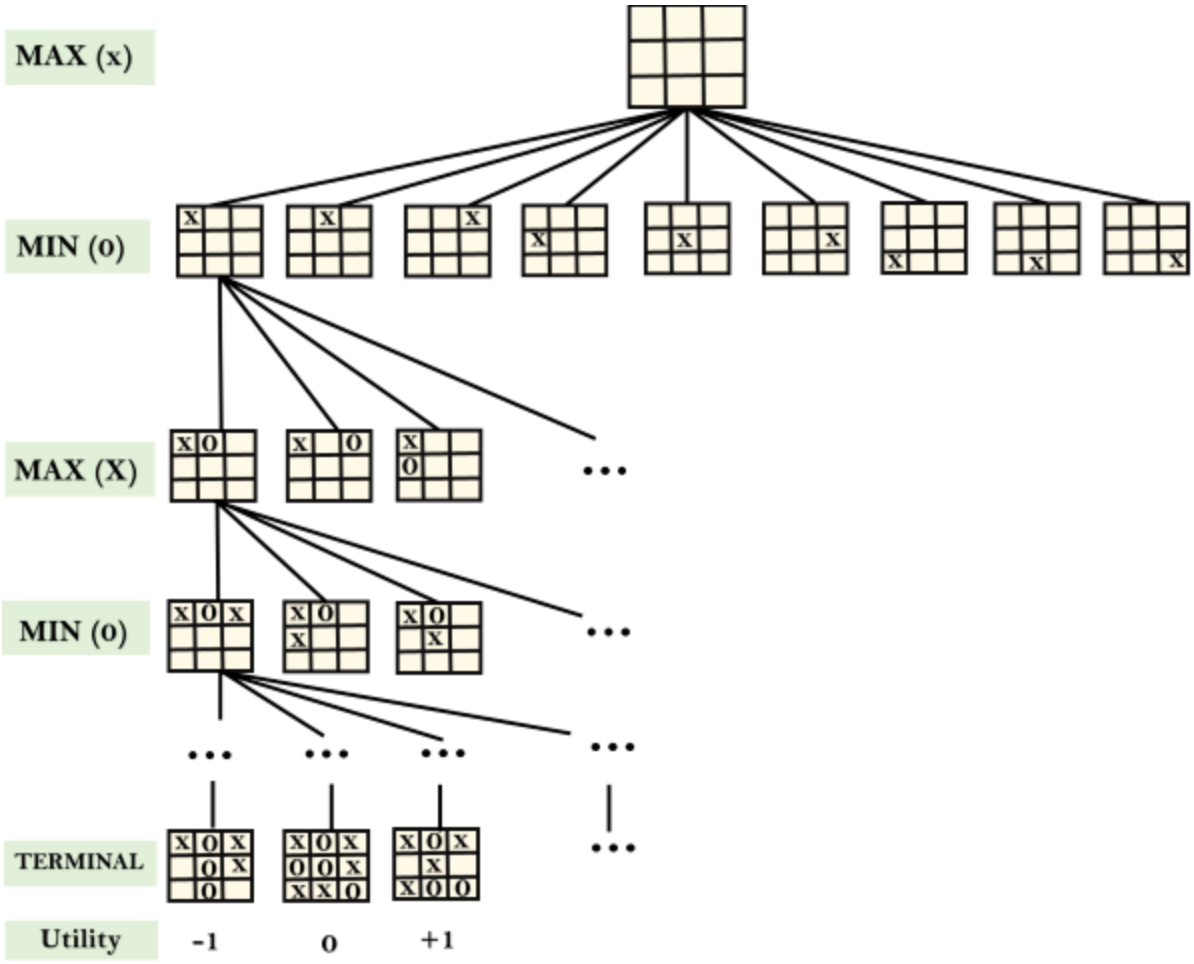


Figure 3.1: Tic-Tac-Toe game tree

Example Explanation:

- From the initial state, MAX has 9 possible moves as he starts first. MAX place x and MIN place o, and both player plays alternatively until we reach a leaf node where one player has three in a row or all squares are filled.
- Both players will compute each node, minimax, the minimax value which is the best achievable utility against an optimal adversary.
- Suppose both the players are well aware of the tic-tac-toe and playing the best play. Each player is doing his best to prevent another one from winning. MIN is acting against Max in the game.
- So in the game tree, we have a layer of Max, a layer of MIN, and each layer is called as Ply. Max place x, then MIN puts o to prevent Max from winning, and this game continues until the terminal node.
- In this either MIN wins, MAX wins, or it's a draw. This game-tree is the whole search space of possibilities that MIN and MAX are playing tic-tac-toe and taking turns alternately.

Hence adversarial Search for the minimax procedure works as follows:

- It aims to find the optimal strategy for MAX to win the game.

- It follows the approach of Depth-first search.
- In the game tree, optimal leaf node could appear at any depth of the tree.
- Propagate the minimax values up to the tree until the terminal node discovered.

In a given game tree, the optimal strategy can be determined from the minimax value of each node, which can be written as MINIMAX(n). MAX prefer to move to a state of maximum value and MIN prefer to move to a state of minimum value then:

For a state S MINIMAX(s) =

$$\begin{cases} \text{UTILITY}(s) & \text{If TERMINAL-TEST}(s) \\ \max_{a \in \text{Actions}(s)} \text{MINIMAX}(\text{RESULT}(s, a)) & \text{If PLAYER}(s) = \text{MAX} \\ \min_{a \in \text{Actions}(s)} \text{MINIMAX}(\text{RESULT}(s, a)) & \text{If PLAYER}(s) = \text{MIN}. \end{cases}$$

3.3.1 Mini-Max Algorithm

Mini-max algorithm is a recursive or backtracking algorithm which is used in decision-making and game theory. It provides an optimal move for the player assuming that opponent is also playing optimally. The algorithm uses recursion to search through the game-tree. It is mostly used for game playing in AI. Such as Chess, Checkers, tic-tac-toe, go, and various tow-players game. This Algorithm computes the minimax decision for the current state.

In this algorithm two players play the game, one is called MAX and other is called MIN. Both the players fight it as the opponent player gets the minimum benefit while they get the maximum benefit. Both Players of the game are opponent of each other, where MAX will select the maximized value and MIN will select the minimized value. The minimax algorithm performs a depth-first search algorithm for the exploration of the complete game tree. The minimax algorithm proceeds all the way down to the terminal node of the tree, then backtrack the tree as the recursion.

3.3.1.1 Pseudo-code for Min-Max Algorithm

```
function minimax(node, depth, maximizingPlayer) is
if depth ==0 or node is a terminal node then
return static evaluation of node

if MaximizingPlayer then      // for Maximizer Player
maxEva= -infinity
 for each child of node do
 eva= minimax(child, depth-1, false)
maxEva= max(maxEva,eva)       //gives Maximum of the values
return maxEva

else                  // for Minimizer player
 minEva= +infinity
 for each child of node do
```

eva= minimax(child, depth-1, true)
minEva= min(minEva, eva) //gives minimum of the values
return minEva

Initial call:

Minimax (node, 3, true)

3.3.1.2 Working of Min-Max Algorithm:

The working of the minimax algorithm can be easily described using an example. Below we have taken an example of game-tree which is representing the two-player game. In this example, there are two players one is called Maximizer and other is called Minimizer. Maximizer will try to get the Maximum possible score, and Minimizer will try to get the minimum possible score. This algorithm applies DFS, so in this game-tree, we have to go all the way through the leaves to reach the terminal nodes. At the terminal node, the terminal values are given so we will compare those value and backtrack the tree until the initial state occurs. Following are the main steps involved in solving the two-player game tree:

Step-1: In the first step, the algorithm generates the entire game-tree and apply the utility function to get the utility values for the terminal states. In the below tree diagram, let's take A is the initial state of the tree. Suppose maximizer takes first turn which has worst-case initial value =- infinity, and minimizer will take next turn which has worst-case initial value = +infinity.

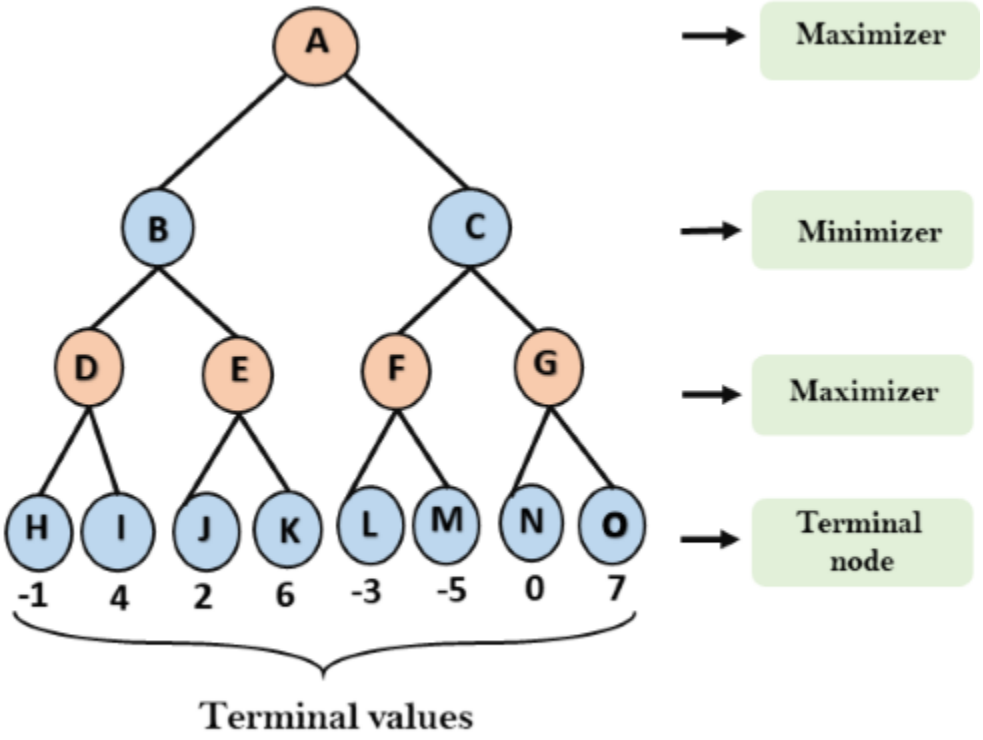


Figure 3.2: Min-Max Algorithm

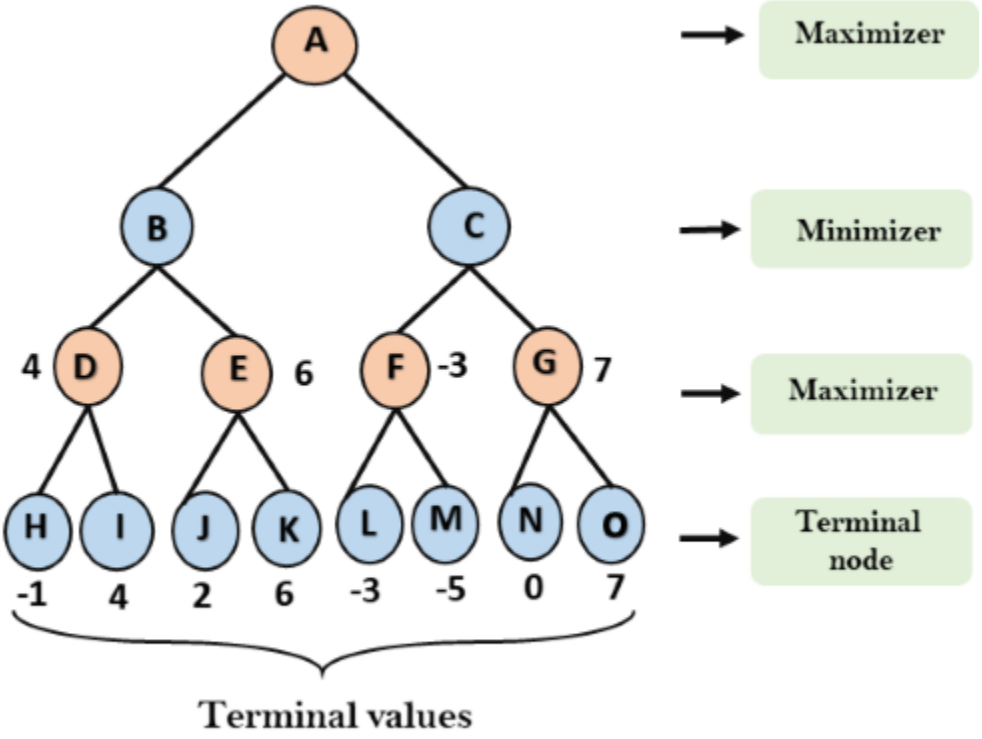
Step 2: Now, first we find the utilities value for the Maximizer, its initial value is -∞, so we will compare each value in terminal state with initial value of Maximizer and determines the higher nodes values. It will find the maximum among the all.

For node D max(-1,- -∞) => max(-1,4)= 4

For Node E max(2, -∞) => max(2, 6)= 6

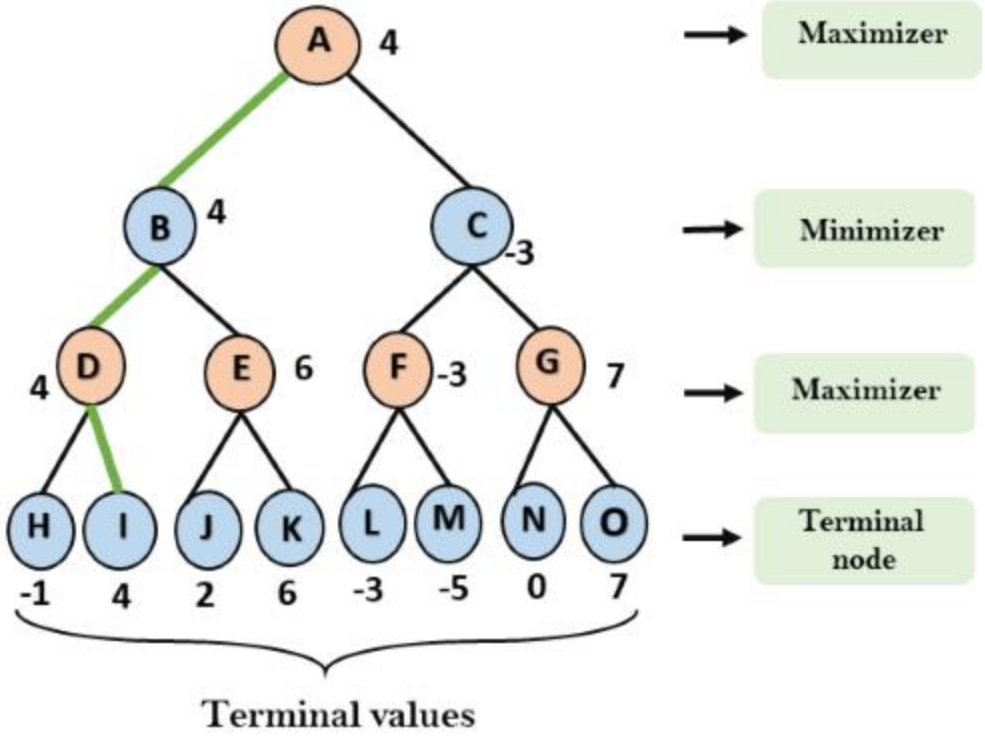
For Node F max(-3, -∞) => max(-3,-5) = -3

For node G max(0, -∞) = max(0, 7) = 7



Step 3: Now it's a turn for Maximizer, and it will again choose the maximum of all nodes value and find the maximum value for the root node. In this game tree, there are only 4 layers, hence we reach immediately to the root node, but in real games, there will be more than 4 layers.

For node A max(4, -3)= 4



That was the complete workflow of the minimax two player game.

3.3.1.3 Properties of Mini-Max algorithm:

- **Completeness**: Min-Max algorithm is Complete. It will definitely find a solution (if exist), in the finite search tree.
- **Optimality**: Min-Max algorithm is optimal if both opponents are playing optimally.
- **Time complexity:** As it performs DFS for the game-tree, so the time complexity of Min-Max algorithm is O(bm), where b is branching factor of the game-tree, and m is the maximum depth of the tree.
- **Space Complexity:** Space complexity of Mini-max algorithm is also similar to DFS which is O(bm).

3.3.1.4 Limitation of the minimax Algorithm:

The main drawback of the minimax algorithm is that it gets really slow for complex games such as Chess, go, etc. This type of games has a huge branching factor, and the player has lots of choices to decide. This limitation of the minimax algorithm can be improved from alpha-beta pruning which we have discussed in the next topic.

3.3.2 Alpha-Beta Pruning

Alpha-beta pruning is a modified version of the minimax algorithm. It is an optimization technique for the minimax algorithm. As we have seen in the minimax search algorithm that the number of game states it has to examine are exponential in depth of the tree. Since we cannot eliminate the exponent,

but we can cut it to half. Hence there is a technique by which without checking each node of the game tree we can compute the correct minimax decision, and this technique is called pruning. This involves two threshold parameter Alpha and beta for future expansion, so it is called alpha-beta pruning. It is also called as Alpha-Beta Algorithm.

Alpha-beta pruning can be applied at any depth of a tree, and sometimes it not only prune the tree leaves but also entire sub-tree.

The two-parameter can be defined as:

a. Alpha: The best (highest-value) choice we have found so far at any point along the path of Maximizer. The initial value of alpha is -∞.
b. Beta: The best (lowest-value) choice we have found so far at any point along the path of Minimizer. The initial value of beta is +∞.

The Alpha-beta pruning to a standard minimax algorithm returns the same move as the standard algorithm does, but it removes all the nodes which are not really affecting the final decision but making algorithm slow. Hence by pruning these nodes, it makes the algorithm fast.

Condition for Alpha-beta pruning

The main condition which required for alpha-beta pruning is $\alpha \geq \beta$

Key points about alpha-beta pruning:

- The Max player will only update the value of alpha.
- The Min player will only update the value of beta.
- While backtracking the tree, the node values will be passed to upper nodes instead of values of alpha and beta.
- We will only pass the alpha, beta values to the child nodes.

3.3.2.1 Pseudo-code for Alpha-beta Pruning

```
function minimax(node, depth, alpha, beta, maximizingPlayer) is
if depth ==0 or node is a terminal node then
return static evaluation of node

if MaximizingPlayer then      // for Maximizer Player
  maxEva= -infinity
  for each child of node do
  eva= minimax(child, depth-1, alpha, beta, False)
 maxEva= max(maxEva, eva)
 alpha= max(alpha, maxEva)
  if beta<=alpha
break
return maxEva
```

```
else                    // for Minimizer player
  minEva= +infinity
  for each child of node do
  eva= minimax(child, depth-1, alpha, beta, true)
  minEva= min(minEva, eva)
  beta= min(beta, eva)
   if beta<=alpha
 break
return minEva
```

3.3.2.2 Working of Alpha-Beta Pruning

Let's take an example of two-player search tree to understand the working of Alpha-beta pruning

Step 1: At the first step the, Max player will start first move from node A where α= -∞ and β= +∞, these value of alpha and beta passed down to node B where again α= -∞ and β= +∞, and Node B passes the same value to its child D.

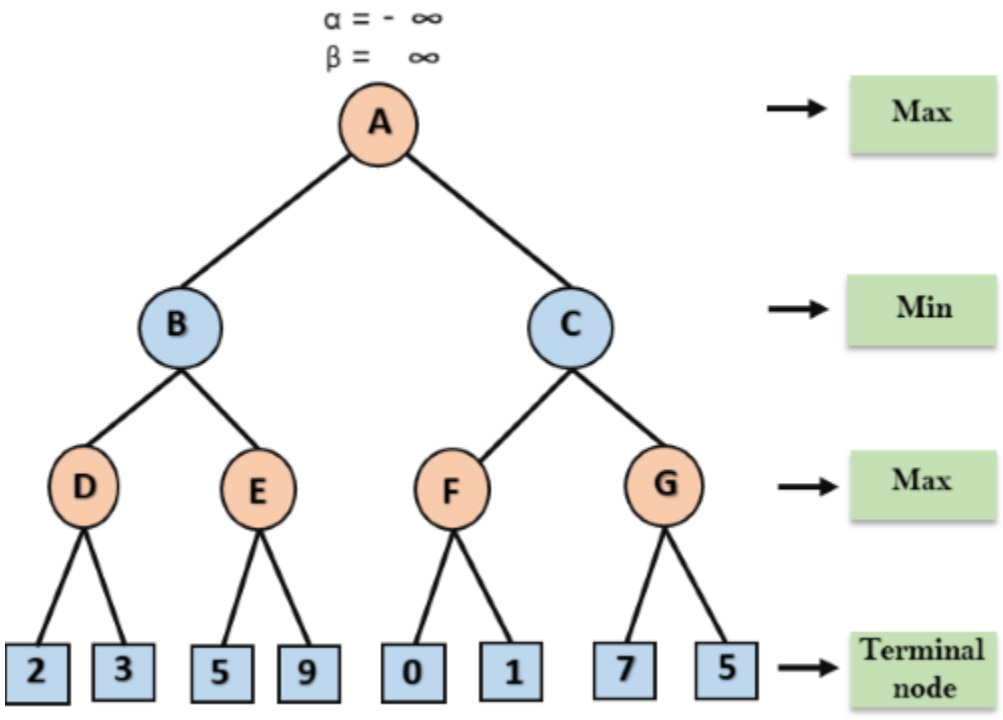
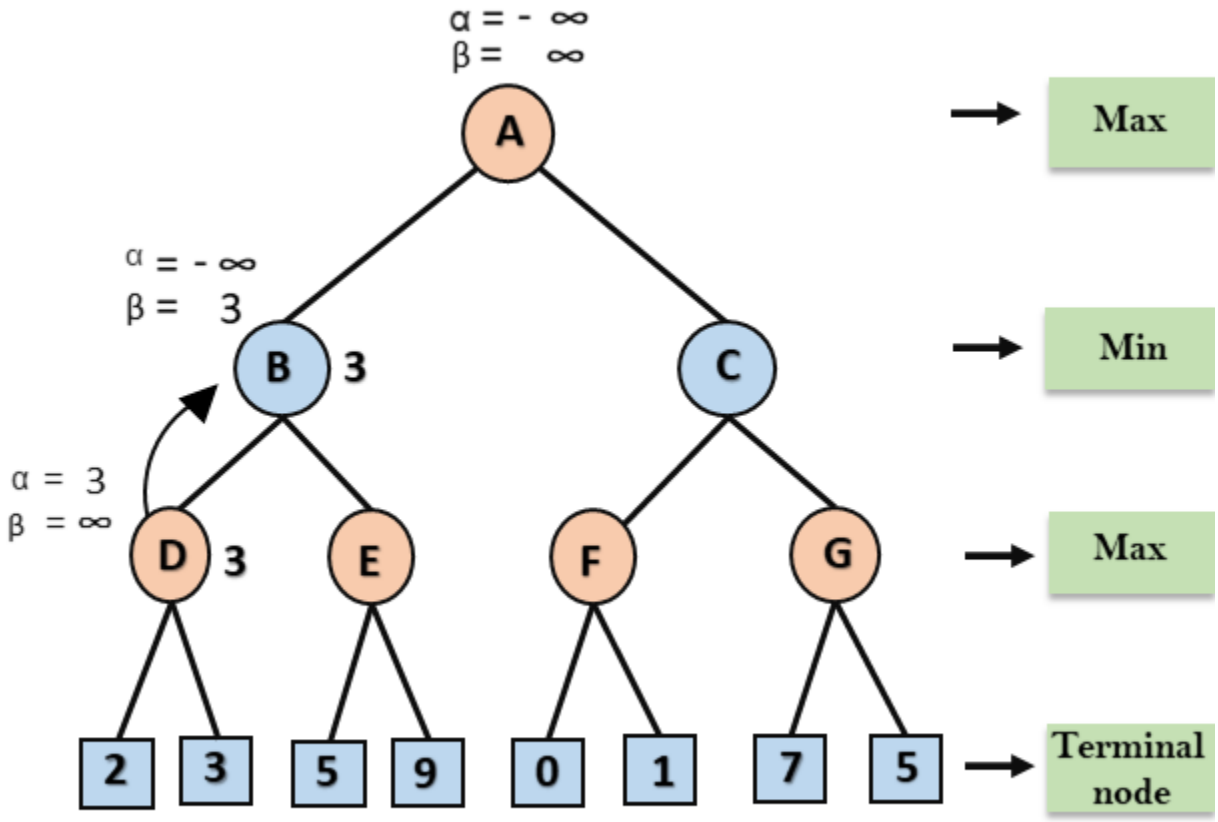


Figure 3.3: Alpha-Beta Pruning

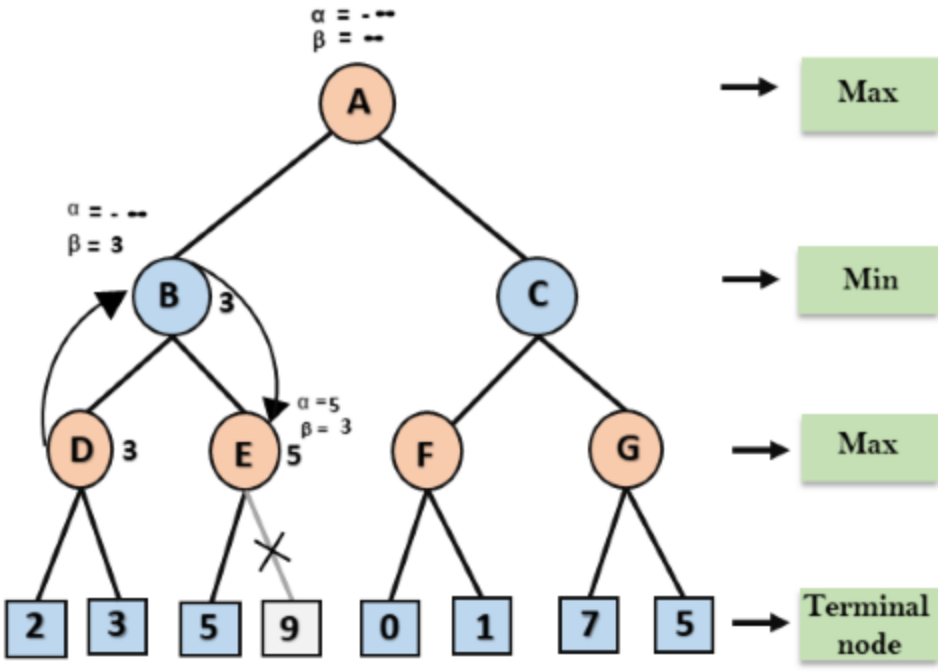
Step 2: At Node D, the value of α will be calculated as its turn for Max. The value of α is compared with firstly 2 and then 3, and the max (2, 3) = 3 will be the value of α at node D and node value will also 3.

Step 3: Now algorithm backtrack to node B, where the value of β will change as this is a turn of Min, Now β= +∞, will compare with the available subsequent nodes value, i.e. min (∞, 3) = 3, hence at node B now α= -∞, and β= 3.



In the next step, algorithm traverse the next successor of Node B which is node E, and the values of α= -∞, and β= 3 will also be passed.

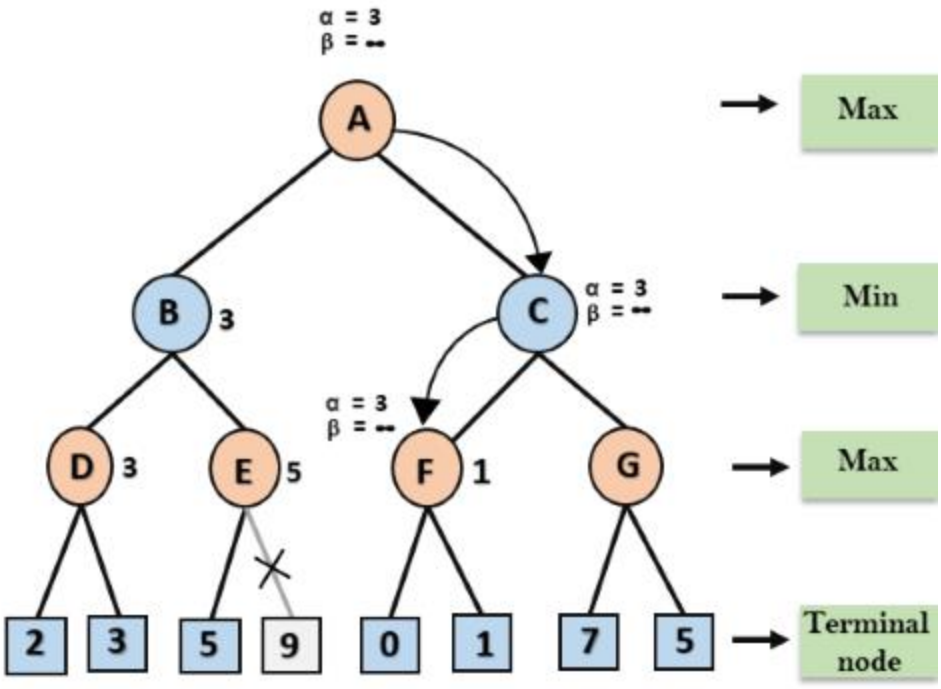
Step 4: At node E, Max will take its turn, and the value of alpha will change. The current value of alpha will be compared with 5, so max (-∞, 5) = 5, hence at node E α= 5 and β= 3, where α>=β, so the right successor of E will be pruned, and algorithm will not traverse it, and the value at node E will be 5.



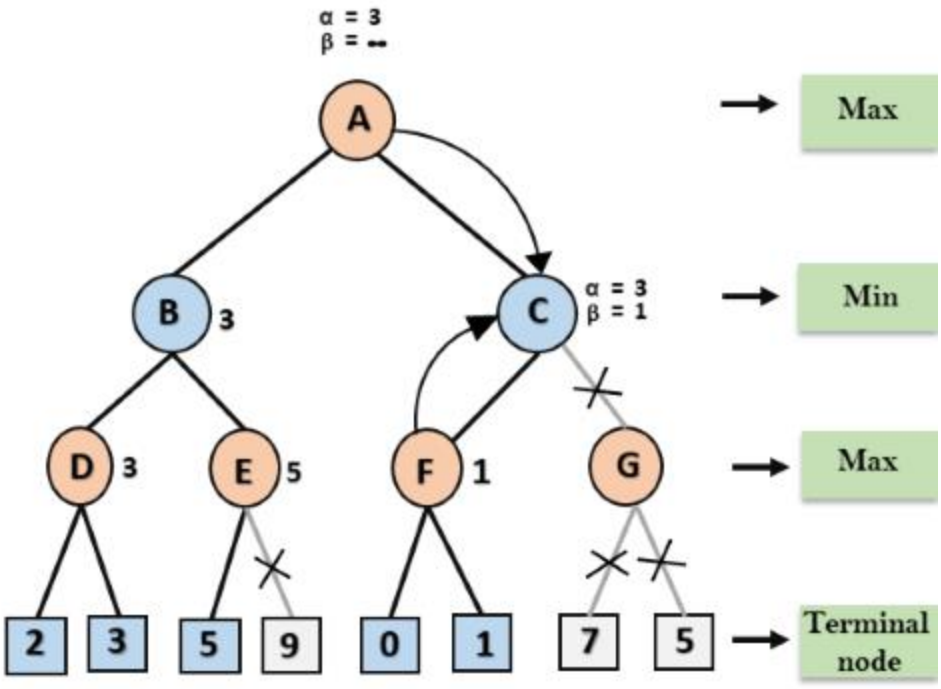
Step 5: At next step, algorithm again backtrack the tree, from node B to node A. At node A, the value of alpha will be changed the maximum available value is 3 as max (-∞, 3)= 3, and β= +∞, these two values now passes to right successor of A which is Node C.

At node C, α=3 and β= +∞, and the same values will be passed on to node F.

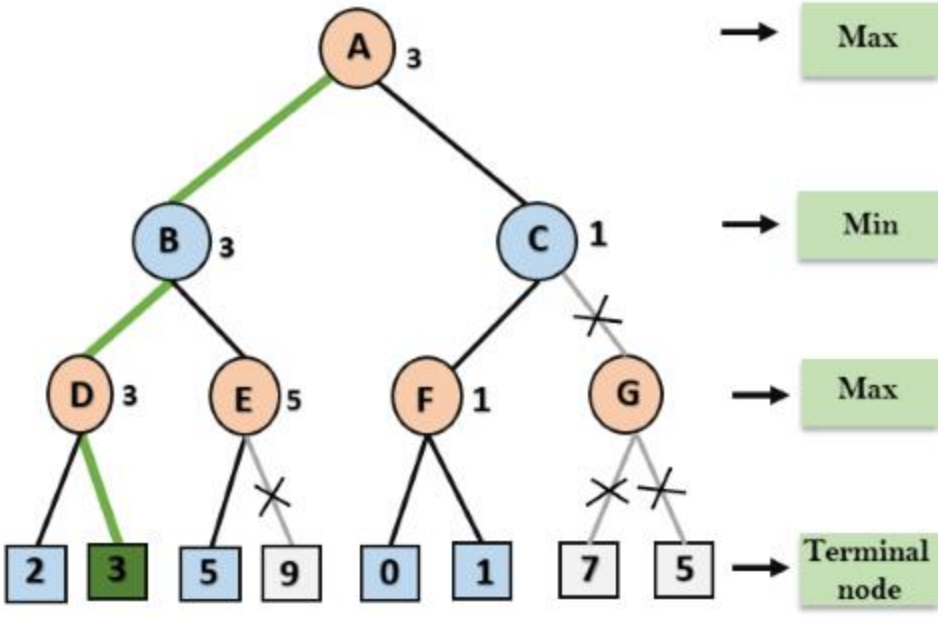
Step 6: At node F, again the value of α will be compared with left child which is 0, and max(3,0)= 3, and then compared with right child which is 1, and max(3,1)= 3 still α remains 3, but the node value of F will become 1.



Step 7: Node F returns the node value 1 to node C, at C α= 3 and β= +∞, here the value of beta will be changed, it will compare with 1 so min (∞, 1) = 1. Now at C, α=3 and β= 1, and again it satisfies the condition α>=β, so the next child of C which is G will be pruned, and the algorithm will not compute the entire sub-tree G.



Step 8: C now returns the value of 1 to A here the best value for A is max (3, 1) = 3. Following is the final game tree which is the showing the nodes which are computed and nodes which has never computed. Hence the optimal value for the maximizer is 3 for this example.



3.3.2.3 Move Ordering in Alpha-Beta pruning:

The effectiveness of alpha-beta pruning is highly dependent on the order in which each node is examined. Move order is an important aspect of alpha-beta pruning.

It can be of two types:

a. **Worst ordering:** In some cases, alpha-beta pruning algorithm does not prune any of the leaves of the tree, and works exactly as minimax algorithm. In this case, it also consumes more time because of alpha-beta factors, such a move of pruning is called worst ordering. In this case, the best move occurs on the right side of the tree. The time complexity for such an order is O(bm).
b. **Ideal ordering:** The ideal ordering for alpha-beta pruning occurs when lots of pruning happens in the tree, and best moves occur at the left side of the tree. We apply DFS hence it first search left of the tree and go deep twice as minimax algorithm in the same amount of time. Complexity in ideal ordering is $O(b^{m/2})$.

3.3.2.4 Rules to find good ordering:

Following are some rules to find good ordering in alpha-beta pruning:

- Occur the best move from the shallowest node.
- Order the nodes in the tree such that the best nodes are checked first.

- Use domain knowledge while finding the best move. Ex: for Chess, try order: captures first, then threats, then forward moves, backward moves.
- We can bookkeep the states, as there is a possibility that states may repeat.

4.0 Conclusion

We have looked at a variety of games to understand what optimal play means and to understand how to play well in practice. The most important ideas are as follows:

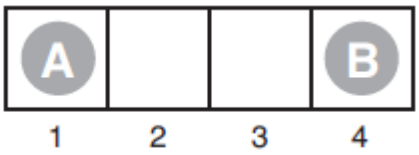
- A game can be defined by the initial state (how the board is set up), the legal actions in each state, the result of each action, a terminal test (which says when the game is over), and a utility function that applies to terminal states.
- In two-player zero-sum games with perfect information, the minimax algorithm can select optimal moves by a depth-first enumeration of the game tree.
- The alpha–beta search algorithm computes the same optimal move as minimax, but achieves much greater efficiency by eliminating subtrees that are provably irrelevant.
- Usually, it is not feasible to consider the whole game tree (even with alpha–beta), so we. need to cut the search off at some point and apply a heuristic evaluation function that estimates the utility of a state.
- Many game programs precompute tables of best moves in the opening and endgame so that they can look up a move rather than search.
- Games of chance can be handled by an extension to the minimax algorithm that evaluates a chance node by taking the average utility of all its children, weighted by the probability of each child.

5.0 Summary

We hope you enjoyed this unit. This unit discusses game trees as used in AI and their categories. Now, let us attempt the questions below.

6.0 Tutor Marked Assignment

1. Consider the two-player game described in Figure below.



The starting position of a simple game. Player A moves first. The two players take turns moving, and each player must move his token to an open adjacent space in either direction. If the opponent occupies an adjacent space, then a player may jump over the opponent to the next open space if any. (For example, if A is on 3 and B is on 2, then A may move back to 1.) The game ends when one player reaches the opposite end of the board. If player A reaches space 4 first, then the value of the game to A is +1; if player B reaches space 1 first, then the value of the game to A is −1.

a. Draw the complete game tree, using the following conventions:
    i. Write each state as (sA, sB), where sA and sB denote the token locations.
    ii. Put each terminal state in a square box and write its game value in a circle.
    iii. Put loop states (states that already appear on the path to the root) in double square boxes. Since their value is unclear, annotate each with a "?" in a circle.
b. Now mark each node with its backed-up minimax value (also in a circle). Explain how you handled the "?" values and why.
c. Explain why the standard minimax algorithm would fail on this game tree and briefly sketch how you might fix it, drawing on your answer to (b). Does your modified algorithm give optimal decisions for all games with loops?
d. This 4-square game can be generalized to n squares for any n > 2. Prove that A wins if n is even and loses if n is odd.

2. Consider the family of generalized tic-tac-toe games, defined as follows. Each particular game is specified by a set S of squares and a collection W of winning positions. Each winning position is a subset of S. For example, in standard tic-tac-toe, S is a set of 9 squares and W is a collection of 8 subsets ofW: the three rows, the three columns, and the two diagonals. In other respects, the game is identical to standard tic-tac-toe. Starting from an empty board, players alternate placing their marks on an empty square. A player who marks every square in a winning position wins the game. It is a tie if all squares are marked and neither player has won.
    a. Let N = |S|, the number of squares. Give an upper bound on the number of nodes in the complete game tree for generalized tic-tac-toe as a function of N.
    b. Give a lower bound on the size of the game tree for the worst case, where W = {}.
    c. Propose a plausible evaluation function that can be used for any instance of generalized tic-tac-toe. The function may depend on S and W.
    d. Assume that it is possible to generate a new board and check whether it is a winning position in 100N machine instructions and assume a 2 gigahertz processor. Ignore memory limitations. Using your estimate in (a), roughly how large a game tree can be completely solved by alpha–beta in a second of CPU time? a minute? an hour?

7.0 References/Further Readings

1) Russell, S., & Norvig, P. (2005). AI a modern approach. Learning, 2(3), 4.
2) Rich E., & Knight K. (1991) Artificial Intelligence, McGraw-Hill, New York, NY
3) Adversarial Search. https://www.javatpoint.com/ai-adversarial-search retrieved 7/8/2019
4) Mini-Max Algorithm in Artificial Intelligence. https://www.javatpoint.com/mini-max-algorithm-in-ai retrieved 7/8/2019
5) Alpha-Beta Pruning. https://www.javatpoint.com/ai-alpha-beta-pruning retrieved 7/8/2019

Unit Four: knowledge representation

CONTENT

1.0 Introduction

Humans are best at understanding, reasoning, and interpreting knowledge. Human knows things, which is knowledge and as per their knowledge they perform various actions in the real world. But how machines do all these things comes under knowledge representation and reasoning.

2.0 Objectives

At the end of this unit, you should be able to do the following:

- Define knowledge representation
- Outline what is supposed to be represented
- Define knowledge as it applies to AI
- Enumerate and define the types of knowledge
- Explain the relationship between knowledge and intelligence
- Know the building blocks of the AI knowledge cycle and how they relate with each other

- Outline and explain the different approaches used for knowledge representation
- Discuss the requirements for a knowledge representation system
- Understand the architecture of knowledge-based agents
- outline the techniques of knowledge representation along with their advantages and disadvantages.

3.0 Main Content

3.1 What is knowledge representation?

Knowledge representation and reasoning (KR, KRR) is the part of Artificial intelligence which concerned with AI agents thinking and how thinking contributes to intelligent behavior of agents. It is responsible for representing information about the real world so that a computer can understand and can utilize this knowledge to solve the complex real-world problems such as diagnosis a medical condition or communicating with humans in natural language. It is also a way which describes how we can represent knowledge in artificial intelligence. Knowledge representation is not just storing data into some database, but it also enables an intelligent machine to learn from that knowledge and experiences so that it can behave intelligently like a human.

3.2 What to Represent

Following are the kind of knowledge which needs to be represented in AI systems:

a. **Object:** All the facts about objects in our world domain. E.g., Guitars contains strings, trumpets are brass instruments.
b. **Events:** Events are the actions which occur in our world.
c. **Performance:** It describe behavior which involves knowledge about how to do things.
d. **Meta-knowledge:** It is knowledge about what we know.
e. **Facts:** Facts are the truths about the real world and what we represent.
f. **Knowledge-Base:** The central component of the knowledge-based agents is the knowledge base. It is represented as KB. The Knowledgebase is a group of the Sentences (Here, sentences are used as a technical term and not identical with the English language).

3.3 Knowledge

Knowledge is awareness or familiarity gained by experiences of facts, data, and situations. Following are the types of knowledge in artificial intelligence:

3.3.1 Types of knowledge

The following are the various types of knowledge:

Figure 3.1: Types of Knowledge

a. Declarative Knowledge:

Declarative knowledge is to know about something. It includes concepts, facts, and objects. It is also called descriptive knowledge and expressed in declarative sentences. It is simpler than procedural language.

b. Procedural Knowledge

It is also known as imperative knowledge. Procedural knowledge is a type of knowledge which is responsible for knowing how to do something. It can be directly applied to any task. It includes rules, strategies, procedures, agendas, etc. Procedural knowledge depends on the task on which it can be applied.

c. Meta-knowledge:

Knowledge about the other types of knowledge is called Meta-knowledge.

d. Heuristic knowledge:

Heuristic knowledge is representing knowledge of some experts in a filed or subject. Heuristic knowledge is rules of thumb based on previous experiences, awareness of approaches, and which are good to work but not guaranteed.

e. Structural knowledge:

Structural knowledge is basic knowledge to problem-solving. It describes relationships between various concepts such as kind of, part of, and grouping of something. It describes the relationship that exists between concepts or objects.

3.3.2 The relation between knowledge and intelligence

Knowledge of real-worlds plays a vital role in intelligence and same for creating artificial intelligence. Knowledge plays an important role in demonstrating intelligent behavior in AI agents. An agent is only able to accurately act on some input when he has some knowledge or experience about that input.

Let's suppose if you met some person who is speaking in a language which you don't know, then how you will able to act on that. The same thing applies to the intelligent behavior of the agents. As we can see in below diagram, there is one decision maker which act by sensing the environment and using knowledge. But if the knowledge part will not present then, it cannot display intelligent behavior.

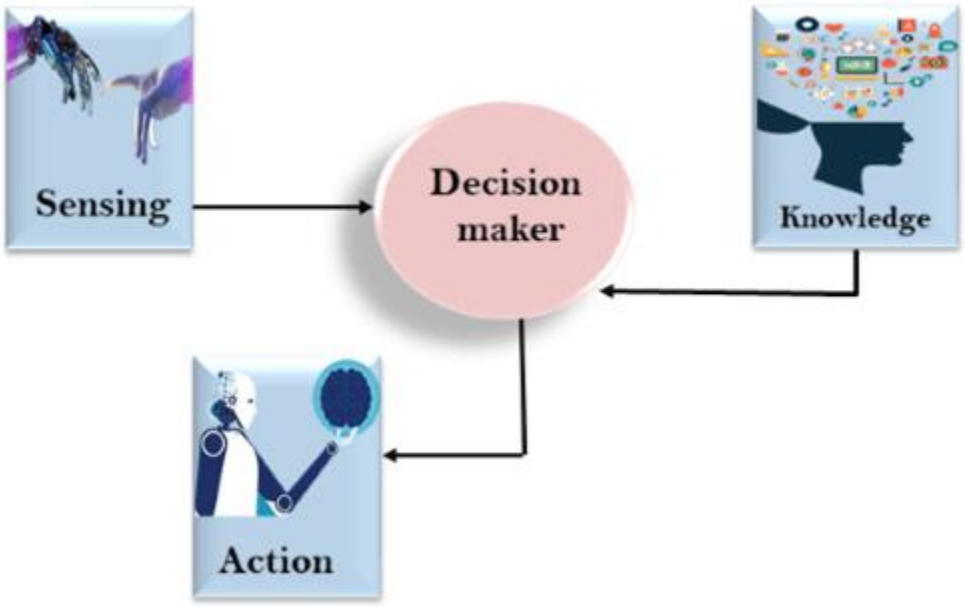


Figure 3.2: relation between knowledge and intelligence

3.3.3 AI knowledge cycle

An Artificial intelligence system has the following components for displaying intelligent behavior:

- Perception
- Learning
- Knowledge Representation and Reasoning
- Planning
- Execution

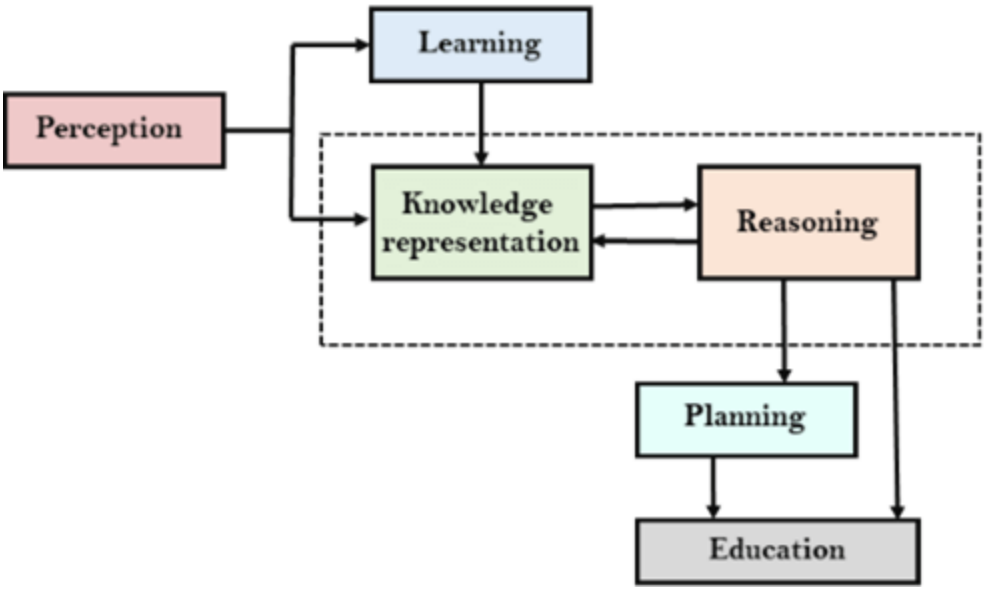


Figure 3.3: AI Knowledge Cycle

The above diagram is showing how an AI system can interact with the real world and what components help it to show intelligence. AI system has Perception component by which it retrieves information from its environment. It can be visual, audio or another form of sensory input. The learning component is responsible for learning from data captured by Perception comportment. In the complete cycle, the main components are knowledge representation and Reasoning. These two components are involved in showing the intelligence in machine-like humans. These two components are independent with each other but also coupled together. The planning and execution depend on analysis of Knowledge representation and reasoning.

3.3.4 Approaches to knowledge representation

There are mainly four approaches to knowledge representation, which are given below:

a. Simple relational knowledge:

It is the simplest way of storing facts which uses the relational method, and each fact about a set of the object is set out systematically in columns. This approach of knowledge representation is famous in database systems where the relationship between different entities is represented. This approach has little opportunity for inference.

Example: The following is the simple relational knowledge representation.

Player	Weight	Age
Player1	65	23
Player2	58	18
Player3	75	24

b. Inheritable knowledge:

In the inheritable knowledge approach, all data must be stored into a hierarchy of classes. All classes should be arranged in a generalized form or a hierarchal manner. In this approach, we apply inheritance property. Elements inherit values from other members of a class.

This approach contains inheritable knowledge which shows a relation between instance and class, and it is called instance relation. Every individual frame can represent the collection of attributes and its value. In this approach, objects and values are represented in Boxed nodes. We use Arrows which point from objects to their values.

Example:

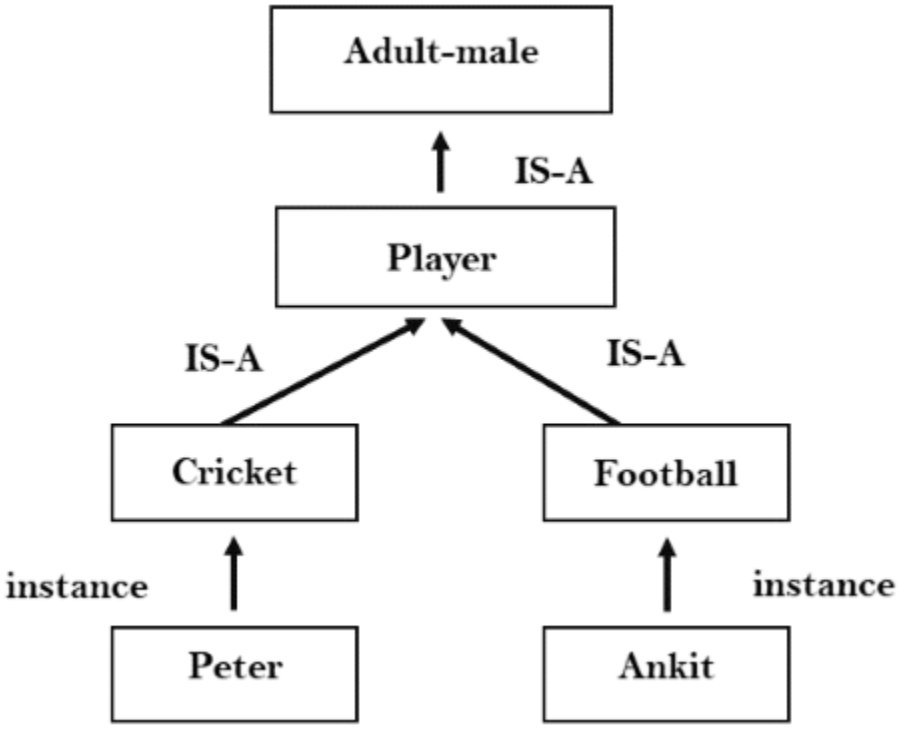


Figure 3.4: Inheritable Knowledge

c. Inferential knowledge:

Inferential knowledge approach represents knowledge in the form of formal logics. This approach can be used to derive more facts. It guaranteed correctness.

Example: Let's suppose there are two statements:

1. Marcus is a man
2. All men are mortal

Then it can represent as;

man(Marcus)

∀x = man (x) ----------> mortal (x)s

d. Procedural knowledge:

Procedural knowledge approach uses small programs and codes which describes how to do specific things, and how to proceed. In this approach, one important rule is used which is If-Then rule. In this knowledge, we can use various coding languages such as LISP language and Prolog language. We can easily represent heuristic or domain-specific knowledge using this approach. But it is not necessary that we can represent all cases in this approach.

3.4 Requirements for knowledge Representation system

A good knowledge representation system must possess the following properties.

a. **Representational Accuracy**:
   KR system should have the ability to represent all kind of required knowledge.
b. **Inferential Adequacy**:
   KR system should have ability to manipulate the representational structures to produce new knowledge corresponding to existing structure.
c. **Inferential Efficiency**:
   The ability to direct the inferential knowledge mechanism into the most productive directions by storing appropriate guides.
d. **Acquisitional efficiency:** The ability to acquire the new knowledge easily using automatic methods.

3.5 Knowledge-Based Agent in Artificial intelligence

An intelligent agent needs knowledge about the real world for taking decisions and reasoning to act efficiently. Knowledge-based agents are those agents who have the capability of maintaining an internal state of knowledge, reason over that knowledge, update their knowledge after observations and take actions. These agents can represent the world with some formal representation and act intelligently.

Knowledge-based agents are composed of two main parts:

a. Knowledge-base and
b. Inference system.

A knowledge-based agent must able to do the following:

a. An agent should be able to represent states, actions, etc.
b. An agent Should be able to incorporate new percepts

c. An agent can update the internal representation of the world
d. An agent can deduce the internal representation of the world
e. An agent can deduce appropriate actions.

3.6 The architecture of knowledge-based agent

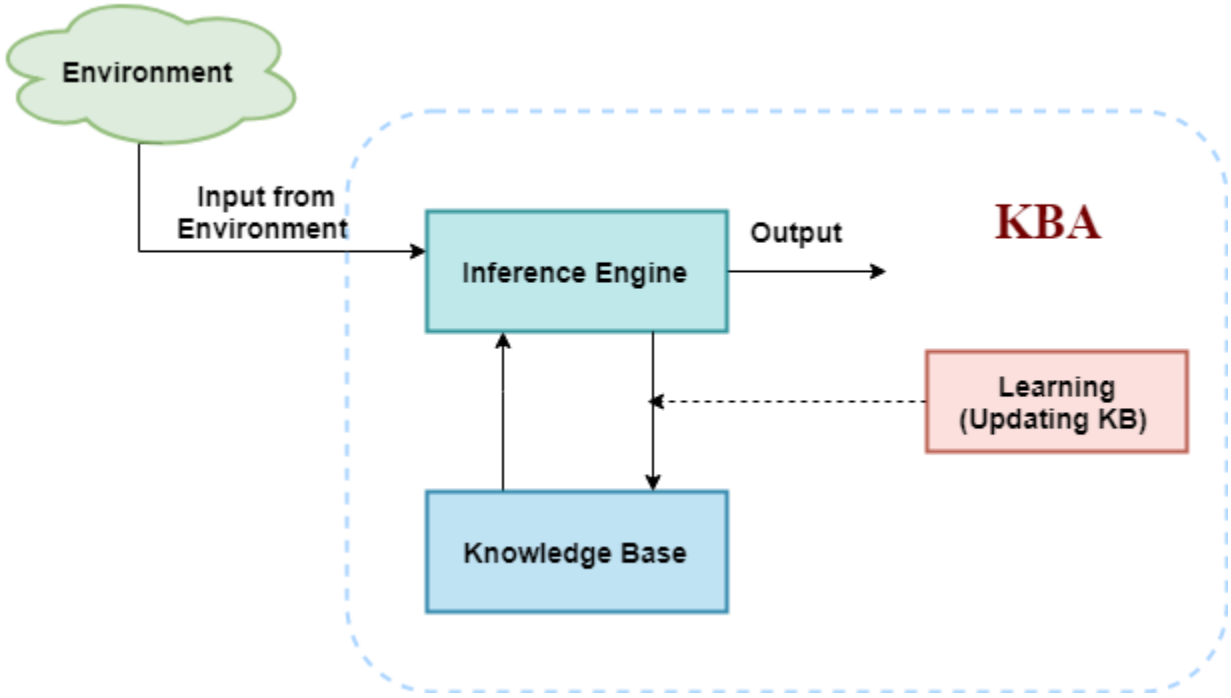


Figure 3.5: Knowledge Based Agent Architecture

The above diagram is representing a generalized architecture for a knowledge-based agent. The knowledge-based agent (KBA) take input from the environment by perceiving the environment. The input is taken by the inference engine of the agent and which also communicate with KB to decide as per the knowledge store in KB. The learning element of KBA regularly updates the KB by learning new knowledge.

i. **Knowledge base:**

Knowledge-base is a central component of a knowledge-based agent, it is also known as KB. It is a collection of sentences (here 'sentence' is a technical term and it is not identical to sentence in English). These sentences are expressed in a language which is called a knowledge representation language. The Knowledge-base of KBA stores fact about the world.

Why use a knowledge base?

Knowledge-base is required for updating knowledge for an agent to learn with experiences and take action as per the knowledge.

ii. Inference system

Inference means deriving new sentences from old. Inference system allows us to add a new sentence to the knowledge base. A sentence is a proposition about the world. Inference system applies logical rules to the KB to deduce new information.

Inference system generates new facts so that an agent can update the KB. An inference system works mainly in two rules which are given as:

- Forward chaining
- Backward chaining

iii. Operations Performed by KBA

Following are three operations which are performed by KBA in order to show the intelligent behavior:

1. TELL: This operation tells the knowledge base what it perceives from the environment.
2. ASK: This operation asks the knowledge base what action it should perform.
3. Perform: It performs the selected action.

A generic knowledge-based agent:

Following is the structure outline of a generic knowledge-based agents program:

```
function KB-AGENT(percept):
persistent: KB, a knowledge base
      t, a counter, initially 0, indicating time
TELL(KB, MAKE-PERCEPT-SENTENCE(percept, t))
Action = ASK(KB, MAKE-ACTION-QUERY(t))
TELL(KB, MAKE-ACTION-SENTENCE(action, t))
 t = t + 1
 return action
```

The knowledge-based agent takes percept as input and returns an action as output. The agent maintains the knowledge base, KB, and it initially has some background knowledge of the real world. It also has a counter to indicate the time for the whole process, and this counter is initialized with zero.

Each time when the function is called, it performs its three operations:

1. Firstly it TELLs the KB what it perceives.
2. Secondly, it asks KB what action it should take
3. Third agent program TELLS the KB that which action was chosen.

The MAKE-PERCEPT-SENTENCE generates a sentence as setting that the agent perceived the given percept at the given time.

The MAKE-ACTION-QUERY generates a sentence to ask which action should be done at the current time.

MAKE-ACTION-SENTENCE generates a sentence which asserts that the chosen action was executed.

3.10 Various levels of knowledge-based agent

A knowledge-based agent can be viewed at different levels which are given below:

1. Knowledge level

Knowledge level is the first level of knowledge-based agent, and in this level, we need to specify what the agent knows, and what the agent goals are. With these specifications, we can fix its behavior. For example, suppose an automated taxi agent needs to go from a station A to station B, and he knows the way from A to B, so this comes at the knowledge level.

2. Logical level:

At this level, we understand that how the knowledge representation of knowledge is stored. At this level, sentences are encoded into different logics. At the logical level, an encoding of knowledge into logical sentences occurs. At the logical level we can expect to the automated taxi agent to reach to the destination B.

3. Implementation level:

This is the physical representation of logic and knowledge. At the implementation level agent perform actions as per logical and knowledge level. At this level, an automated taxi agent actually implement his knowledge and logic so that he can reach to the destination.

Approaches to designing a knowledge-based agent:

There are mainly two approaches to build a knowledge-based agent:

1. Declarative approach: We can create a knowledge-based agent by initializing with an empty knowledge base and telling the agent all the sentences with which we want to start with. This approach is called Declarative approach.
2. Procedural approach: In the procedural approach, we directly encode desired behavior as a program code. Which means we just need to write a program that already encodes the desired behavior or agent.

However, in the real world, a successful agent can be built by combining both declarative and procedural approaches, and declarative knowledge can often be compiled into more efficient procedural code.

3.11 Techniques of knowledge representation

There are mainly four ways of knowledge representation which are given as follows:

a. Logical Representation
b. Semantic Network Representation
c. Frame Representation
d. Production Rules

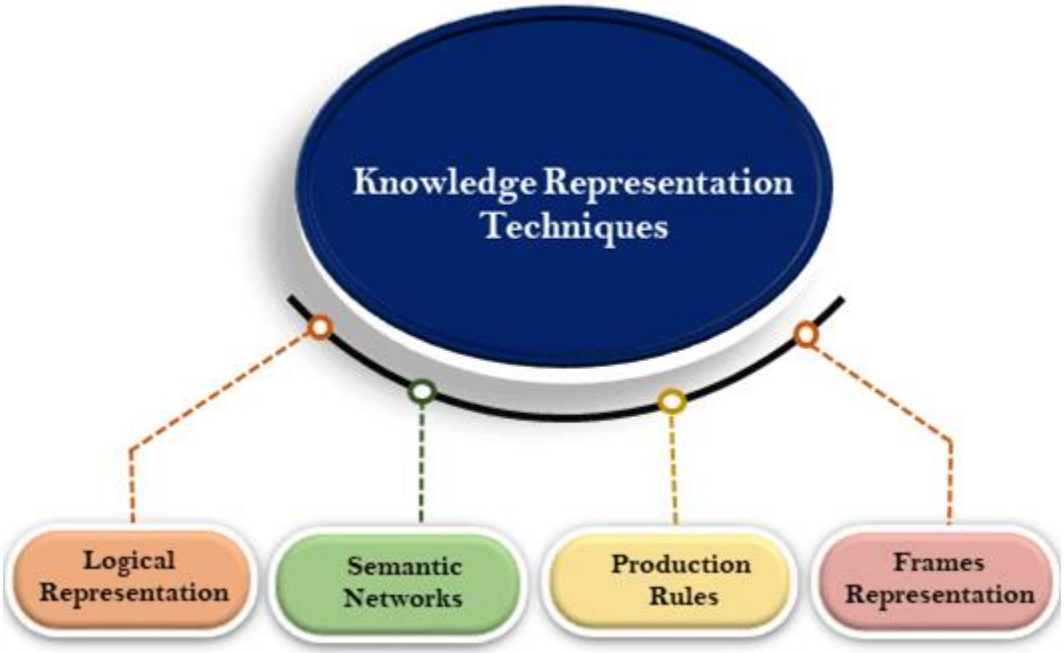


Figure 3.6: Techniques of Knowledge Representation

3.11.1 Logical Representation

Logical representation is a language with some concrete rules which deals with propositions and has no ambiguity in representation. Logical representation means drawing a conclusion based on various conditions. This representation lays down some important communication rules. It consists of precisely defined syntax and semantics which supports the sound inference. Each sentence can be translated into logics using syntax and semantics.

Syntax:

Syntaxes are the rules which:

a. Decide how we can construct legal sentences in the logic.
b. determines which symbol we can use in knowledge representation.
c. decides How to write those symbols.

Semantics:

Semantics are the rules by which we can interpret the sentence in the logic. Semantic also involves assigning a meaning to each sentence.

Logical representation can be categorized into mainly two logics:

a. Propositional Logics
b. Predicate logics

Advantages of logical representation:

a. Logical representation enables us to do logical reasoning.
b. Logical representation is the basis for the programming languages.

Disadvantages of logical Representation:

a. Logical representations have some restrictions and are challenging to work with.
b. Logical representation technique may not be very natural, and inference may not be so efficient.

Note: Do not be confused with logical representation and logical reasoning as logical representation is a representation language and reasoning is a process of thinking logically.

3.11.2 Semantic Network Representation

Semantic networks are alternative of predicate logic for knowledge representation. In Semantic networks, we can represent our knowledge in the form of graphical networks. This network consists of nodes representing objects and arcs which describe the relationship between those objects. Semantic networks can categorize the object in different forms and can also link those objects. Semantic networks are easy to understand and can be easily extended.

This representation consists of mainly two types of relations:

a. IS-A relation (Inheritance)
b. Kind-of-relation

Example: Following are some statements which we need to represent in the form of nodes and arcs.

Statements:

a. Jerry is a cat.
b. Jerry is a mammal
c. Jerry is owned by Priya.

d. Jerry is brown colored.
e. All Mammals are animal.

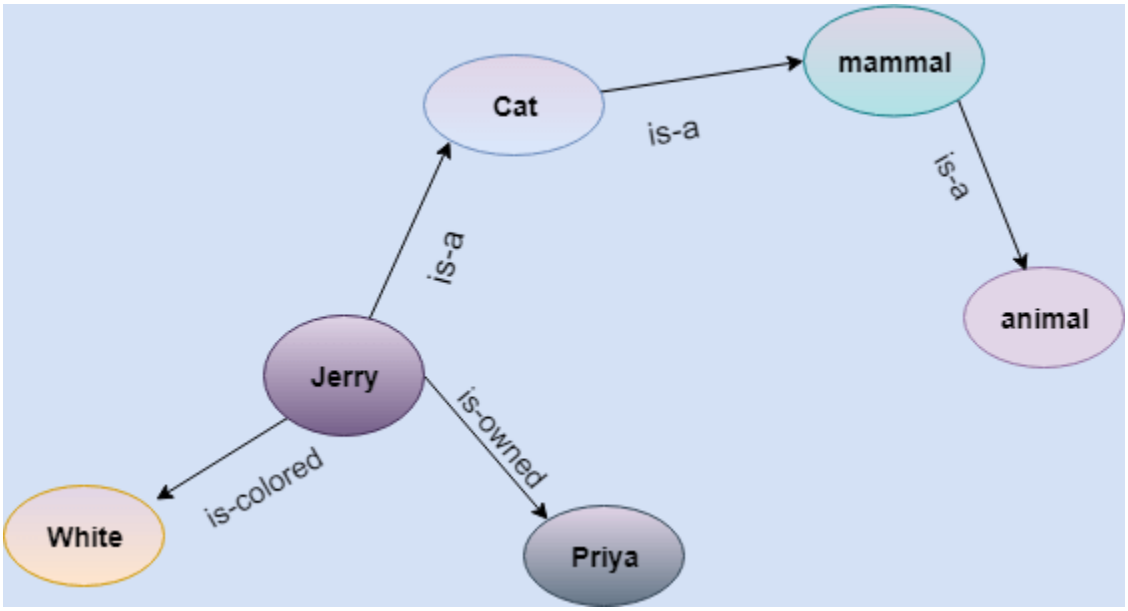


Figure 3.7: Semantic Network Representation

In the above diagram, we have represented the different type of knowledge in the form of nodes and arcs. Each object is connected with another object by some relation.

Drawbacks in Semantic representation:

1. Semantic networks take more computational time at runtime as we need to traverse the complete network tree to answer some questions. It might be possible in the worst-case scenario that after traversing the entire tree, we find that the solution does not exist in this network.
2. Semantic networks try to model human-like memory (Which has 1015 neurons and links) to store the information, but in practice, it is not possible to build such a vast semantic network.
3. These types of representations are inadequate as they do not have any equivalent quantifier, e.g., for all, for some, none, etc.
4. Semantic networks do not have any standard definition for the link names.
5. These networks are not intelligent and depend on the creator of the system.

Advantages of Semantic network:

1. Semantic networks are a natural representation of knowledge.
2. Semantic networks convey meaning in a transparent manner.
3. These networks are simple and easily understandable.

3.11.3 Frame Representation

A frame is a record like structure which consists of a collection of attributes and its values to describe an entity in the world. Frames are the AI data structure which divides knowledge into substructures by representing stereotypes situations. It consists of a collection of slots and slot values. These slots may be of any type and sizes. Slots have names and values which are called facets.

Facets: The various aspects of a slot is known as Facets. Facets are features of frames which enable us to put constraints on the frames. Example: IF-NEEDED facts are called when data of any particular slot is needed. A frame may consist of any number of slots, and a slot may include any number of facets and facets may have any number of values. A frame is also known as slot-filter knowledge representation in artificial intelligence.

Frames are derived from semantic networks and later evolved into our modern-day classes and objects. A single frame is not much useful. Frames system consist of a collection of frames which are connected. In the frame, knowledge about an object or event can be stored together in the knowledge base. The frame is a type of technology which is widely used in various applications including Natural language processing and machine visions.

Example: 1

Let's take an example of a frame for a book

Slots	Filters
Title	Artificial Intelligence
Genre	Computer Science
Author	Peter Norvig
Edition	Third Edition
Year	1996
Page	1152

Example 2:

Let's suppose we are taking an entity, Peter. Peter is an engineer as a profession, and his age is 25, he lives in city London, and the country is England. So following is the frame representation for this:

Slots	Filter
Name	Peter
Profession	Doctor
Age	25
Marital status	Single
Weight	78

Advantages of frame representation:

1. The frame knowledge representation makes the programming easier by grouping the related data.
2. The frame representation is comparably flexible and used by many applications in AI.
3. It is very easy to add slots for new attribute and relations.
4. It is easy to include default data and to search for missing values.
5. Frame representation is easy to understand and visualize.

Disadvantages of frame representation:

1. In frame system inference mechanism is not be easily processed.
2. Inference mechanism cannot be smoothly proceeded by frame representation.
3. Frame representation has a much-generalized approach.

3.11.4 Production Rules

Production rules system consist of (condition, action) pairs which mean, "If condition then action". It has mainly three parts:

1. The set of production rules
2. Working Memory
3. The recognize-act-cycle

In production rules agent checks for the condition and if the condition exists then production rule fires and corresponding action is carried out. The condition part of the rule determines which rule may be applied to a problem. And the action part carries out the associated problem-solving steps. This complete process is called a recognize-act cycle.

The working memory contains the description of the current state of problems-solving and rule can write knowledge to the working memory. This knowledge match and may fire other rules.

If there is a new situation (state) generates, then multiple production rules will be fired together, this is called conflict set. In this situation, the agent needs to select a rule from these sets, and it is called a conflict resolution.

Example:

- IF (at bus stop AND bus arrives) THEN action (get into the bus)
- IF (on the bus AND paid AND empty seat) THEN action (sit down).
- IF (on bus AND unpaid) THEN action (pay charges).
- IF (bus arrives at destination) THEN action (get down from the bus).

Advantages of Production rule:

1. The production rules are expressed in natural language.
2. The production rules are highly modular, so we can easily remove, add or modify an individual rule.

Disadvantages of Production rule:

1. Production rule system does not exhibit any learning capabilities, as it does not store the result of the problem for the future uses.
2. During the execution of the program, many rules may be active hence rule-based production systems are inefficient.

4.0 Conclusion

This unit takes you from the definition of knowledge to the techniques of knowledge representation. The AI knowledge cycle was introduced and the relationship of each block was discussed. You should be able to easily focus on your area of interest and understand the how each aspect relates with each other.

5.0 Summary

We hope you enjoyed this unit. This unit knowledge representation in AI. Now, let us attempt the questions below.

6.0 Tutor Marked Assignment

1. What is an inference system? Mention the two rules under which and inference system operates
2. Discuss with examples the relationship between the inference engine and the knowledge base in a KBA.

7.0 References/Further Readings

1) Russell, S., & Norvig, P. (2005). AI a modern approach. Learning, 2(3), 4.
2) Rich E., & Knight K. (1991) Artificial Intelligence, McGraw-Hill, New York, NY
3) Shi, Z. (2011). Advanced artificial intelligence (Vol. 1). World Scientific.
4) What is knowledge representation? https://www.javatpoint.com/knowledge-representation-in-ai retrieved 8/7/2019
5) Knowledge-Based Agent in Artificial intelligence https://www.javatpoint.com/knowledge-based-agent-in-ai retrieved 8/7/2019
6) Techniques of knowledge representation https://www.javatpoint.com/ai-techniques-of-knowledge-representation retrieved 8/7/2019

MODULE 3: APPLICATION OF AI TECHNIQUES

Unit One: Natural Language
Unit Two: Scene Analysis,
Unit Three: Expert Systems,
Unit Four: KBCS Robot Planning

Unit One: Natural Language,

CONTENT

1.0 Introduction

We use the English language to communicate between an intelligent system and Natural Language Processing (NLP) Processing of Natural Language plays an important role in various systems.

For Example:

A robot, it is used to perform as per your instructions. The input and output of an N.L.P system can be −

- Speech
- Written Text

Components of NLP

Basically, there are two components of Natural Language Processing systems:

a. Natural Language Understanding (NLU)

In this, we have to understand the basic tasks −

- Basically, the mapping to given input in natural language into useful representations.
- Analyzing different aspects of the language.

b. Natural Language Generation (NLG)

We have to produce meaningful phrases and sentences. That is in the form of natural language from internal representation.

As this process involves:

- *Text planning:* In this process, we have to retrieve the relevant content from a knowledge base.
- *Sentence planning:* We have to choose the required words for setting a tone of the sentence.
- *Text Realization*: Basically, it's process of mapping sentence plan into sentence structure. Although, the NLU is harder than NLG.

Difficulties in NLU

a. Lexical ambiguity: It's predefined at a very primitive level such as word-level.

b. Syntax Level ambiguity: In this, we can define a sentence in a parsed way in different ways.

c. Referential ambiguity: Referential ambiguity says that we have to refer something using pronouns only.

5. Natural Language Processing – Terminologies

a. **Phonology:** It's study of organizing sound.

b. **Morphology:** Basically, it's study of the construction of words from primitive meaningful units.

c. **Morpheme:** As we can say that it's primitive unit of meaning in a language:

- **Syntax:** In this, we have to arrange words to make a sentence. Also, involves determining the structural role of words. That is in the sentence and in phrases.
- **Semantics:** It defines the meaning of words. Moreover, how to combine words into meaningful phrases and sentences
- **Pragmatics:** It deals with use and understanding sentences in different situations. Also, defines how the interpretation of the sentence is affected.
- **World Knowledge:** It includes the general knowledge about the world.

There are over a trillion pages of information on the Web, almost all of it in natural language. An agent that wants to do knowledge acquisition needs to understand (at least partially) the ambiguous, messy languages that humans use. We examine the problem from the point of view of specific information-seeking tasks: text classification, information retrieval, and information extraction. One common factor in addressing these tasks is the use of language models: models that predict the probability distribution of language expressions.

2.0 Objectives

At the end of this unit, you should be able to:

- Outline the components of NLP
- Define the terminologies used in NLP
- Outline discuss the concepts and implementation of the four language models
- Implement text classification by data compression
- Outline the IR scoring functions
- Be able to evaluate IR systems
- Discuss the working principle of PageRank algorithm
- Discuss the working principle behind the HITS algorithm
- Outline some information extraction methods
- Discuss and implement information extraction using any of the methods

3.0 Main Content

3.1 LANGUAGE MODELS

Formal languages, such as the programming languages Java or Python, have precisely defined language models. A language can be defined as a set of strings; “print(2 + 2)” is a legal program in the language Python, whereas “2)+(2 print” is not. Since there are an infinite number of legal programs, they cannot be enumerated; instead they are specified by a set of rules called a grammar. Formal languages also have rules that define the meaning or semantics of a program; for example, the rules say that the “meaning” of “2 + 2” is 4, and the meaning of “1/0” is that an error is signaled.

Natural languages, such as English or Spanish, cannot be characterized as a definitive set of sentences. Everyone agrees that "Not to be invited is sad" is a sentence of English, but people disagree on the grammaticality of "To be not invited is sad." Therefore, it is more fruitful to define a natural language model as a probability distribution over sentences rather than a definitive set. That is, rather than asking if a string of words is or is not a member of the set defining the language, we instead ask for P(S =words)—what is the probability that a random sentence would be words.

Natural languages are also ambiguous. AMBIGUITY "He saw her duck" can mean either that he saw a waterfowl belonging to her, or that he saw her move to evade something. Thus, again, we cannot speak of a single meaning for a sentence, but rather of a probability distribution over possible meanings.

Finally, natural languages are difficult to deal with because they are very large, and constantly changing. Thus, our language models are, at best, an approximation. We start with the simplest possible approximations and move up from there.

3.1.1 N-gram character models

Ultimately, a written text is composed of characters—letters, digits, punctuation, and spaces in English (and more exotic characters in some other languages). Thus, one of the simplest language models is a probability distribution over sequences of characters. We write $P(c_{1:N})$ for the probability of a sequence of N characters, c_1 through c_N. In one Web collection, P("the")=0.027 and P("zgq")=0.000000002. A sequence of written symbols of length n is called an n-gram (from the Greek root for writing or letters), with special case "unigram" for 1-gram, "bigram" for 2-gram, and "trigram" for 3-gram. A model of the probability distribution of n-letter sequences is thus called an n-gram model. (But be careful: we can have n-gram models over sequences of words, syllables, or other units; not just over characters.)

An n-gram model is defined as a Markov chain of order n − 1. Recall from page 568 that in a Markov chain the probability of character ci depends only on the immediately preceding characters, not on any other characters. So in a trigram model (Markov chain of order 2) we have

$$P(c_i \mid c_{1:i-1}) = P(c_i \mid c_{i-2:i-1}) . \quad (1)$$

We can define the probability of a sequence of characters P(c1:N) under the trigram model by first factoring with the chain rule and then using the Markov assumption:

$$P(c_{1:N}) = \prod_{i=1}^{N} (c_i|c_{1:i-1}) = \prod_{i=1}^{N} (c_i|c_{1-2:i-1}) \quad (2)$$

For a trigram character model in a language with 100 characters, $P(c_i|c_{1-2:i-1})$ has a million entries, and can be accurately estimated by counting character sequences in a body of text of CORPUS 10 million characters or more. We call a body of text a corpus (plural corpora), from the Latin word for body.

What can we do with n-gram character models? One task for which they are well suited is language identification: given a text, determine what natural language it is written in. This is a relatively easy task; even with short texts such as "Hello, world" or "Wie geht es dir," it is easy to identify the first as English and the second as German. Computer systems identify languages with greater than 99% accuracy; occasionally, closely related languages, such as Swedish and Norwegian, are confused.

One approach to language identification is to first build a trigram character model of each candidate language, $P(c_i|c_{1-2:i-1}), \ell$ where the variable ℓ ranges over languages. For each ℓ the model is built by

counting trigrams in a corpus of that language. (About 100,000 characters of each language are needed.) That gives us a model of P(Text | Language), but we want to select the most probable language given the text, so we apply Bayes' rule followed by the Markov assumption to get the most probable language:

$$\ell^* = \underset{\ell}{\operatorname{argmax}}\ P(\ell|c_{1:N}) \quad (3)$$

$$= \underset{\ell}{\operatorname{argmax}}\ P(\ell)P(c_{1:N}|\ell) \quad (4)$$

$$= \underset{\ell}{\operatorname{argmax}}\ P(\ell)\prod_{i=1}^{N} P(c_i|c_{1-2:i-1}), \ell \quad (5)$$

The trigram model can be learned from a corpus, but what about the prior probability $P(\ell)$? We may have some estimate of these values; for example, if we are selecting a random Web page we know that English is the most likely language and that the probability of Macedonian will be less than 1%. The exact number we select for these priors is not critical because the trigram model usually selects one language that is several orders of magnitude more probable than any other.

Other tasks for character models include spelling correction, genre classification, and named-entity recognition. Genre classification means deciding if a text is a news story, a legal document, a scientific article, etc. While many features help make this classification, counts of punctuation and other character n-gram features go a long way (Kessler et al., 1997). Named-entity recognition is the task of finding names of things in a document and deciding what class they belong to. For example, in the text "Mr. Sopersteen was prescribed aciphex," we should recognize that "Mr. Sopersteen" is the name of a person and "aciphex" is the name of a drug. Character-level models are good for this task because they can associate the character sequence "ex␣" ("ex" followed by a space) with a drug name and "steen␣" with a person name, and thereby identify words that they have never seen before.

3.1.2 Smoothing n-gram models

The major complication of n-gram models is that the training corpus provides only an estimate of the true probability distribution. For common character sequences such as "␣th" any English corpus will give a good estimate: about 1.5% of all trigrams. On the other hand, "␣ht" is very uncommon—no dictionary words start with ht. It is likely that the sequence would have a count of zero in a training corpus of standard English. Does that mean we should assign P("␣th")=0? If we did, then the text "The program issues an http request" would have an English probability of zero, which seems wrong. We have a problem in generalization: we want our language models to generalize well to texts they haven't seen yet. Just because we have never seen "␣http" before does not mean that our model should claim that it is impossible.

Thus, we will adjust our language model so that sequences that have a count of zero in the training corpus will be assigned a small nonzero probability (and the other counts will be adjusted downward slightly so that the probability still sums to 1). The process od adjusting the probability of low-frequency counts is called smoothing.

The simplest type of smoothing was suggested by Pierre-Simon Laplace in the 18th century: he said that, in the lack of further information, if a random Boolean variable X has been false in all n observations so far then the estimate for P(X =true) should be 1/(n+2). That is, he assumes that with two more trials, one

might be true and one false. Laplace smoothing (also called add-one smoothing) is a step in the right direction, but performs relatively poorly.

A better approach is a backoff model, in which we start by estimating n-gram counts, but for any particular sequence that has a low (or zero) count, we back off to (n −1)-grams. Linear interpolation smoothing is a backoff model that combines trigram, bigram, and unigram models by linear interpolation. It defines the probability estimate as

$$P(c_i|c_{i-2:i-1}) = \lambda_3 P(c_i|c_{i-2:i-1}) + \lambda_2 P(c_i|c_{i-1}) + \lambda_1 P(c_i) , \quad (6)$$

where $\lambda_3 + \lambda_2 + \lambda_1 = 1$. The parameter values λ_i can be fixed, or they can be trained with an expectation–maximization algorithm. It is also possible to have the values of λ_i depend on the counts: if we have a high count of trigrams, then we weigh them relatively more; if only a low count, then we put more weight on the bigram and unigram models. One camp of researchers has developed ever more sophisticated smoothing models, while the other camp suggests gathering a larger corpus so that even simple smoothing models work well. Both are getting at the same goal: reducing the variance in the language model.

One complication: note that the expression $P(c_i \mid c_{i-2:i-1})$ asks for $P(c_1 \mid c_{-1:0})$ when i = 1, but there are no characters before c_1. We can introduce artificial characters, for example, defining c0 to be a space character or a special "begin text" character. Or we can fall back on lower-order Markov models, in effect defining c-1:0 to be the empty sequence and thus $P(c_1 \mid c_{-1:0}) = P(c_1)$.

3.1.3 Model evaluation

With so many possible n-gram models—unigram, bigram, trigram, interpolated smoothing with different values of λ, etc.—how do we know what model to choose? We can evaluate a model with cross-validation. Split the corpus into a training corpus and a validation corpus. Determine the parameters of the model from the training data. Then evaluate the model on the validation corpus.

The evaluation can be a task-specific metric, such as measuring accuracy on language identification. Alternatively we can have a task-independent model of language quality: calculate the probability assigned to the validation corpus by the model; the higher the probability the better. This metric is inconvenient because the probability of a large corpus will be a very small number, and floating-point underflow becomes an issue. A different way of describing the probability of a sequence is with a measure called perplexity, defined as

$$\text{Perplexity}(c_{1:N}) = P(c_{1:N})^{-1/N} . \quad (7)$$

Perplexity can be thought of as the reciprocal of probability, normalized by sequence length. It can also be thought of as the weighted average branching factor of a model. Suppose there are 100 characters in our language, and our model says they are all equally likely. Then for a sequence of any length, the perplexity will be 100. If some characters are more likely than others, and the model reflects that, then the model will have a perplexity less than 100.

3.1.4 N-gram word models

Now we turn to n-gram models over words rather than characters. All the same mechanism applies equally to word and character models. The main difference is that the vocabulary— the set of symbols

that make up the corpus and the model—is larger. There are only about 100 characters in most languages, and sometimes we build character models that are even more restrictive, for example by treating "A" and "a" as the same symbol or by treating all punctuation as the same symbol. But with word models we have at least tens of thousands of symbols, and sometimes millions. The wide range is because it is not clear what constitutes a word. In English a sequence of letters surrounded by spaces is a word, but in some languages, like Chinese, words are not separated by spaces, and even in English many decisions must be made to have a clear policy on word boundaries: how many words are in "ne'er-do-well"? Or in "(Tel:1-800-960-5660x123)"?

Word n-gram models need to deal with out of vocabulary words. With character models, we didn't have to worry about someone inventing a new letter of the alphabet. But with word models there is always the chance of a new word that was not seen in the training corpus, so we need to model that explicitly in our language model. This can be done by adding just one new word to the vocabulary: <UNK>, standing for the unknown word. We can estimate n-gram counts for <UNK> by this trick: go through the training corpus, and the first time any individual word appears it is previously unknown, so replace it with the symbol <UNK>. All subsequent appearances of the word remain unchanged. Then compute n-gram counts for the corpus as usual, treating <UNK> just like any other word. Then when an unknown word appears in a test set, we look up its probability under <UNK>. Sometimes multiple unknown-word symbols are used, for different classes. For example, any string of digits might be replaced with <NUM>, or any email address with <EMAIL>. To get a feeling for what word models can do, we built unigram, bigram, and trigram models over the words in this book and then randomly sampled sequences of words from the models.

The results are
Unigram: logical are as are confusion a may right tries agent goal the was . . .
Bigram: systems are very similar computational approach would be represented . . .
Trigram: planning and scheduling are integrated the success of naive bayes model is . . .

Even with this small sample, it should be clear that the unigram model is a poor approximation of either English or the content of an AI textbook, and that the bigram and trigram models are much better. The models agree with this assessment: the perplexity was 891 for the unigram model, 142 for the bigram model and 91 for the trigram model. With the basics of n-gram models—both character- and word-based—established, we can turn now to some language tasks.

3.2 TEXT CLASSIFICATION

We now consider in depth the task of text classification, also known as categorization: given a text of some kind, decide which of a predefined set of classes it belongs to. Language identification and genre classification are examples of text classification, as is sentiment analysis (classifying a movie or product review as positive or negative) and spam detection (classifying an email message as spam or not-spam). Since "not-spam" is awkward, researchers have coined the term ham for not-spam. We can treat spam detection as a problem in supervised learning. A training set is readily available: the positive (spam) examples are in my spam folder, the negative (ham) examples are in my inbox. Here is an excerpt:

Spam: Wholesale FashionWatches -57% today. Designer watches for cheap ...
Spam: You can buy ViagraFr$1.85 All Medications at unbeatable prices! ...
Spam: WE CAN TREAT ANYTHING YOU SUFFER FROM JUST TRUST US ...

Spam: Sta.rt earn*ing the salary yo,u d-eserve by o'btaining the prope,r crede'ntials!

Ham: The practical significance of hypertree width in identifying more ...
Ham: Abstract: We will motivate the problem of social identity clustering: ...
Ham: Good to see you my friend. Hey Peter, It was good to hear from you. ...
Ham: PDS implies convexity of the resulting optimization problem (Kernel Ridge ...

From this excerpt we can start to get an idea of what might be good features to include in the supervised learning model. Word n-grams such as "for cheap" and "You can buy" seem to be indicators of spam (although they would have a nonzero probability in ham as well).

Character-level features also seem important: spam is more likely to be all uppercase and to have punctuation embedded in words. Apparently the spammers thought that the word bigram "you deserve" would be too indicative of spam, and thus wrote "yo,u d-eserve" instead. A character model should detect this. We could either create a full character n-gram model of spam and ham, or we could handcraft features such as "number of punctuation marks embedded in words."

Note that we have two complementary ways of talking about classification. In the language-modeling approach, we define one n-gram language model for P(Message | spam) by training on the spam folder, and one model for P(Message | ham) by training on the inbox.

Then we can classify a new message with an application of Bayes' rule:

$$= \underset{c\epsilon\{spam,ham\}}{\operatorname{argmax}} P(c \ \ |message) = \underset{c\epsilon\{spam,ham\}}{\operatorname{argmax}} P(message|c)\, P(c) \qquad (8)$$

where P(c) is estimated just by counting the total number of spam and ham messages. This approach works well for spam detection, just as it did for language identification.

In the machine-learning approach we represent the message as a set of feature/value pairs and apply a classification algorithm h to the feature vector X. We can make the language-modeling and machine-learning approaches compatible by thinking of the n-grams as features. This is easiest to see with a unigram model. The features are the words in the vocabulary: "a," "aardvark," . . ., and the values are the number of times each word appears in the message. That makes the feature vector large and sparse. If there are 100,000 words in the language model, then the feature vector has length 100,000, but for a short email message almost all the features will have count zero. This unigram representation has been called the bag of words model. You can think of the model as putting the words of the training corpus in a bag and then selecting words one at a time. The notion of order of the words is lost; a unigram model gives the same probability to any permutation of a text. Higher-order n-gram models maintain some local notion of word order.

With bigrams and trigrams the number of features is squared or cubed, and we can add in other, non-n-gram features: the time the message was sent, whether a URL or an image is part of the message, an ID number for the sender of the message, the sender's number of previous spam and ham messages, and so on. The choice of features is the most important part of creating a good spam detector—more important than the choice of algorithm for processing the features. In part this is because there is a lot of training data, so if we can propose a feature, the data can accurately determine if it is good or not. It

is necessary to constantly update features, because spam detection is an adversarial task; the spammers modify their spam in response to the spam detector's changes.

It can be expensive to run algorithms on a very large feature vector, so often a process of feature selection is used to keep only the features that best discriminate between spam and ham. For example, the bigram "of the" is frequent in English, and may be equally frequent in spam and ham, so there is no sense in counting it. Often the top hundred or so features do a good job of discriminating between classes.

Once we have chosen a set of features, we can apply any of the supervised learning techniques we have seen; popular ones for text categorization include k-nearest-neighbors, support vector machines, decision trees, naive Bayes, and logistic regression. All of these have been applied to spam detection, usually with accuracy in the 98%–99% range. With a carefully designed feature set, accuracy can exceed 99.9%.

3.2.1 Classification by data compression

Another way to think about classification is as a problem in data compression. A lossless compression algorithm takes a sequence of symbols, detects repeated patterns in it, and writes a description of the sequence that is more compact than the original. For example, the text "0.142857142857142857" might be compressed to "0.[142857]*3." Compression algorithms work by building dictionaries of subsequences of the text, and then referring to entries in the dictionary. The example here had only one dictionary entry, "142857."

In effect, compression algorithms are creating a language model. The LZW algorithm in particular directly models a maximum-entropy probability distribution. To do classification by compression, we first lump together all the spam training messages and compress them as a unit. We do the same for the ham. Then when given a new message to classify, we append it to the spam messages and compress the result. We also append it to the ham and compress that. Whichever class compresses better—adds the fewer number of additional bytes for the new message—is the predicted class. The idea is that a spam message will tend to share dictionary entries with other spam messages and thus will compress better when appended to a collection that already contains the spam dictionary.

Experiments with compression-based classification on some of the standard corpora for text classification—the 20-Newsgroups data set, the Reuters-10 Corpora, the Industry Sector corpora—indicate that whereas running off-the-shelf compression algorithms like gzip, RAR, and LZW can be quite slow, their accuracy is comparable to traditional classification algorithms. This is interesting in its own right, and also serves to point out that there is promise for algorithms that use character n-grams directly with no preprocessing of the text or feature selection: they seem to be captiring some real patterns.

3.3 INFORMATION RETRIEVAL

Information retrieval is the task of finding documents that are relevant to a user's need for information. The best-known examples of information retrieval systems are search engines on the World Wide Web.

A Web user can type a query such as [AI book]2 into a search engine and see a list of relevant pages. In this section, we will see how such systems are built. An IR information retrieval (henceforth IR) system can be characterized by

1. A corpus of documents. Each system must decide what it wants to treat as a document: a paragraph, a page, or a multipage text.
2. Queries posed in a query language. A query specifies what the user wants to know. The query language can be just a list of words, such as [AI book]; or it can specify a phrase of words that must be adjacent, as in [“AI book”]; it can contain Boolean operators as in [AI AND book]; it can include non-Boolean operators such as [AI NEAR book] or [AI book site:www.aaai.org].
3. A result set. This is the subset of documents that the IR system judges to be relevant to the query. By relevant, we mean likely to be of use to the person who posed the query, for the particular information need expressed in the query.
4. A presentation of the result set. This can be as simple as a ranked list of document titles or as complex as a rotating color map of the result set projected onto a three-dimensional space, rendered as a two-dimensional display.

The earliest IR systems worked on a Boolean keyword model. Each word in the document collection is treated as a Boolean feature that is true of a document if the word occurs in the document and false if it does not. So the feature “retrieval” is true in this context but false for other contexts. The query language is the language of Boolean expressions over features. A document is relevant only if the expression evaluates to true. For example, the query [information AND retrieval] is true in this context.

This model has the advantage of being simple to explain and implement. However, it has some disadvantages. First, the degree of relevance of a document is a single bit, so there is no guidance as to how to order the relevant documents for presentation. Second, Boolean expressions are unfamiliar to users who are not programmers or logicians. Users find it unintuitive that when they want to know about farming in the states of Kansas and Nebraska they need to issue the query [farming (Kansas OR Nebraska)]. Third, it can be hard to formulate an appropriate query, even for a skilled user. Suppose we try [information AND retrieval AND models AND optimization] and get an empty result set. We could try [information OR retrieval OR models OR optimization], but if that returns too many results, it is difficult to know what to try next.

3.3.1 IR scoring functions

Most IR systems have abandoned the Boolean model and use models based on the statistics of word counts. We describe the BM25 scoring function, BM25 SCORING which comes from the Okapi project of Stephen Robertson and Karen Sparck Jones at London’s City College, and has been used in search engines such as the open-source Lucene project.

A scoring function takes a document and a query and returns a numeric score; the most relevant documents have the highest scores. In the BM25 function, the score is a linear weighted combination of scores for each of the words that make up the query. Three factors affect the weight of a query term: First, the frequency with which a query term appears in a document (also known as TF for term frequency). For the query [farming in Kansas], documents that mention “farming” frequently will have higher scores. Second, the inverse document frequency of the term, or IDF. The word “in” appears in almost every document, so it has a high document frequency, and thus a low inverse document

frequency, and thus it is not as important to the query as "farming" or "Kansas." Third, the length of the document.

A million-word document will probably mention all the query words, but may not actually be about the query. A short document that mentions all the words is a much better candidate. The BM25 function takes all three of these into account. We assume we have created an index of the N documents in the corpus so that we can look up TF(q_i, d_j), the count of the number of times word qi appears in document d_j. We also assume a table of document frequency counts, DF(q_i), that gives the number of documents that contain the word q_i.

Then, given a document d_j and a query consisting of the words $q_{1:N}$, we have:

$$\text{BM25}(d_j, q_{1:N}) = \sum_{i=1}^{N} IDF(q_i).\frac{TF(q_i,d_j).(k+1)}{TF(q_i,d_j)+k.(1-b+b.\frac{|d_j|}{L})}, \quad (9)$$

where $|d_j|$ is the length of document d_j in words, and L is the average document length in the corpus: $L = \frac{\sum_i |d_i|}{N}$. We have two parameters, k and b, that can be tuned by cross-validation; typical values are k = 2.0 and b = 0.75. IDF(q_i) is the inverse document frequency of word qi, given by

$$IDF(q_i) = log\frac{N-DF(q_i)+0.5}{DF(q_i)+0.5}. \quad (10)$$

Of course, it would be impractical to apply the BM25 scoring function to every document in the corpus. Instead, systems create an index ahead of time that lists, for each vocabulary word, the documents that contain the word. This is called the hit list for the word. Then when given a query, we intersect the hit lists of the query words and only score the documents in the intersection.

3.3.2 IR system evaluation

How do we know whether an IR system is performing well? We undertake an experiment in which the system is given a set of queries and the result sets are scored with respect to human relevance judgments. Traditionally, there have been two measures used in the scoring: recall and precision. We explain them with the help of an example. Imagine that an IR system has returned a result set for a single query, for which we know which documents are and are not relevant, out of a corpus of 100 documents. The document counts in each category are given in the following table:

	I result set	Not in result set
Relevant	30	20
Not relevant	10	40

Precision measures the proportion of documents in the result set that are actually relevant. In our example, the precision is 30/(30 + 10)=.75. The false positive rate is 1 − .75=.25. Recall measures the proportion of all the relevant documents in the collection that are in the result set. In our example, recall is 30/(30 + 20)=.60. The false negative rate is 1 − .60=.40. In a very large document collection, such as the World Wide Web, recall is difficult to compute, because there is no easy way to examine every page on the Web for relevance.

All we can do is either estimate recall by sampling or ignore recall completely and just judge precision. In the case of a Web search engine, there may be thousands of documents in the result set, so it makes more sense to measure precision for several different sizes, such as “P@10” (precision in the top 10 results) or “P@50,” rather than to estimate precision in the entire result set.

It is possible to trade off precision against recall by varying the size of the result set returned. In the extreme, a system that returns every document in the document collection is guaranteed a recall of 100%, but will have low precision. Alternately, a system could return a single document and have low recall, but a decent chance at 100% precision. A summary of both measures is the F1 score, a single number that is the harmonic mean of precision and recall, 2PR/(P + R).

3.3.3 IR refinements

There are many possible refinements to the system described here, and indeed Web search engines are continually updating their algorithms as they discover new approaches and as the Web grows and changes.

One common refinement is a better model of the effect of document length on relevance. Singhal et al. (1996) observed that simple document length normalization schemes tend to favor short documents too much and long documents not enough. They propose a pivoted document length normalization scheme; the idea is that the pivot is the document length at which the old-style normalization is correct; documents shorter than that get a boost and longer ones get a penalty.

The BM25 scoring function uses a word model that treats all words as completely independent, but we know that some words are correlated: “couch” is closely related to both “couches” and “sofa.” Many IR systems attempt to account for these correlations.

For example, if the query is [couch], it would be a shame to exclude from the result set those documents that mention “COUCH” or “couches” but not “couch.” Most IR systems do case folding of “COUCH” to “couch,” and use a stemming algorithm to reduce “couches” to the stem form “couch,” both in the query and the documents. This typically yields a small increase in recall (on the order of 2% for English). However, it can harm precision. For example, stemming “stocking” to “stock” will tend to decrease precision for queries about either foot coverings or financial instruments, although it could improve recall for queries about warehousing. Stemming algorithms based on rules (e.g., remove “-ing”) cannot avoid this problem, but algorithms based on dictionaries (don’t remove “-ing” if the word is already listed in the dictionary) can. While stemming has a small effect in English, it is more important in other languages. In German, for example, it is not uncommon to see words like “Lebensversicherungsgesellschaftsangestellter” (life insurance company employee).

Languages such as Finnish, Turkish, Inuit, and Yupik have recursive morphological rules that in principle generate words of unbounded length. The next step is to recognize synonyms, such as “sofa” for “couch.” As with stemming, this has the potential for small gains in recall, but can hurt precision. A user who gives the query [Tim Couch] wants to see results about the football player, not sofas. The problem is that “languages abhor absolute synonyms just as nature abhors a vacuum” (Cruse, 1986). That is, anytime there are two words that mean the same thing, speakers of the language conspire to evolve the meanings to remove the confusion. Related words that are not synonyms also play an important role in ranking—terms like “leather”, “wooden,” or “modern” can serve to confirm that the document really is

about “couch.” Synonyms and related words can be found in dictionaries or by looking for correlations in documents or in queries—if we find that many users who ask the query [new sofa] follow it up with the query [new couch], we can in the future alter [new sofa] to be [new sofa OR new couch].

As a final refinement, IR can be improved by considering metadata—data outside of the text of the document. Examples include human-supplied keywords and publication data. On the Web, hypertext links between documents are a crucial source of information.

3.3.4 The PageRank algorithm

PageRank was one of the two original ideas that set Google’s search apart from other Web search engines when it was introduced in 1997. (The other innovation was the use of anchor text—the underlined text in a hyperlink—to index a page, even though the anchor text was on a different page than the one being indexed.) PageRank was invented to solve the problem of the tyranny of TF scores: if the query is [IBM], how do we make sure that IBM’s home page, ibm.com, is the first result, even if another page mentions the term “IBM” more frequently?

```
function HITS(query) returns pages with hub and authority numbers
  pages ← EXPAND-PAGES(RELEVANT-PAGES(query))
  for each p in pages do
    p.AUTHORITY ← 1
    p.HUB ← 1
  repeat until convergence do
    for each p in pages do
      p.AUTHORITY ← Σ_i INLINK_i(p).HUB
      p.HUB ← Σ_i OUTLINK_i(p).AUTHORITY
    NORMALIZE(pages)
  return pages
```

Figure 1.1 *The HITS algorithm for computing hubs and authorities with respect to a query. RELEVANT-PAGES fetches the pages that match the query, and EXPAND-PAGES adds in every page that links to or is linked from one of the relevant pages. NORMALIZE divides each page’s score by the sum of the squares of all pages’ scores (separately for both the authority and hubs scores).*

The idea is that ibm.com has many in-links (links to the page), so it should be ranked higher: each in-link is a vote for the quality of the linked-to page. But if we only counted in-links, then it would be possible for a Web spammer to create a network of pages and have them all point to a page of his choosing, increasing the score of that page. Therefore, the PageRank algorithm is designed to weight links from high-quality sites more heavily. What is a high-quality site? One that is linked to by other high-quality sites. The definition is recursive, but we will see that the recursion bottoms out properly. The PageRank for a page p is defined as:

$$PR(p) = \frac{1-d}{N} + d\sum_i \frac{PR(in_i)}{C(in_i)}, \qquad (11)$$

where PR(p) is the PageRank of page p, N is the total number of pages in the corpus, in_i are the pages that link in to p, and $C(in_i)$ is the count of the total number of out-links on page ini. The constant d is a damping factor. It can be understood through the random surfer model: imagine a Web surfer who starts at some random page and begins exploring.

With probability d (we'll assume d=0.85) the surfer clicks on one of the links on the page (choosing uniformly among them), and with probability 1 − d she gets bored with the page and restarts on a random page anywhere on the Web. The PageRank of page p is then the probability that the random surfer will be at page p at any point in time. PageRank can be computed by an iterative procedure: start with all pages having PR(p)=1, and iterate the algorithm, updating ranks until they converge.

3.3.5 The HITS algorithm

The Hyperlink-Induced Topic Search algorithm, also known as "Hubs and Authorities" or HITS, is another influential link-analysis algorithm (see Figure 1.1). HITS differs from PageRank in several ways. First, it is a query-dependent measure: it rates pages with respect to a query. That means that it must be computed anew for each query—a computational burden that most search engines have elected not to take on. Given a query, HITS first finds a set of pages that are relevant to the query. It does that by intersecting hit lists of query words, and then adding pages in the link neighborhood of these pages—pages that link to or are linked from one of the pages in the original relevant set.

Each page in this set is considered an authority on the query to the degree that other HUB pages in the relevant set point to it. A page is considered a hub to the degree that it points to other authoritative pages in the relevant set. Just as with PageRank, we don't want to merely count the number of links; we want to give more value to the high-quality hubs and authorities. Thus, as with PageRank, we iterate a process that updates the authority score of a page to be the sum of the hub scores of the pages that point to it, and the hub score to be the sum of the authority scores of the pages it points to. If we then normalize the scores and repeat k times, the process will converge.

Both PageRank and HITS played important roles in developing our understanding of Web information retrieval. These algorithms and their extensions are used in ranking billions of queries daily as search engines steadily develop better ways of extracting yet finer signals of search relevance.

3.4 INFORMATION EXTRACTION

Information extraction is the process of acquiring knowledge by skimming a text and looking for occurrences of a particular class of object and for relationships among objects. A typical task is to extract instances of addresses from Web pages, with database fields for street, city, state, and zip code; or instances of storms from weather reports, with fields for temperature, wind speed, and precipitation. In a limited domain, this can be done with high accuracy. As the domain gets more general, more complex linguistic models and more complex learning techniques are necessary. But so far there are no complete models of this kind, so for the limited needs of information extraction, we define limited models that approximate the full English model, and concentrate on just the parts that are needed for the task at hand. The models we describe in this section are approximations in the same way that the simple 1-CNF logical model is an approximation of the full, wiggly, logical model.

In this section we describe three different approaches to information extraction, in order of increasing complexity on several dimensions.

3.4.1 Finite-state automata for information extraction

The simplest type of information extraction system is an attribute-based extraction system that assumes that the entire text refers to a single object and the task is to extract attributes of that object. For example, the problem of extracting from the text "IBM ThinkBook 970. Our price: $399.00" the set of attributes {Manufacturer=IBM, Model=ThinkBook970, Price=$399.00}. We can address this problem by defining a temTEMPLATE plate (also known as a pattern) for each attribute we would like to extract. The template is defined by a finite state automaton, the simplest example of which is the regular expression, or regex. Regular expressions are used in Unix commands such as grep, in programming languages such as Perl, and in word processors such as Microsoft Word. The details vary slightly from one tool to another and so are best learned from the appropriate manual, but here we show how to build up a regular expression template for prices in dollars:

[0-9]	matches any digit from 0 to 9
[0-9]+	matches one or more digits
[.][0-9][0-9]	matches a period followed by two digits
([.][0-9][0-9])?	matches a period followed by two digits, or nothing
[$][0-9]+([.][0-9][0-9])?	matches $249.99 or $1.23 or $1000000 or . . .

Templates are often defined with three parts: a prefix regex, a target regex, and a postfix regex. For prices, the target regex is as above, the prefix would look for strings such as "price:" and the postfix could be empty. The idea is that some clues about an attribute come from the attribute value itself and some come from the surrounding text.

If a regular expression for an attribute matches the text exactly once, then we can pull out the portion of the text that is the value of the attribute. If there is no match, all we can do is give a default value or leave the attribute missing; but if there are several matches, we need a process to choose among them. One strategy is to have several templates for each attribute, ordered by priority. So, for example, the top-priority template for price might look for the prefix "our price:"; if that is not found, we look for the prefix "price:" and if that is not found, the empty prefix. Another strategy is to take all the matches and find some way to choose among them. For example, we could take the lowest price that is within 50% of the highest price. That will select $78.00 as the target from the text "List price $99.00, special sale price $78.00, shipping $3.00."

One step up from attribute-based extraction systems are relational extraction systems, which deal with multiple objects and the relations among them. Thus, when these systems see the text "$249.99," they need to determine not just that it is a price, but also which object has that price. A typical relational-based extraction system is FASTUS, which handles news stories about corporate mergers and acquisitions. It can read the story

Bridgestone Sports Co. said Friday it has set up a joint venture in Taiwan with a local concern and a Japanese trading house to produce golf clubs to be shipped to Japan. and extract the relations:

$e \in JointVentures \wedge Product(e, \text{“golf clubs”}) \wedge Date(e, \text{“Friday”})$

∧ Member(e, "Bridgestone Sports Co") ∧ Member(e, "a local concern")
∧ Member(e, "a Japanese trading house") .

A relational extraction system can be built as a series of cascaded finite-state transducers. That is, the system consists of a series of small, efficient finite-state automata (FSAs), where each automaton receives text as input, transduces the text into a different format, and passes it along to the next automaton. FASTUS consists of five stages:

1. Tokenization
2. Complex-word handling
3. Basic-group handling
4. Complex-phrase handling
5. Structure merging

FASTUS's first stage is tokenization, which segments the stream of characters into tokens (words, numbers, and punctuation). For English, tokenization can be fairly simple; just separating characters at white space or punctuation does a fairly good job. Some tokenizers also deal with markup languages such as HTML, SGML, and XML.

The second stage handles complex words, including collocations such as "set up" and "joint venture," as well as proper names such as "Bridgestone Sports Co." These are recognized by a combination of lexical entries and finite-state grammar rules. For example, a company name might be recognized by the rule CapitalizedWord+ ("Company" | "Co" | "Inc" | "Ltd")

The third stage handles basic groups, meaning noun groups and verb groups. The idea is to chunk these into units that will be managed by the later stages. The example sentence would emerge from this stage as the following sequence of tagged groups:

1. NG: Bridgestone Sports Co.
2. VG: said
3. NG: Friday
4. NG: it
5. VG: had set up
6. NG: a joint venture
7. PR: in
8. NG: Taiwan
9. PR: with
10 NG: a local concern
11 CJ: and
12 NG: a Japanese trading house
13 VG: to produce
14 NG: golf clubs
15 VG: to be shipped
16 PR: to
17 NG: Japan

Here NG means noun group, VG is verb group, PR is preposition, and CJ is conjunction.

The fourth stage combines the basic groups into complex phrases. Again, the aim is to have rules that are finite-state and thus can be processed quickly, and that result in unambiguous (or nearly unambiguous) output phrases. One type of combination rule deals with domain-specific events. For example, the rule Company+ SetUp JointVenture ("with" Company+)? captures one way to describe the formation of a joint venture. This stage is the first one in the cascade where the output is placed into a database template as well as being placed in the output stream. The final stage merges structures that were built up in the previous step. If the next sentence says "The joint venture will start production in

January," then this step will notice that there are two references to a joint venture, and that they should be merged into one. This is an instance of the identity uncertainty problem.

In general, finite-state template-based information extraction works well for a restricted domain in which it is possible to predetermine what subjects will be discussed, and how they will be mentioned. The cascaded transducer model helps modularize the necessary knowledge, easing construction of the system. These systems work especially well when they are reverse-engineering text that has been generated by a program. For example, a shopping site on the Web is generated by a program that takes database entries and formats them into Web pages; a template-based extractor then recovers the original database. Finite-state information extraction is less successful at recovering information in highly variable format, such as text written by humans on a variety of subjects.

3.4.2 Probabilistic models for information extraction

When information extraction must be attempted from noisy or varied input, simple finite-state approaches fare poorly. It is too hard to get all the rules and their priorities right; it is better to use a probabilistic model rather than a rule-based model. The simplest probabilistic model for sequences with hidden state is the hidden Markov model, or HMM.

Recall from Section 15.3 that an HMM models a progression through a sequence of hidden states, xt, with an observation et at each step. To apply HMMs to information extraction, we can either build one big HMM for all the attributes or build a separate HMM for each attribute. We'll do the second. The observations are the words of the text, and the hidden states are whether we are in the target, prefix, or postfix part of the attribute template, or in the background (not part of a template). For example, here is a brief text and the most probable (Viterbi) path for that text for two HMMs, one trained to recognize the speaker in a talk announcement, and one trained to recognize dates. The "-" indicates a background state:

Text:	There	will	be	a	seminar	by	Dr.	Andrew	McCallum	on	Friday
Speaker:	-	-	-	-	PRE	PRE	TARGET	TARGET	TARGET	POST	-
Date:	-	-	-	-	-	-	-	-	-	PRE	TARGET

HMMs have two big advantages over FSAs for extraction. First, HMMs are probabilistic, and thus tolerant to noise. In a regular expression, if a single expected character is missing, the regex fails to match; with HMMs there is graceful degradation with missing characters/words, and we get a probability indicating the degree of match, not just a Boolean match/fail. Second, HMMs can be trained from data; they don't require laborious engineering of templates, and thus they can more easily be kept up to date as text changes over time.

Note that we have assumed a certain level of structure in our HMM templates: they all consist of one or more target states, and any prefix states must precede the targets, postfix states most follow the targets, and other states must be background. This structure makes it easier to learn HMMs from examples. With a partially specified structure, the forward–backward algorithm can be used to learn both the transition probabilities $P(X_t \mid X_{t-1})$ between states and the observation model, $P(E_t \mid X_t)$, which says how likely each word is in each state. For example, the word "Friday" would have high probability in one or more of the target states of the date HMM, and lower probability elsewhere.

With sufficient training data, the HMM automatically learns a structure of dates that we find intuitive: the date HMM might have one target state in which the high-probability words are “Monday,” “Tuesday,” etc., and which has a high-probability transition to a target state with words “Jan”, “January,” “Feb,” etc. Figure 22.2 shows the HMM for the speaker of a talk announcement, as learned from data. The prefix covers expressions such as “Speaker:” and “seminar by,” and the target has one state that covers titles and first names and another state that covers initials and last names.

Once the HMMs have been learned, we can apply them to a text, using the Viterbi algorithm to find the most likely path through the HMM states. One approach is to apply each attribute HMM separately; in this case you would expect most of the HMMs to spend most of their time in background states. This is appropriate when the extraction is sparse—when the number of extracted words is small compared to the length of the text.

The other approach is to combine all the individual attributes into one big HMM, which would then find a path that wanders through different target attributes, first finding a speaker target, then a date target, etc. Separate HMMs are better when we expect just one of each attribute in a text and one big HMM is better when the texts are more free-form and dense with attributes. With either approach, in the end we have a collection of target attribute observations, and have to decide what to do with them. If every expected attribute has one target filler then the decision is easy: we have an instance of the desired relation. If there are multiple fillers, we need to decide which to choose, as we discussed with template-based systems. HMMs have the advantage of supplying probability numbers that can help make the choice. If some targets are missing, we need to decide if this is an instance of the desired relation at all, or if the targets found are false positives. A machine learning algorithm can be trained to make this choice.

3.4.3 Conditional random fields for information extraction

One issue with HMMs for the information extraction task is that they model a lot of probabilities that we don’t really need. An HMM is a generative model; it models the full joint probability of observations and hidden states, and thus can be used to generate samples. That is, we can use the HMM model not only to parse a text and recover the speaker and date, but also to generate a random instance of a text containing a speaker and a date. Since we’re not interested in that task, it is natural to ask whether we might be better off with a model that doesn’t bother modeling that possibility. All we need in order to understand a text is a discriminative model, one that models the conditional probability of the hidden attributes given the observations (the text). Given a text e1:N, the conditional model finds the hidden state sequence X1:N that maximizes $P(X_{1:N} \mid e_{1:N})$.

Modeling this directly gives us some freedom. We don’t need the independence assumptions of the Markov model—we can have an xt that is dependent on x1. A framework for this type of model is the conditional random CONDITIONAL field, or CRF, which models a conditional probability distribution of a set of target variables given a set of observed variables. Like Bayesian networks, CRFs can represent many different structures of dependencies among the variables. One common structure is the linear-chain conditional random field for representing Markov dependencies among variables in a temporal sequence. Thus, HMMs are the temporal version of naive Bayes models, and linear-chain CRFs are the temporal version of logistic regression, where the predicted target is an entire state sequence rather than a single binary variable.

$$\boldsymbol{P}(\boldsymbol{X}_{1:N}|\boldsymbol{e}_{1:N}) = \alpha\, e^{[\sum_{i=1}^{N} F(\boldsymbol{X}_{i-1}, \boldsymbol{X}_i, \boldsymbol{e}, i)]} \quad (12)$$

where α is a normalization factor (to make sure the probabilities sum to 1), and F is a feature function defined as the weighted sum of a collection of k component feature functions:

$$F(\boldsymbol{X}_{i-1}, \boldsymbol{X}_i, \boldsymbol{e}, i) = \sum_k \lambda_k f_k(\boldsymbol{X}_{i-1}, \boldsymbol{X}_i, \boldsymbol{e}, i) \quad (13)$$

The λ_k parameter values are learned with a MAP (maximum a posteriori) estimation procedure that maximizes the conditional likelihood of the training data. The feature functions are the key components of a CRF. The function fk has access to a pair of adjacent states, x_{i-1} and x_i, but also the entire observation (word) sequence e, and the current position in the temporal sequence, i. This gives us a lot of flexibility in defining features. We can define a simple feature function, for example one that produces a value of 1 if the current word is ANDREW and the current state is SPEAKER:

$$f_1(\boldsymbol{X}_{i-1}, \boldsymbol{X}_i, \boldsymbol{e}, i) = \begin{cases} 1 & \text{if } \boldsymbol{X}_i = SPEAKER \text{ and } \boldsymbol{e}_i = ANDREW \\ 0 & otherwise \end{cases} \quad (14)$$

How are features like these used? It depends on their corresponding weights. If $\lambda_1 > 0$, then whenever f_1 is true, it increases the probability of the hidden state sequence $x_{1:N}$. This is another way of saying "the CRF model should prefer the target state SPEAKER for the word ANDREW." If on the other hand $\lambda_1 < 0$, the CRF model will try to avoid this association, and if $\lambda_1 = 0$, this feature is ignored. Parameter values can be set manually or can be learned from data. Now consider a second feature function:

$$f_2(\boldsymbol{X}_{i-1}, \boldsymbol{X}_i, \boldsymbol{e}, i) = \begin{cases} 1 & \text{if } \boldsymbol{X}_i = SPEAKER \text{ and } \boldsymbol{e}_{i+1} = SAID \\ 0 & otherwise \end{cases} \quad (15)$$

This feature is true if the current state is SPEAKER and the next word is "said." One would therefore expect a positive λ2 value to go with the feature. More interestingly, note that both f1 and f2 can hold at the same time for a sentence like "Andrew said . . . ." In this case, the two features overlap each other and both boost the belief in x1 = SPEAKER. Because of the independence assumption, HMMs cannot use overlapping features; CRFs can. Furthermore, a feature in a CRF can use any part of the sequence $e_{1:N}$. Features can also be defined over transitions between states. The features we defined here were binary, but in general, a feature function can be any real-valued function. For domains where we have some knowledge about the types of features we would like to include, the CRF formalism gives us a great deal of flexibility in defining them. This flexibility can lead to accuracies that are higher than with less flexible models such as HMMs.

4.0 Conclusion

The main points of this chapter are as follows:

- Probabilistic language models based on n-grams recover a surprising amount of information about a language. They can perform well on such diverse tasks as language identification, spelling correction, genre classification, and named-entity recognition.

- These language models can have millions of features, so feature selection and preprocessing of the data to reduce noise is important.
- Text classification can be done with naive Bayes n-gram models or with any of the classification algorithms we have previously discussed. Classification can also be seen as a problem in data compression.
- Information retrieval systems use a very simple language model based on bags of words, yet still manage to perform well in terms of recall and precision on very large corpora of text. On Web corpora, link-analysis algorithms improve performance.
- Question answering can be handled by an approach based on information retrieval, for questions that have multiple answers in the corpus. When more answers are available in the corpus, we can use techniques that emphasize precision rather than recall.
- Information-extraction systems use a more complex model that includes limited notions of syntax and semantics in the form of templates. They can be built from finite state automata, HMMs, or conditional random fields, and can be learned from examples.
- In building a statistical language system, it is best to devise a model that can make good use of available data, even if the model seems overly simplistic.

5.0 Summary

We hope you enjoyed this unit. This unit explains how AI is applied to NLP. Now, let us attempt the questions below.

6.0 Tutor Marked Assignment

i. This exercise explores the quality of the n-gram model of language. Find or create a monolingual corpus of 100,000 words or more. Segment it into words, and compute the frequency of each word. How many distinct words are there? Also count frequencies of bigrams (two consecutive words) and trigrams (three consecutive words). Now use those frequencies to generate language: from the unigram, bigram, and trigram models, in turn, generate a 100-word text by making random choices according to the frequency counts. Compare the three generated texts with actual language. Finally, calculate the perplexity of each model.

7.0 References/Further Readings

1) Russell, S., & Norvig, P. (2005). AI a modern approach. Learning, 2(3), 4.
2) Rich E., & Knight K. (1991) Artificial Intelligence, McGraw-Hill, New York, NY
3) Shi, Z. (2011). Advanced artificial intelligence (Vol. 1). World Scientific.
4) DataFlair Team, What is Natural Language Processing in Artificial Intelligence? https://data-flair.training/blogs/ai-natural-language-processing/. Published January 24, 2018 · Updated November 15, 2018. Retrieved 07/29/2019

Unit Two: Scene Analysis,

CONTENT

1.0 Introduction

A video camera for robotic applications might produce a million 24-bit pixels at 60 Hz; a rate of 10 GB per minute. The problem for a vision-capable agent then is: Which aspects of the rich visual stimulus should be considered to help the agent make good action choices, and which aspects should be ignored? Vision—and all perception—serves to further the agent's goals, not as an end to itself.

We can characterize three FEATURE broad approaches to the problem. The feature extraction approach, as exhibited by Drosophila, emphasizes simple computations applied directly to the sensor observations. In the recognition approach an agent draws distinctions among the objects it encounters based on visual and other information. Recognition could mean labeling each image with a yes or no as to whether it contains food that we should forage, or contains Grandma's face. Finally, in the reconstruction approach an agent builds a geometric model of the world from an image or a set of images.

The last thirty years of research have produced powerful tools and methods for addressing these approaches. Understanding these methods requires an understanding of the processes by which images are formed. Therefore, we now cover the physical and statistical phenomena that occur in the production of an image.

2.0 Objectives

At the end of this section, you should be able to:

- Outline the major information derivable from an image
- Perform early image-processing operations
- Carry out object recognition by appearance
- Perform operations for reconstructing a 3D world scene

3.0 Main Content

3.1 IMAGE FORMATION

Imaging distorts the appearance of objects. For example, a picture taken looking down a long straight set of railway tracks will suggest that the rails converge and meet. As another example, if you hold your hand in front of your eye, you can block out the moon, which is not smaller than your hand. As you move your hand back and forth or tilt it, your hand will seem to shrink and grow in the image, but it is not doing so in reality (Figure 24.1). Models of these effects are essential for both recognition and reconstruction.

3.1.1 Images without lenses: The pinhole camera

Image sensors gather light scattered from objects in a scene and create a two-dimensional image. In the eye, the image is formed on the retina, which consists of two types of cells: about 100 million rods, which are sensitive to light at a wide range of wavelengths, and 5 million cones. Cones, which are essential for color vision, are of three main types, each of which is sensitive to a different set of wavelengths. In cameras, the image is formed on an image plane, which can be a piece of film coated with silver halides or a rectangular grid of a few million photosensitive pixels, each a complementary

metal-oxide semiconductor (CMOS) or charge-coupled device (CCD). Each photon arriving at the sensor produces an effect, whose strength depends on the wavelength of the photon. The output of the sensor is the sum of all effects due to photons observed in some time window, meaning that image sensors report a weighted average of the intensity of light arriving at the sensor.

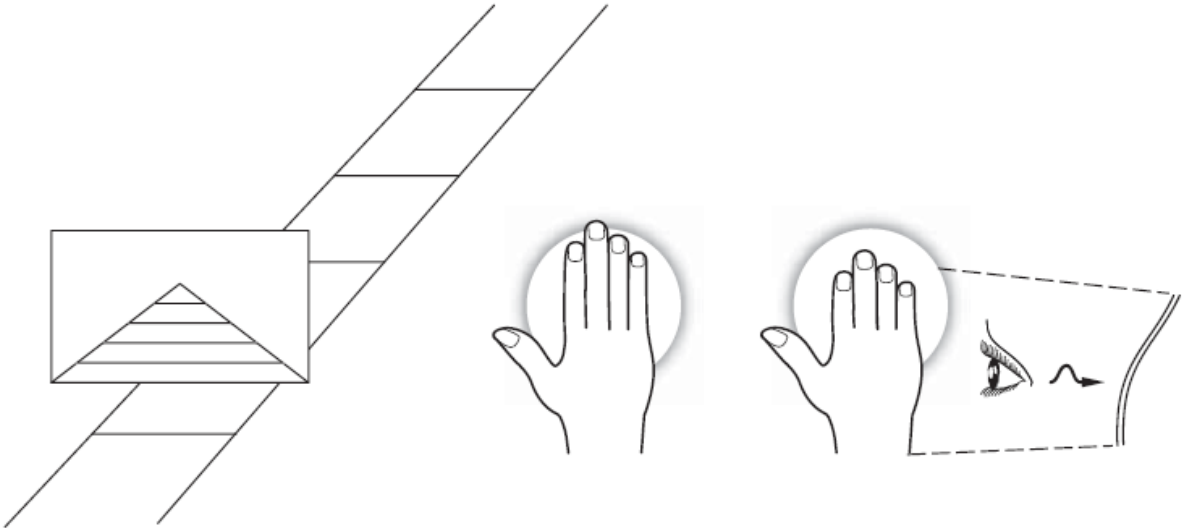


Figure 3.1 Imaging distorts geometry. Parallel lines appear to meet in the distance, as in the image of the railway tracks on the left. In the center, a small hand blocks out most of a large moon. On the right is a foreshortening effect: the hand is tilted away from the eye, making it appear shorter than in the center figure.

To see a focused image, we must ensure that all the photons from approximately the same spot in the scene arrive at approximately the same point in the image plane. The simplest way to form a focused image is to view stationary objects with a pinhole camera, which consists of a pinhole opening, O, at the front of a box, and an image plane at the back of the box (Figure 3.2). Photons from the scene must pass through the pinhole, so if it is small enough then nearby photons in the scene will be nearby in the image plane, and the image will be in focus.

The geometry of scene and image is easiest to understand with the pinhole camera. We use a three-dimensional coordinate system with the origin at the pinhole, and consider a point P in the scene, with coordinates (X, Y,Z). P gets projected to the point P' in the image plane with coordinates (x, y, z). If f is the distance from the pinhole to the image plane, then by similar triangles, we can derive the following equations:

$$\frac{-x}{f} = \frac{X}{Z}, \frac{-y}{f} = \frac{Y}{Z} \Rightarrow x = \frac{-fX}{Z}, y = \frac{-fY}{Z}$$

These equations define an image-formation process known as perspective projection. Note hat the Z in the denominator means that the farther away an object is, the smaller its image will be. Also, note that the minus signs mean that the image is inverted, both left–right and up–down, compared with the scene.

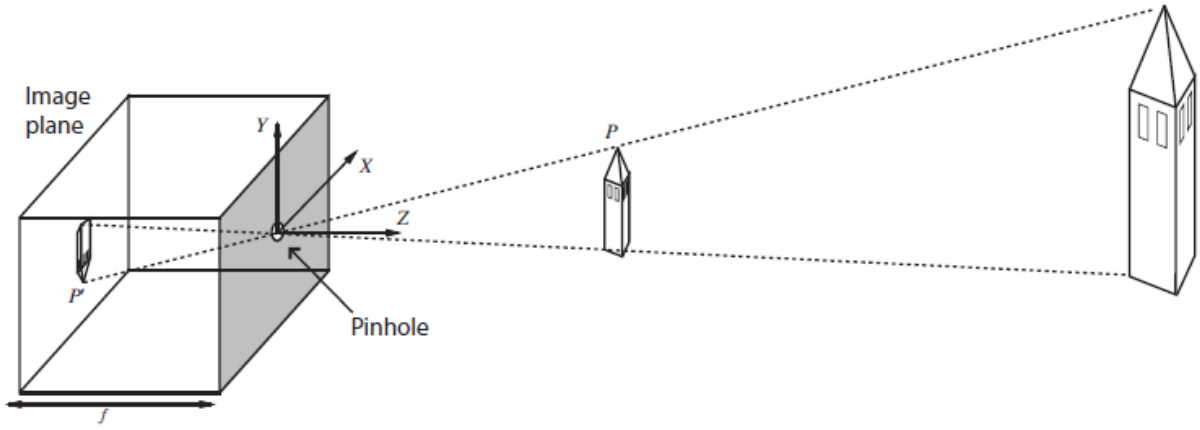


Figure 3.2 Each light-sensitive element in the image plane at the back of a pinhole camera receives light from a the small range of directions that passes through the pinhole. If the pinhole is small enough, the result is a focused image at the back of the pinhole. The process of projection means that large, distant objects look the same as smaller, nearby objects. Note that the image is projected upside down.

Under perspective projection, distant objects look small. This is what allows you to cover the moon with your hand (Figure 3.1). An important result of this effect is that parallel lines converge to a point on the horizon. (Think of railway tracks, Figure 3.1.) A line in the scene in the direction (U, V,W) and passing through the point (X_0, Y_0, Z_0) can be described as the set of points ($X_0 + \lambda U$, $Y_0 + \lambda V$,$Z_0 + \lambda W$), with λ varying between $-\infty$ and $+\infty$.

Different choices of (X_0, Y_0, Z_0) yield different lines parallel to one another. The projection of a point P_λ from this line onto the image plane is given by

$$f\frac{X_0+\lambda U}{Z_0+\lambda W}, ff\frac{Y_0+\lambda V}{Z_0+\lambda W}. \qquad (1)$$

As $\lambda \rightarrow \infty$or $\lambda \rightarrow -\infty$, this becomes $p \infty = (fU/W, fV/W)$ if $W \neq 0$. This means that two parallel lines leaving different points in space will converge in the image—for large λ, the image points are nearly the same, whatever the value of (X_0, Y_0, Z_0) (again, think railway tracks, Figure 24.1). We call $p \infty$ the vanishing point associated with the family of straight lines with direction (U, V, W). Lines with the same direction share the same vanishing point.

3.1.2 Lens systems

The drawback of the pinhole camera is that we need a small pinhole to keep the image in focus. But the smaller the pinhole, the fewer photons get through, meaning the image will be dark. We can gather more photons by keeping the pinhole open longer, but then we will get motion blur—objects in the scene that move will appear blurred because they send photons to multiple locations on the image plane. If we can't keep the pinhole open longer, we can try to make it bigger. More light will enter, but light from a small patch of object in the scene will now be spread over a patch on the image plane, causing a blurred image.

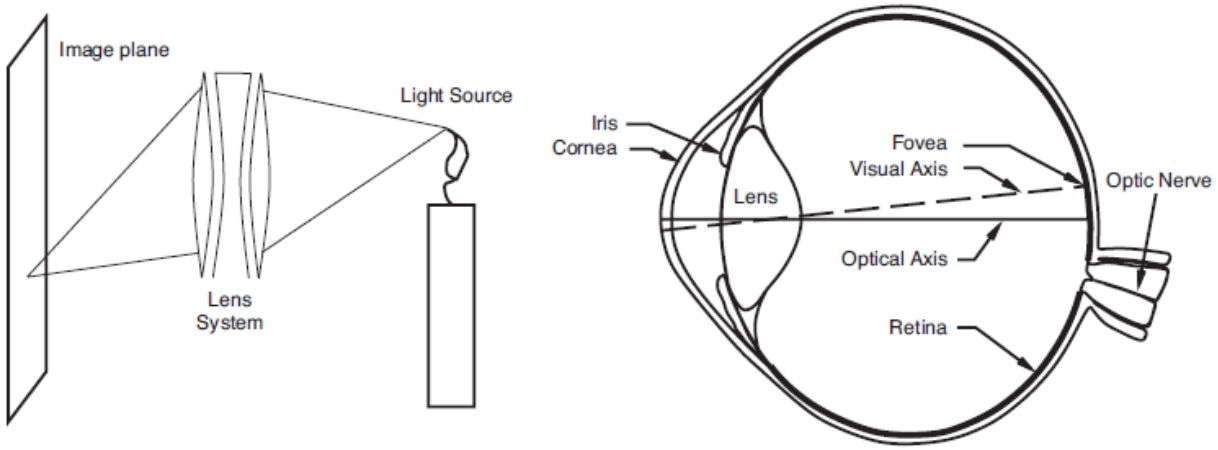


Figure 3.3 Lenses collect the light leaving a scene point in a range of directions, and steer it all to arrive at a single point on the image plane. Focusing works for points lying close to a focal plane in space; other points will not be focused properly. In cameras, elements of the lens system move to change the focal plane, whereas in the eye, the shape of the lens is changed by specialized muscles.

Vertebrate eyes and modern cameras use a lens system to gather sufficient light while keeping the image in focus. A large opening is covered with a lens that focuses light from nearby object locations down to nearby locations in the image plane. However, lens systems have a limited depth of field: they can focus light only from points that lie within a range of depths (centered around a focal plane). Objects outside this range will be out of focus in the image. To move the focal plane, the lens in the eye can change shape (Figure 24.3); in a camera, the lenses move back and forth.

3.1.3 Scaled orthographic projection

Perspective effects aren't always pronounced. For example, spots on a distant leopard may look small because the leopard is far away, but two spots that are next to each other will have about the same size. This is because the difference in distance to the spots is small compared to the distance to them, and so we can simplify the projection model. The appropriate model is scaled orthographic projection. The idea is as follows: If the depth Z of points on the object varies within some range $Z_0 \pm \Delta Z$, with $\Delta Z * Z_0$, then the perspective scaling factor f/Z can be approximated by a constant $s = f/Z_0$. The equations for projection from the scene coordinates (X, Y,Z) to the image plane become $x = sX$ and $y = sY$. Scaled orthographic projection is an approximation that is valid only for those parts of the scene with not much internal depth variation. For example, scaled orthographic projection can be a good model for the features on the front of a distant building.

3.1.4 Light and shading

The brightness of a pixel in the image is a function of the brightness of the surface patch in the scene that projects to the pixel. We will assume a linear model (current cameras have nonlinearities at the extremes of light and dark, but are linear in the middle). Image brightness is a strong, if ambiguous, cue to the shape of an object, and from there to its identity. People are usually able to distinguish the three main causes of varying brightness and reverse-engineer the object's properties. The first cause is overall

intensity of the light. Even though a white object in shadow may be less bright than a black object in direct sunlight, the eye can distinguish relative brightness well, and perceive the white object as white. Second, different points in the scene may reflect more or less of the light. Usually, the result is that people perceive these points as lighter or darker, and so see texture or markings on the object. Third, surface patches facing the light are brighter than surface patches tilted away from the light, an effect known as shading. Typically, people can tell that this shading comes from the geometry of the object, but sometimes get shading and markings mixed up. For example, a streak of dark makeup under a cheekbone will often look like a shading effect, making the face look thinner.

Most surfaces reflect light by a process of diffuse reflection. Diffuse reflection scatters light evenly across the directions leaving a surface, so the brightness of a diffuse surface doesn't depend on the viewing direction. Most cloth, paints, rough wooden surfaces, vegetation, and rough stone are diffuse. Mirrors are not diffuse, because what you see depends on the direction in which you look at the mirror. The behavior of a perfect mirror is known as specular reflection. Some surfaces—such as brushed metal, plastic, or a wet floor—display small patches where specular reflection has occurred, called specularities. These are easy to identify, because they are small and bright (Figure 24.4). For almost all purposes, it is enough to model all surfaces as being diffuse with specularities.

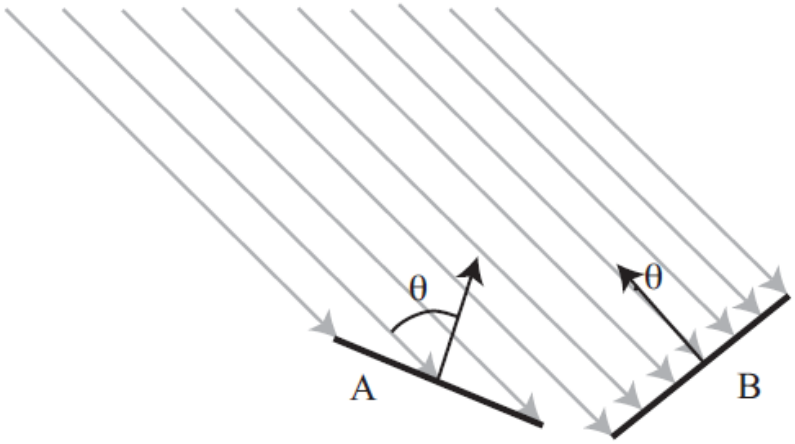


Figure 3.5 Two surface patches are illuminated by a distant point source, whose rays are shown as gray arrowheads. Patch A is tilted away from the source (ϑ is close to 90^0) and collects less energy, because it cuts fewer light rays per unit surface area. Patch B, facing the source (ϑ is close to 0^0), collects more energy.

The main source of illumination outside is the sun, whose rays all travel parallel to one another. We model this behavior as a distant point light source. This is the most important model of lighting, and is quite effective for indoor scenes as well as outdoor scenes. The amount of light collected by a surface patch in this model depends on the angle θ between the illumination direction and the normal to the surface.

A diffuse surface patch illuminated by a distant point light source will reflect some fraction of the light it collects; this fraction is called the diffuse albedo. White paper and snow have a high albedo, about 0.90, whereas flat black velvet and charcoal have a low albedo of about 0.05 (which means that 95% of the

incoming light is absorbed within the fibers of the velvet or the pores of the charcoal). Lambert's cosine law states that the brightness of a diffuse patch is given by

$$I = \rho I_0 \cos\theta, \quad (2)$$

where ρ is the diffuse albedo, I_0 is the intensity of the light source and θ is the angle between the light source direction and the surface normal (see Figure 24.5). Lampert's law predicts bright image pixels come from surface patches that face the light directly and dark pixels come from patches that see the light only tangentially, so that the shading on a surface provides some shape information. We explore this cue in Section 24.4.5. If the surface is not reached by the light source, then it is in shadow. Shadows are very seldom a uniform black, because the shadowed surface receives some light from other sources. Outdoors, the most important such source is the sky, which is quite bright. Indoors, light reflected from other surfaces illuminates shadowed patches. These interreflections can have a significant effect on the brightness of other surfaces, too. These effects are sometimes modeled by adding a constant ambient illumination term to the predicted intensity.

3.1.5 Color

Fruit is a bribe that a tree offers to animals to carry its seeds around. Trees have evolved to have fruit that turns red or yellow when ripe, and animals have evolved to detect these color changes. Light arriving at the eye has different amounts of energy at different wavelengths; this can be represented by a spectral energy density function. Human eyes respond to light in the 380–750nm wavelength region, with three different types of color receptor cells, which have peak receptiveness at 420mm (blue), 540nm (green), and 570nm (red). The human eye can capture only a small fraction of the full spectral energy density function—but it is enough to tell when the fruit is ripe.

The principle of trichromacy states that for PRINCIPLE OF any spectral energy density, no matter how complicated, it is possible to construct another spectral energy density consisting of a mixture of just three colors—usually red, green, and blue—such that a human can't tell the difference between the two. That means that our TVs and computer displays can get by with just the three red/green/blue (or R/G/B) color elements. It makes our computer vision algorithms easier, too. Each surface can be modeled with three different albedos for R/G/B. Similarly, each light source can be modeled with three R/G/B intensities. We then apply Lambert's cosine law to each to get three R/G/B pixel values. This model predicts, correctly, that the same surface will produce different colored image patches under different-colored lights. In fact, human observers are quite good at ignoring the effects of different colored lights and are able to estimate the color of the surface under white light, an effect known as color constancy.

Quite accurate color constancy algorithms are now available; simple versions show up in the "auto white balance" function of your camera. Note that if we wanted to build a camera for mantis shrimp, we would need 12 different pixel colors, corresponding to the 12 types of color receptors of the crustacean.

3.2 EARLY IMAGE-PROCESSING OPERATIONS

We have seen how light reflects off objects in the scene to form an image consisting of, say, five million 3-byte pixels. With all sensors there will be noise in the image, and in any case there is a lot of data to deal with. So how do we get started on analyzing this data?

In this section we will study three useful image-processing operations: edge detection, texture analysis, and computation of optical flow. These are called “early” or “low-level” operations because they are the first in a pipeline of operations. Early vision operations are characterized by their local nature (they can be carried out in one part of the image without regard for anything more than a few pixels away) and by their lack of knowledge: we can perform these operations without consideration of the objects that might be present in the scene. This makes the low-level operations good candidates for implementation in parallel hardware—either in a graphics processor unit (GPU) or an eye. We will then look at one mid-level operation: segmenting the image into regions.

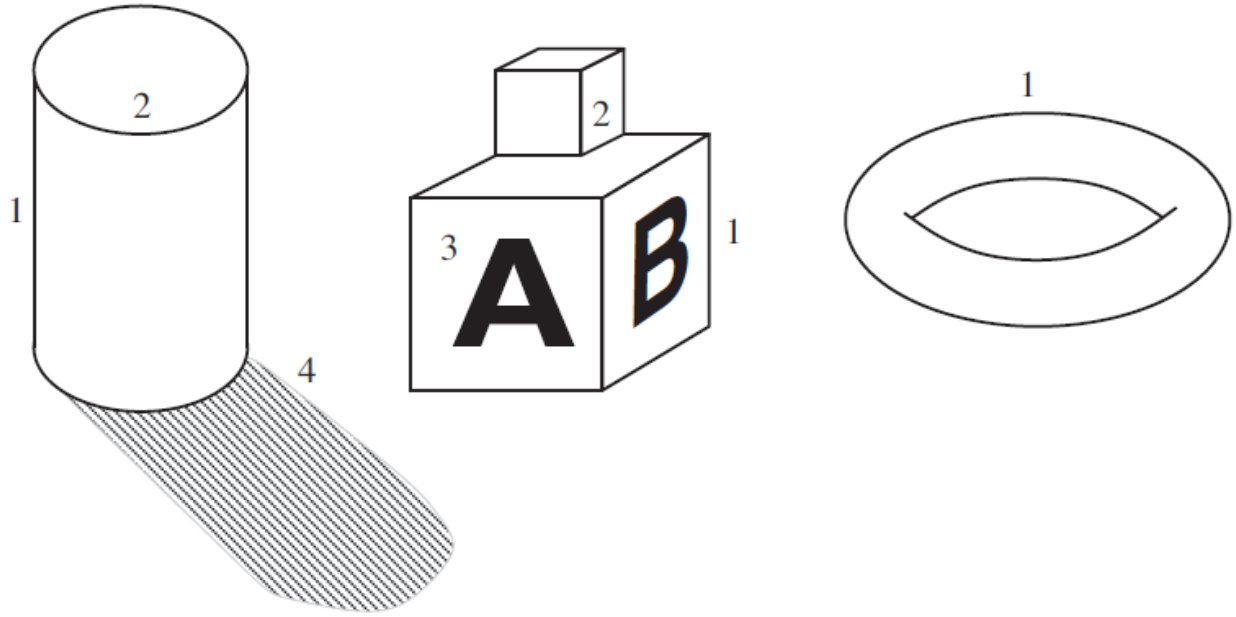


Figure 3.6 Different kinds of edges: (1) depth discontinuities; (2) surface orientation discontinuities; (3) reflectance discontinuities; (4) illumination discontinuities (shadows).

3.2.1 Edge detection

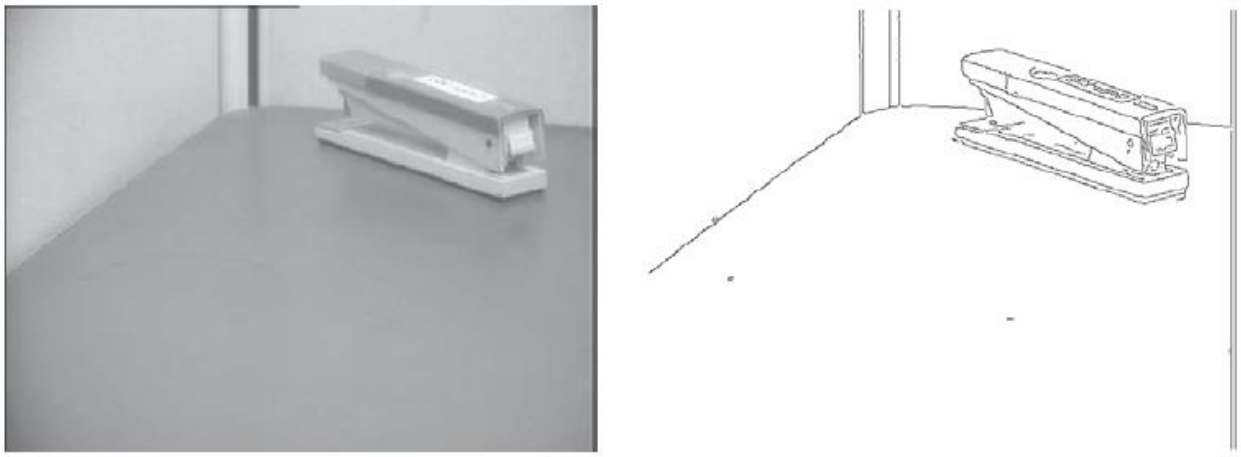
Edges are straight lines or curves EDGE in the image plane across which there is a “significant” change in image brightness. The goal of edge detection is to abstract away from the messy, multimegabyte image and toward a more compact, abstract representation, as in Figure 3.6. The motivation is that edge contours in the image correspond to important scene contours.

In the figure we have three examples of depth discontinuity, labeled 1; two surface-normal discontinuities, labeled 2; a reflectance discontinuity, labeled 3; and an illumination discontinuity (shadow), labeled 4. Edge detection is concerned only with the image, and thus does not distinguish between these different types of scene discontinuities; later processing will.

Figure 3.7(a) shows an image of a scene containing a stapler resting on a desk, and (b) shows the output of an edge-detection algorithm on this image. As you can see, there is a difference between the output and an ideal line drawing. There are gaps where no edge appears, and there are “noise” edges that do not correspond to anything of significance in the scene. Later stages of processing will have to correct for these errors.

How do we detect edges in an image? Consider the profile of image brightness along a one-dimensional cross-section perpendicular to an edge—for example, the one between the left edge of the desk and the wall. It looks something like what is shown in Figure 24.8 (top). Edges correspond to locations in images where the brightness undergoes a sharp change, so a naive idea would be to differentiate the image and look for places where the magnitude of the derivative I'(x) is large. That almost works. In Figure 24.8 (middle), we see that there is indeed a peak at x=50, but there are also subsidiary peaks at other locations (e.g., x=75).

These arise because of the presence of noise in the image. If we smooth the image first, the spurious peaks are diminished, as we see in the bottom of the figure.



(a) (b)

Figure 3.7 (a) Photograph of a stapler. (b) Edges computed from (a).

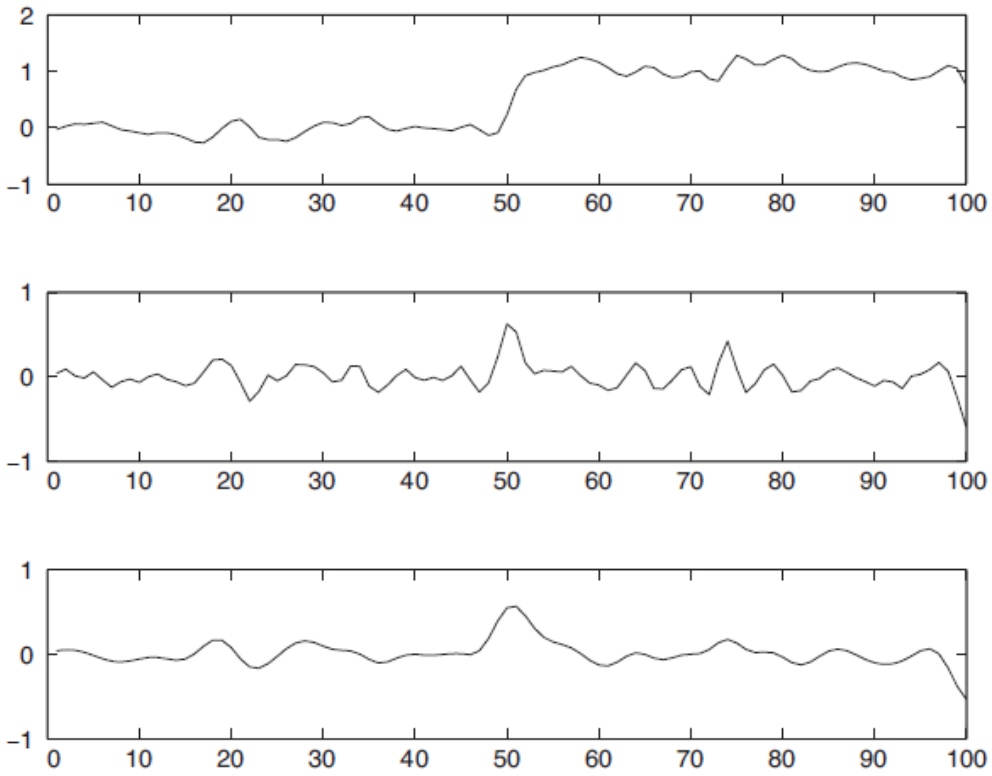


Figure 3.8 Top: Intensity profile I(x) along a one-dimensional section across an edge at x=50. Middle: The derivative of intensity, I'(x). Large values of this function correspond to edges, but the function is noisy. Bottom: The derivative of a smoothed version of the intensity, (I ∗ Gσ)', which can be computed in one step as the convolution I ∗G'$_\sigma$. The noisy candidate edge at x=75 has disappeared.

The measurement of brightness at a pixel in a CCD camera is based on a physical process involving the absorption of photons and the release of electrons; inevitably there will be statistical fluctuations of the measurement—noise. The noise can be modeled with a Gaussian probability distribution, with each pixel independent of the others. One way to smooth an image is to assign to each pixel the average of its neighbors. This tends to cancel out extreme values. But how many neighbors should we consider—one pixel away, or two, or more? One good answer is a weighted average that weights the nearest pixels the most, then gradually decreases the weight for more distant pixels. The Gaussian filter does just that.

(Users of Photoshop recognize this as the Gaussian blur operation.) Recall that the Gaussian function with standard deviation σ and mean 0 is

$$N_\sigma(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-x^2/2\sigma^2} \quad \text{in one dimension, or} \qquad (3)$$

$$N_\sigma(x,y) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-(x^2+x^2)/2\sigma^2} \quad \text{in two dimensions.} \qquad (4)$$

The application of the Gaussian filter replaces the intensity $I(x_0, y_0)$ with the sum, over all (x, y) pixels, of $I(x, y)N_\sigma(d)$, where d is the distance from (x_0, y_0) to (x, y). This kind of weighted sum is so common that

there is a special name and notation for it. We say that the function h is the convolution of two functions f and g (denoted $f * g$) if we have

$h(x) = (f * g)(x) = \sum_{u=-\infty}^{+\infty} f(u)\, g(x - u)$ in one dimension, or (5)

$h(x, y) = (f * g)(x, y) = \sum_{u=-\infty}^{+\infty} \sum_{u=-\infty}^{+\infty} f(u, v)\, g(x - u, y - v)$ in two dimensions. (6)

So the smoothing function is achieved by convolving the image with the Gaussian, $I * N_\sigma$. A σ of 1 pixel is enough to smooth over a small amount of noise, whereas 2 pixels will smooth a larger amount, but at the loss of some detail. Because the Gaussian's influence fades quickly at a distance, we can replace the ±∞ in the sums with ±3σ.

We can optimize the computation by combining smoothing and edge finding into a single operation. It is a theorem that for any functions f and g, the derivative of the convolution, $(f * g)'$, is equal to the convolution with the derivative, $f * (g')$. So rather than smoothing the image and then differentiating, we can just convolve the image with the derivative of the smoothing function, N'_σ. We then mark as edges those peaks in the response that are above some threshold.

There is a natural generalization of this algorithm from one-dimensional cross sections to general two-dimensional images. In two dimensions edges may be at any angle θ. Considering the image brightness as a scalar function of the variables x, y, its gradient is a vector

$$\nabla I = \begin{pmatrix} \frac{\partial I}{\partial x} \\ \frac{\partial I}{\partial y} \end{pmatrix} = \begin{pmatrix} I_x \\ I_y \end{pmatrix} \quad (7)$$

Edges correspond to locations in images where the brightness undergoes a sharp change, and so the magnitude of the gradient, $\|\nabla I\|$, should be large at an edge point. Of independent interest is the direction of the gradient

$$\frac{\nabla I}{\|\nabla I\|} = \begin{pmatrix} \cos\theta \\ \sin\theta \end{pmatrix} \quad (8)$$

This gives us a θ = θ(x, y) at every pixel, which defines the edge orientation at that pixel.

As in one dimension, to form the gradient we don't compute ∇I, but rather $\nabla(I * N_\sigma)$, the gradient after smoothing the image by convolving it with a Gaussian. And again, the shortcut is that this is equivalent to convolving the image with the partial derivatives of a Gaussian. Once we have computed the gradient, we can obtain edges by finding edge points and linking them together. To tell whether a point is an edge point, we must look at other points a small distance forward and back along the direction of the gradient. If the gradient magnitude at one of these points is larger, then we could get a better edge point by shifting the edge curve very slightly. Furthermore, if the gradient magnitude is too small, the point cannot be an edge point. So at an edge point, the gradient magnitude is a local maximum along the direction of the gradient, and the gradient magnitude is above a suitable threshold. Once we have marked edge pixels by this algorithm, the next stage is to link those pixels that belong to the same edge curves. This can be done by assuming that any two neighboring edge pixels with consistent orientations must belong to the same edge curve.

3.2.2 Texture

In everyday language, texture is the visual feel of a surface—what you see evokes what the surface might feel like if you touched it ("texture" has the same root as "textile"). In computational vision, texture refers to a spatially repeating pattern on a surface that can be sensed visually. Examples include the pattern of windows on a building, stitches on a sweater, spots on a leopard, blades of grass on a lawn, pebbles on a beach, and people in a stadium.

Sometimes the arrangement is quite periodic, as in the stitches on a sweater; in other cases, such as pebbles on a beach, the regularity is only statistical. Whereas brightness is a property of individual pixels, the concept of texture makes sense only for a multipixel patch. Given such a patch, we could compute the orientation at each pixel, and then characterize the patch by a histogram of orientations. The texture of bricks in a wall would have two peaks in the histogram (one vertical and one horizontal), whereas the texture of spots on a leopard's skin would have a more uniform distribution of orientations.

Figure 24.9 shows that orientations are largely invariant to changes in illumination. This makes texture an important clue for object recognition, because other clues, such as edges, can yield different results in different lighting conditions. In images of textured objects, edge detection does not work as well as it does for smooth objects. This is because the most important edges can be lost among the texture elements. Quite literally, we may miss the tiger for the stripes. The solution is to look for differences in texture properties, just the way we look for differences in brightness. A patch on a tiger and a patch on the grassy background will have very different orientation histograms, allowing us to find the boundary curve between them.

3.2.3 Optical flow

Next, let us consider what happens when we have a video sequence, instead of just a single static image. When an object in the video is moving, or when the camera is moving relative to an object, the resulting apparent motion in the image is called optical flow. Optical flow describes the direction and speed of motion of features in the image—the optical flow of a video of a race car would be measured in pixels per second, not miles per hour. The optical flow encodes useful information about scene structure. For example, in a video of scenery taken from a moving train, distant objects have slower apparent motion than close objects; thus, the rate of apparent motion can tell us something about distance. Optical flow also enables us to recognize actions. In Figure 24.10(a) and (b), we show two frames from a video of a tennis player. In (c) we display the optical flow vectors computed from these images, showing that the racket and front leg are moving fastest.

The optical flow vector field can be represented at any point (x, y) by its components $v_x(x, y)$ in the x direction and $v_y(x, y)$ in the y direction. To measure optical flow we need to find corresponding points between one time frame and the next. A simple-minded technique is based on the fact that image patches around corresponding points have similar intensity patterns. Consider a block of pixels centered at pixel p, (x_0, y_0), at time t_0. This block of pixels is to be compared with pixel blocks centered at various candidate pixels at $(x_0 + D_x, y_0 + D_y)$ at time $t_0 + D_t$. One possible measure of similarity is the sum of squared differences (SSD):

$$\mathrm{SSD}(D_x, D_y) = \sum_{(x,y)} (I(x, y, t) - I(x + D_x, y + D_y, t + D_t))^2 \qquad (9)$$

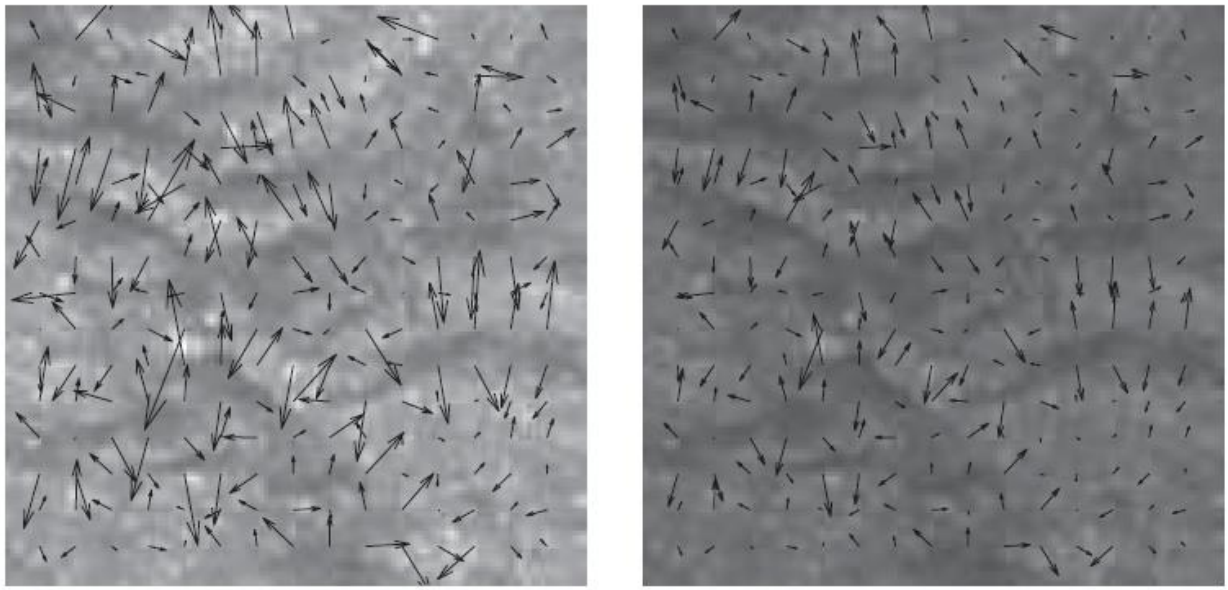


Figure 3.9 Two images of the same texture of crumpled rice paper, with different illumination levels. The gradient vector field (at every eighth pixel) is plotted on top of each one. Notice that, as the light gets darker, all the gradient vectors get shorter. The vectors do not rotate, so the gradient orientations do not change.

Figure 3.10 Two frames of a video sequence. On the right is the optical flow field corresponding to the displacement from one frame to the other. Note how the movement of the tennis racket and the front leg is captured by the directions of the arrows. (Courtesy of Thomas Brox.)

Here, (x, y) ranges over pixels in the block centered at (x_0, y_0). We find the (D_x, D_y) that minimizes the SSD. The optical flow at (x_0, y_0) is then $(v_x, v_y) = (D_x/D_t, D_y/D_t)$. Note that for this to work, there needs to be some texture or variation in the scene. If one is looking at a uniform white wall, then the SSD is going to be nearly the same for the different candidate matches, and the algorithm is reduced to making a blind guess. The best-performing algorithms for measuring optical flow rely on a variety of additional constraints when the scene is only partially textured.

3.2.4 Segmentation of images

Segmentation is the process of breaking an image into regions of similar pixels. Each image pixel can be associated with certain visual properties, such as brightness, color, and texture. Within an object, or a single part of an object, these attributes vary relatively little, whereas across an inter-object boundary there is typically a large change in one or more of these attributes.

There are two approaches to segmentation, one focusing on detecting the boundaries of these regions, and the other on detecting the regions themselves (Figure 24.11). A boundary curve passing through a pixel (x, y) will have an orientation θ, so one way to formalize the problem of detecting boundary curves is as a machine learning classification problem. Based on features from a local neighborhood, we want to compute the probability $P_b(x, y, \theta)$ that indeed there is a boundary curve at that pixel along that orientation. Consider a circular disk centered at (x, y), subdivided into two half disks by a diameter oriented at θ.

If there is a boundary at (x, y, θ) the two half disks might be expected to differ significantly in their brightness, color, and texture. Martin, Fowlkes, and Malik (2004) used features based on differences in histograms of brightness, color, and texture values measured in these two half disks, and then trained a classifier. For this they used a data set of natural images where humans had marked the “ground truth” boundaries, and the goal of the classifier was to mark exactly those boundaries marked by humans and no others.

Boundaries detected by this technique turn out to be significantly better than those found using the simple edge-detection technique described previously. But still there are two limitations.

(1) The boundary pixels formed by thresholding $P_b(x, y, \theta)$ are not guaranteed to form closed curves, so this approach doesn’t deliver regions, and (2) the decision making exploits only local context and does not use global consistency constraints.

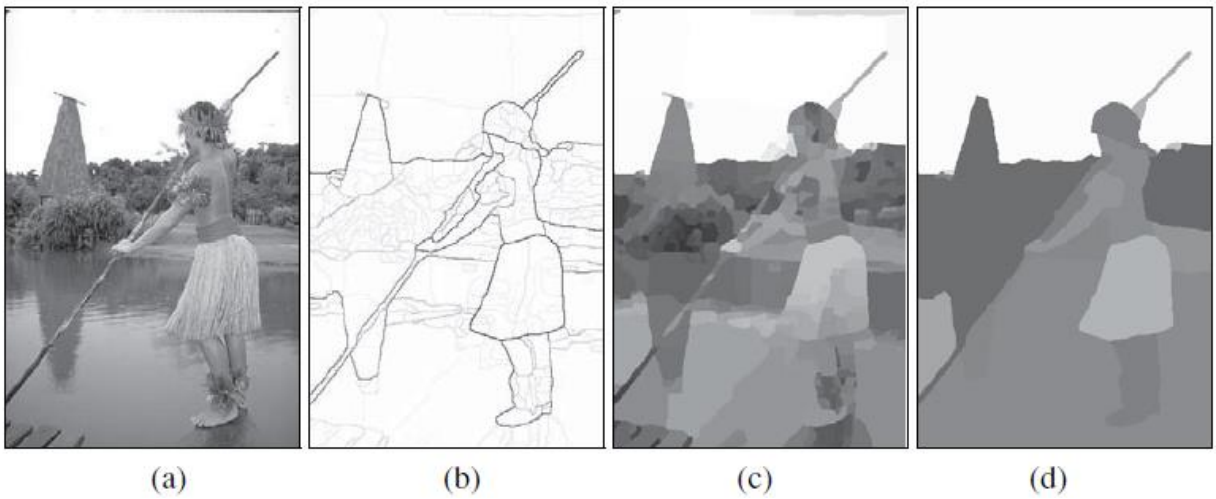


Figure 3.11 (a) Original image. (b) Boundary contours, where the higher the P_b value, the darker the contour. (c) Segmentation into regions, corresponding to a fine partition of the image. Regions are rendered in their mean colors. (d) Segmentation into regions, corresponding to a coarser partition of the

image, resulting in fewer regions. (Courtesy of Pablo Arbelaez, Michael Maire, Charles Fowlkes, and Jitendra Malik)

The alternative approach is based on trying to "cluster" the pixels into regions based on their brightness, color, and texture. Shi and Malik (2000) set this up as a graph partitioning problem. The nodes of the graph correspond to pixels, and edges to connections between pixels. The weight W_{ij} on the edge connecting a pair of pixels i and j is based on how similar the two pixels are in brightness, color, texture, etc. Partitions that minimize a normalized cut criterion are then found. Roughly speaking, the criterion for partitioning the graph is to minimize the sum of weights of connections across the groups of pixels and maximize the sum of weights of connections within the groups.

Segmentation based purely on low-level, local attributes such as brightness and color cannot be expected to deliver the final correct boundaries of all the objects in the scene. To reliably find object boundaries we need high-level knowledge of the likely kinds of objects in the scene. Representing this knowledge is a topic of active research. A popular strategy is to produce an over-segmentation of an image, containing hundreds of homogeneous regions known as superpixels. From there, knowledge-based algorithms can take over; they will find it easier to deal with hundreds of superpixels rather than millions of raw pixels. How to exploit high-level knowledge of objects is the subject of the next section.

3.3 OBJECT RECOGNITION BY APPEARANCE

Appearance is shorthand for what an object tends to look like. Some object categories—for example, baseballs—vary rather little in appearance; all of the objects in the category look about the same under most circumstances. In this case, we can compute a set of features describing each class of images likely to contain the object, then test it with a classifier.

Other object categories—for example, houses or ballet dancers—vary greatly. A house can have different size, color, and shape and can look different from different angles. A dancer looks different in each pose, or when the stage lights change colors. A useful abstraction is to say that some objects are made up of local patterns which tend to move around with respect to one another. We can then find the object by looking at local histograms of detector responses, which expose whether some part is present but suppress the details of where it is.

Testing each class of images with a learned classifier is an important general recipe. It works extremely well for faces looking directly at the camera, because at low resolution and under reasonable lighting, all such faces look quite similar. The face is round, and quite bright compared to the eye sockets; these are dark, because they are sunken, and the mouth is a dark slash, as are the eyebrows. Major changes of illumination can cause some variations in this pattern, but the range of variation is quite manageable. That makes it possible to detect face positions in an image that contains faces. Once a computational challenge, this feature is now commonplace in even inexpensive digital cameras.

For the moment, we will consider only faces where the nose is oriented vertically; we will deal with rotated faces below. We sweep a round window of fixed size over the image, compute features for it, and present the features to a classifier. This strategy is sometimes called the sliding window. Features need to be robust to shadows and to changes in brightness caused by illumination changes. One strategy is to build features out of gradient orientations.

Another is to estimate and correct the illumination in each image window. To find faces of different sizes, repeat the sweep over larger or smaller versions of the image. Finally, we postprocess the responses across scales and locations to produce the final set of detections. Postprocessing is important, because it is unlikely that we have chosen a window size that is exactly the right size for a face (even if we use multiple sizes). Thus, we will likely have several overlapping windows that each report a match for a face. However, if we use a classifier that can report strength of response (for example, logistic regression or a support vector machine) we can combine these partial overlapping matches at nearby locations to yield a single high-quality match. That gives us a face detector that can search over locations and scales. To search rotations as well, we use two steps. We train a regression procedure to estimate the best orientation of any face present in a window. Now, for each window, we estimate the orientation, reorient the window, then test whether a vertical face is present with our classifier. All this yields a system whose architecture is sketched in Figure 3.12.

Training data is quite easily obtained. There are several data sets of marked-up face images, and rotated face windows are easy to build (just rotate a window from a training data set). One trick that is widely used is to take each example window, then produce new examples by changing the orientation of the window, the center of the window, or the scale very slightly. This is an easy way of getting a bigger data set that reflects real images fairly well; the trick usually improves performance significantly. Face detectors built along these lines now perform very well for frontal faces (side views are harder).

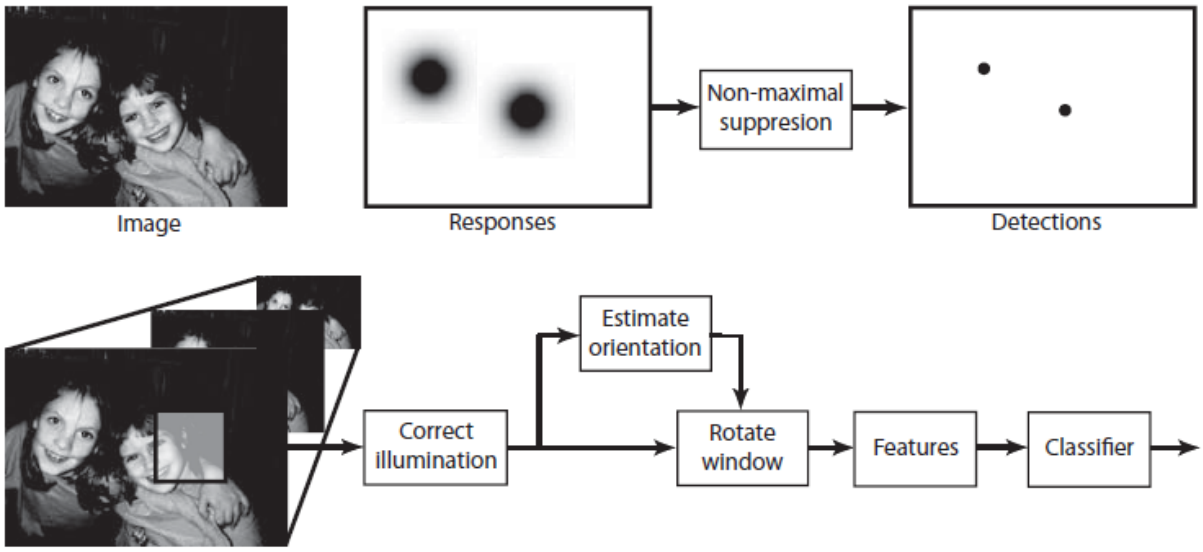


Figure 3.12 Face finding systems vary, but most follow the architecture illustrated in two parts here. On the top, we go from images to responses, then apply non-maximum suppression to find the strongest local response. The responses are obtained by the process illustrated on the bottom. We sweep a window of fixed size over larger and smaller versions of the image, so as to find smaller or larger faces, respectively. The illumination in the window is corrected, and then a regression engine (quite often, a neural net) predicts the orientation of the face. The window is corrected to this orientation and then presented to a classifier. Classifier outputs are then postprocessed to ensure that only one face is placed at each location in the image.

3.3.1 Complex appearance and pattern elements

Many objects produce much more complex patterns than faces do. This is because several effects can move features around in an image of the object. Effects include (Figure 3.13)

- Foreshortening, which causes a pattern viewed at a slant to be significantly distorted.
- Aspect, which causes objects to look different when seen from different directions. Even as simple an object as a doughnut has several aspects; seen from the side, it looks like a flattened oval, but from above it is an annulus.
- Occlusion, where some parts are hidden from some viewing directions. Objects can occlude one another, or parts of an object can occlude other parts, an effect known as self-occlusion.
- Deformation, where internal degrees of freedom of the object change its appearance. For example, people can move their arms and legs around, generating a very wide range of different body configurations.

However, our recipe of searching across location and scale can still work. This is because some structure will be present in the images produced by the object. For example, a picture of a car is likely to show some of headlights, doors, wheels, windows, and hubcaps, though they may be in somewhat different arrangements in different pictures. This suggests modeling objects with pattern elements—collections of parts. These pattern elements may move around with respect to one another, but if most of the pattern elements are present in about the right place, then the object is present. An object recognizer is then a collection of features that can tell whether the pattern elements are present, and whether they are in about the right place.

The most obvious approach is to represent the image window with a histogram of the pattern elements that appear there. This approach does not work particularly well, because too many patterns get confused with one another. For example, if the pattern elements are color pixels, the French, UK, and Netherlands flags will get confused because they have approximately the same color histograms, though the colors are arranged in very different ways. Quite simple modifications of histograms yield very useful features. The trick is to preserve some spatial detail in the representation; for example, headlights tend to be at the front of a car and wheels tend to be at the bottom. Histogram-based features have been successful in a wide variety of recognition applications; we will survey pedestrian detection.

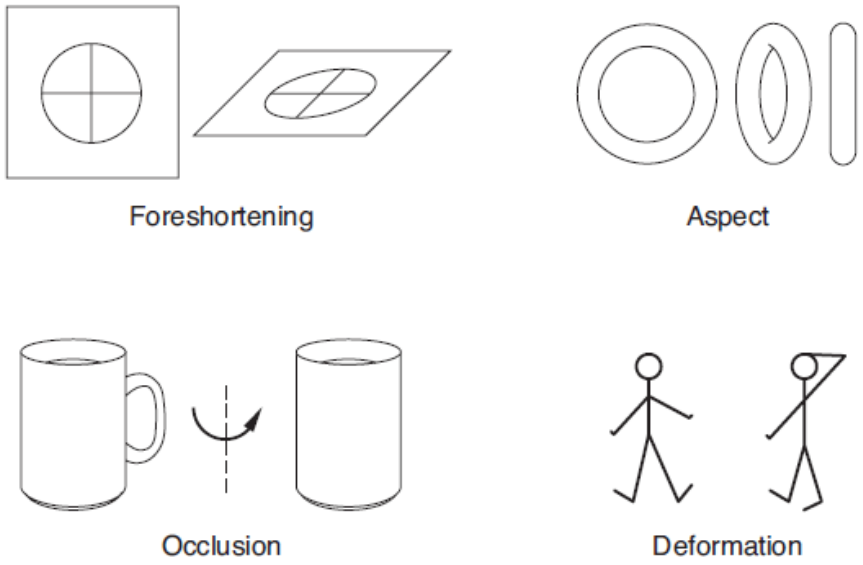


Figure 3.13 Sources of appearance variation. First, elements can foreshorten, like the circular patch on the top left. This patch is viewed at a slant, and so is elliptical in the image. Second, objects viewed from different directions can change shape quite dramatically, a phenomenon known as aspect. On the top right are three different aspects of a doughnut. Occlusion causes the handle of the mug on the bottom left to disappear when the mug is rotated. In this case, because the body and handle belong to the same mug, we have self-occlusion. Finally, on the bottom right, some objects can deform dramatically.

3.3.2 Pedestrian detection with HOG features

The World Bank estimates that each year car accidents kill about 1.2 million people, of whom about two thirds are pedestrians. This means that detecting pedestrians is an important application problem, because cars that can automatically detect and avoid pedestrians might save many lives. Pedestrians wear many different kinds of clothing and appear in many different configurations, but, at relatively low resolution, pedestrians can have a fairly characteristic appearance. The most usual cases are lateral or frontal views of a walk. In these cases, we see either a “lollipop” shape — the torso is wider than the legs, which are together in the stance phase of the walk — or a “scissor” shape — where the legs are swinging in the walk. We expect to see some evidence of arms and legs, and the curve around the shoulders and head also tends to visible and quite distinctive. This means that, with a careful feature construction, we can build a useful moving-window pedestrian detector.

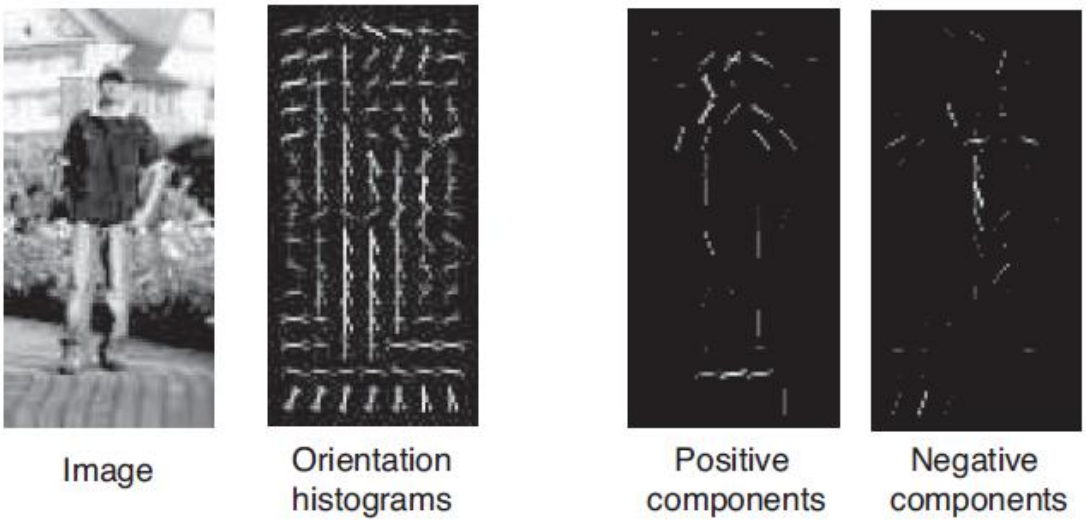


Figure 3.14 Local orientation histograms are a powerful feature for recognizing even quite complex objects. On the left, an image of a pedestrian. On the center left, local orientation histograms for patches. We then apply a classifier such as a support vector machine to find the weights for each histogram that best separate the positive examples of pedestrians from non-pedestrians. We see that the positively weighted components look like the outline of a person. The negative components are less clear; they represent all the patterns that are not pedestrians. Figure from Dalal and Triggs (2005) cIEEE.

There isn't always a strong contrast between the pedestrian and the background, so it is better to use orientations than edges to represent the image window. Pedestrians can move their arms and legs around, so we should use a histogram to suppress some spatial detail in the feature. We break up the window into cells, which could overlap, and build an orientation histogram in each cell. Doing so will produce a feature that can tell whether the head-and shoulders curve is at the top of the window or at the bottom, but will not change if the head moves slightly.

One further trick is required to make a good feature. Because orientation features are not affected by illumination brightness, we cannot treat high-contrast edges specially. This means that the distinctive curves on the boundary of a pedestrian are treated in the same way as fine texture detail in clothing or in the background, and so the signal may be submerged in noise. We can recover contrast information by counting gradient orientations with weights that reflect how significant a gradient is compared to other gradients in the same cell. We will write || ∇Ix || for the gradient magnitude at point x in the image, write C for the cell whose histogram we wish to compute, and write w_x,C for the weight that we will use for the orientation at x for this cell. A natural choice of weight is

$$w_X, C = \frac{||\nabla \mathbf{I_x}||}{\sum_{\mathbf{u}\in C} ||\nabla \mathbf{I_u}||} \qquad (10)$$

Figure 3.15 Another example of object recognition, this one using the SIFT feature (Scale Invariant Feature Transform), an earlier version of the HOG feature. On the left, images of a shoe and a telephone that serve as object models. In the center, a test image. On the right, the shoe and the telephone have been detected by: finding points in the image whose SIFT feature descriptions match a model; computing an estimate of pose of the model; and verifying that estimate. A strong match is usually verified with rare false positives. Images from Lowe (1999) cIEEE.

This compares the gradient magnitude to others in the cell, so gradients that are large compared to their neighbors get a large weight. The resulting feature is usually called a HOG feature (for Histogram Of Gradient orientations). This feature construction is the main way in which pedestrian detection differs from face detection. Otherwise, building a pedestrian detector is very like building a face detector. The detector sweeps a window across the image, computes features for that window, then presents it to a classifier. Non-maximum suppression needs to be applied to the output. In most applications, the scale and orientation of typical pedestrians is known. For example, in driving applications in which a camera is fixed to the car, we expect to view mainly vertical pedestrians, and we are interested only in nearby pedestrians. Several pedestrian data sets have been published, and these can be used for training the classifier. Pedestrians are not the only type of object we can detect. In Figure 24.15 we see that similar techniques can be used to find a variety of objects in different contexts.

3.4 RECONSTRUCTING THE 3D WORLD

Given that all points in the scene that fall along a ray to the pinhole of a pinhole camera are projected to the same point in the image, how do we recover three-dimensional information? Two ideas come to our rescue:

- If we have two (or more) images from different camera positions, then we can triangulate to find the position of a point in the scene.
- We can exploit background knowledge about the physical scene that gave rise to the image. Given an object model P(Scene) and a rendering model P(Image | Scene), we can compute a posterior distribution P(Scene | Image).

There is as yet no single unified theory for scene reconstruction. We survey eight commonly used visual cues: motion, binocular stereopsis, multiple views, texture, shading, contour, and familiar objects. In this

section we show how to go from the two-dimensional image to a three-dimensional representation of the scene.

3.4.1 Motion parallax

If the camera moves relative to the three-dimensional scene, the resulting apparent motion in the image, optical flow, can be a source of information for both the movement of the camera and depth in the scene. To understand this, we state (without proof) an equation that relates the optical flow to the viewer's translational velocity T and the depth in the scene.

The components of the optical flow field are

$$v_x(x,y) = \frac{-T_x + xT_z}{Z(x,y)}, \qquad v_y(x,y) = \frac{-T_y + yT_z}{Z(x,y)}, \tag{11}$$

where Z(x, y) is the z-coordinate of the point in the scene corresponding to the point in the image at (x, y).

Note that both components of the optical flow, $v_x(x, y)$ and $v_y(x, y)$, are zero at the point $x = T_x/T_z$, $y = T_y/T_z$. This point is called the focus of expansion of the flow field. Suppose we change the origin in the x–y plane to lie at the focus of expansion; then the expressions for optical flow take on a particularly simple form. Let (x',y') be the new coordinates defined by $x' = x - T_x/T_z$, $y' = y - T_y/T_z$. Then

$$v_x(x',y') = \frac{x'T_z}{Z(x',y')}, \qquad v_y(x',y') = \frac{y'T_z}{Z(x',y')} \tag{12}$$

Note that there is a scale-factor ambiguity here. If the camera was moving twice as fast, and every object in the scene was twice as big and at twice the distance to the camera, the optical flow field would be exactly the same. But we can still extract quite useful information.

1. Suppose you are a fly trying to land on a wall and you want to know the time-to contact at the current velocity. This time is given by Z/T_z. Note that although the instantaneous optical flow field cannot provide either the distance Z or the velocity component T_z, it can provide the ratio of the two and can therefore be used to control the landing approach. There is considerable experimental evidence that many different animal species exploit this cue.
2. Consider two points at depths Z_1, Z_2, respectively. We may not know the absolute value of either of these, but by considering the inverse of the ratio of the optical flow magnitudes at these points, we can determine the depth ratio Z_1/Z_2. This is the cue of motion parallax, one we use when we look out of the side window of a moving car or train and infer that the slower moving parts of the landscape are farther away.

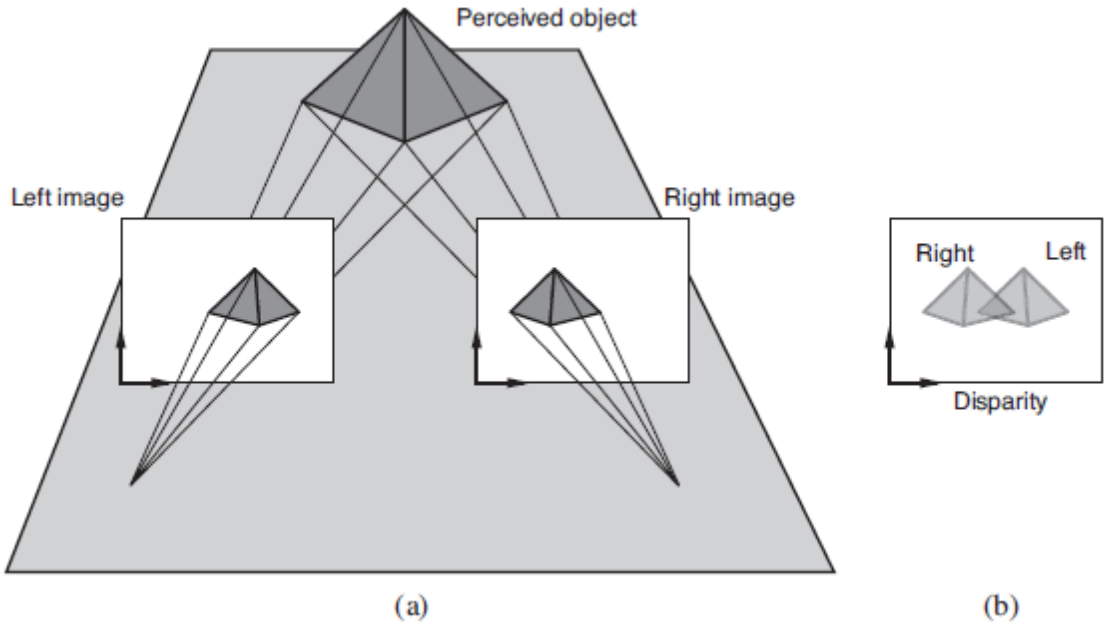


Figure 3.16 Translating a camera parallel to the image plane causes image features to move in the camera plane. The disparity in positions that results is a cue to depth. If we superimpose left and right image, as in (b), we see the disparity.

3.4.2 Binocular stereopsis

Most vertebrates have two eyes. This is useful for redundancy in case of a lost eye, but it helps in other ways too. Most prey have eyes on the side of the head to enable a wider field of vision. Predators have the eyes in the front, enabling them to use binocular stereopsis.

The idea is similar to motion parallax, except that instead of using images over time, we use two (or more) images separated in space. Because a given feature in the scene will be in a different place relative to the z-axis of each image plane, if we superpose the two images, there will be a disparity in the location of the image feature in the two images. You can see this in Figure 3.16, where the nearest point of the pyramid is shifted to the left in the right image and to the right in the left image.

Note that to measure disparity we need to solve the correspondence problem, that is, determine for a point in the left image, the point in the right image that results from the projection of the same scene point. This is analogous to what one has to do in measuring optical flow, and the most simple-minded approaches are somewhat similar and based on comparing blocks of pixels around corresponding points using the sum of squared differences.

In practice, we use much more sophisticated algorithms, which exploit additional constraints. Assuming that we can measure disparity, how does this yield information about depth in the scene? We will need to work out the geometrical relationship between disparity and depth. First, we will consider the case when both the eyes (or cameras) are looking forward with their optical axes parallel. The relationship of the right camera to the left camera is then just a displacement along the x-axis by an amount b, the baseline. We can use the optical flow equations from the previous section, if we think of this as resulting

from a translation vector T acting for time δt, with $T_x = b/\delta t$ and $T_y = T_z = 0$. The horizontal and vertical disparity are given by the optical flow components, multiplied by the time step δt, $H = v_x \delta t$, $V = v_y \delta t$. Carrying out the substitutions, we get the result that H = b/Z, V = 0. In words, the horizontal disparity is equal to the ratio of the baseline to the depth, and the vertical disparity is zero. Given that we know b, we can measure H and recover the depth Z.

Under normal viewing conditions, humans fixate; that is, there is some point in the scene at which the optical axes of the two eyes intersect. Figure 3.3 shows two eyes fixated at a point P0, which is at a distance Z from the midpoint of the eyes. For convenience, we will compute the angular disparity, measured in radians. The disparity at the point of fixation P0 is zero. For some other point P in the scene that is δZ farther away, we can compute the angular displacements of the left and right images of P, which we will call P_L and P_R, respectively. If each of these is displaced by an angle δθ/2 relative to P_0, then the displacement between PL and PR, which is the disparity of P, is just δθ. From Figure 3.17, $\tan\theta = \frac{b/2}{Z}$ *and* $\tan\left(\theta - \frac{\delta\theta}{2}\right) = \frac{b/2}{Z+\delta Z}$, but for small angles, $\tan\theta \approx \theta$, so

$$\delta\theta/2 = \frac{b/2}{Z} - \frac{b/2}{Z+\delta Z} \approx \frac{b\delta Z}{2Z^2} \qquad (13)$$

and, since the actual disparity is δθ, we have

$$disparity = \frac{b\delta Z}{Z^2}.$$

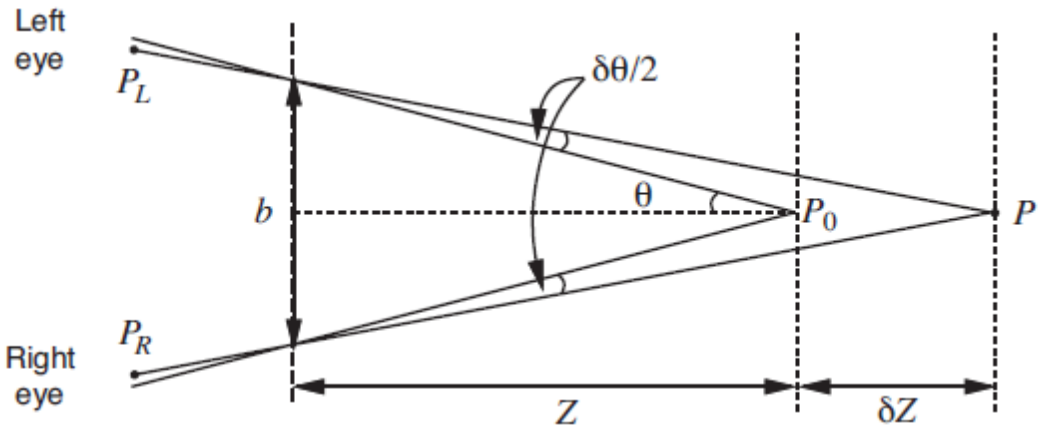


Figure 3.17 The relation between disparity and depth in stereopsis. The centers of projection of the two eyes are b apart, and the optical axes intersect at the fixation point P_0. The point P in the scene projects to points P_L and P_R in the two eyes. In angular terms, the disparity between these is δϑ. See text.

In humans, b (the baseline distance between the eyes) is about 6 cm. Suppose that Z is about 100 cm. If the smallest detectable δθ (corresponding to the pixel size) is about 5 seconds of arc, this gives a δZ of 0.4 mm. For Z = 30 cm, we get the impressively small value δZ = 0.036 mm. That is, at a distance of 30

cm, humans can discriminate depths that differ by as little as 0.036 mm, enabling us to thread needles and the like.

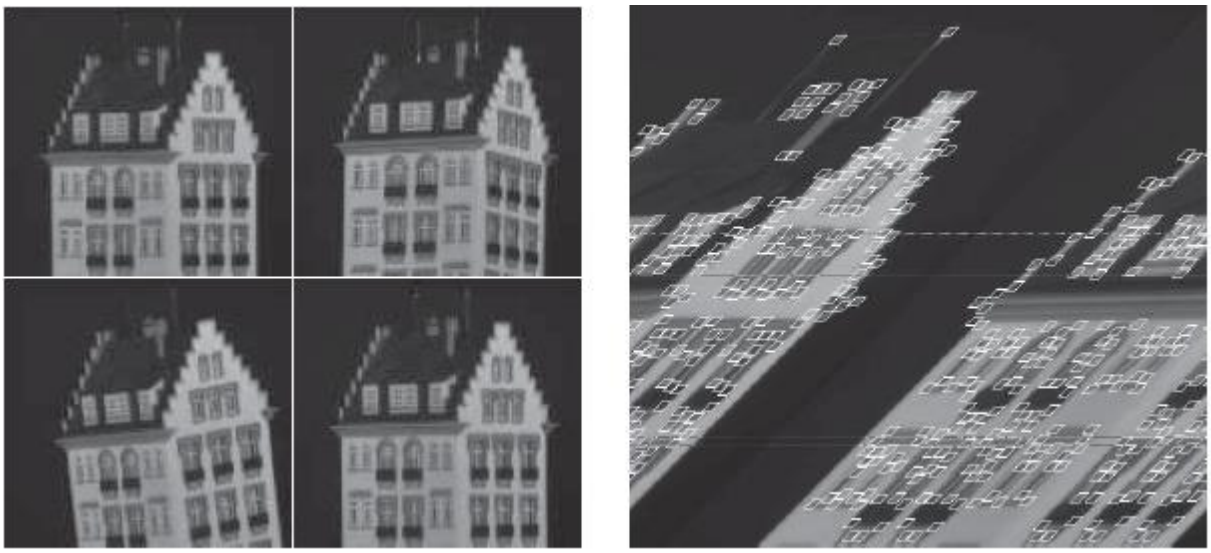


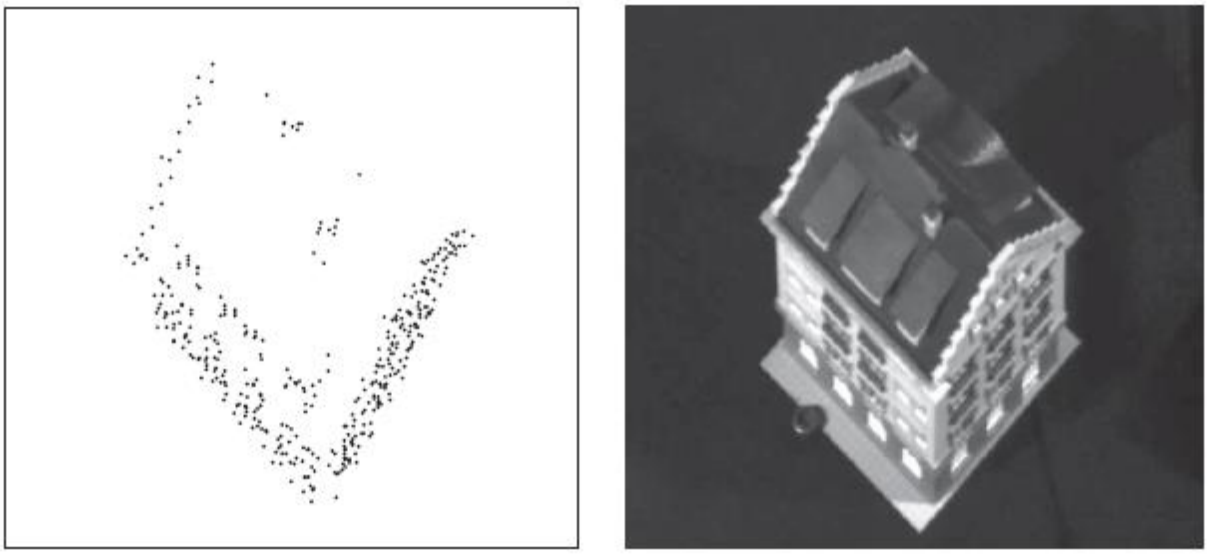
Figure 3.18 (a) Four frames from a video sequence in which the camera is moved and rotated relative to the object. (b) The first frame of the sequence, annotated with small boxes highlighting the features found by the feature detector. (Courtesy of Carlo Tomasi.)

3.4.3 Multiple views

Shape from optical flow or binocular disparity are two instances of a more general framework, that of exploiting multiple views for recovering depth. In computer vision, there is no reason for us to be restricted to differential motion or to only use two cameras converging at a fixation point. Therefore, techniques have been developed that exploit the information available in multiple views, even from hundreds or thousands of cameras. Algorithmically, there are three subproblems that need to be solved:

- The correspondence problem, i.e., identifying features in the different images that are projections of the same feature in the three-dimensional world.
- The relative orientation problem, i.e., determining the transformation (rotation and translation) between the coordinate systems fixed to the different cameras.
- The depth estimation problem, i.e., determining the depths of various points in the world for which image plane projections were available in at least two views

The development of robust matching procedures for the correspondence problem, accompanied by numerically stable algorithms for solving for relative orientations and scene depth, is one of the success stories of computer vision. Results from one such approach due to Tomasi and Kanade (1992) are shown in Figures 3.18 and 3.19.



(a) (b)

Figure 3.19 (a) Three-dimensional reconstruction of the locations of the image features in Figure 3.4, shown from above. (b) The real house, taken from the same position.

3.4.4 Texture

Earlier we saw how texture was used for segmenting objects. It can also be used to estimate distances. In Figure 3.6 we see that a homogeneous texture in the scene results in varying texture elements, or texels, in the image. All the paving tiles in (a) are identical in the scene.

They appear different in the image for two reasons:

1. Differences in the distances of the texels from the camera. Distant objects appear smaller by a scaling factor of 1/Z.
2. Differences in the foreshortening of the texels. If all the texels are in the ground plane then distance ones are viewed at an angle that is farther off the perpendicular, and so are more foreshortened. The magnitude of the foreshortening effect is proportional to cos σ, where σ is the slant, the angle between the Z-axis and n, the surface normal to the texel.

Researchers have developed various algorithms that try to exploit the variation in the appearance of the projected texels as a basis for determining surface normals. However, the accuracy and applicability of these algorithms is not anywhere as general as those based on using multiple views.

Figure 3.20 (a) A textured scene. Assuming that the real texture is uniform allows recovery of the surface orientation. The computed surface orientation is indicated by overlaying a black circle and pointer, transformed as if the circle were painted on the surface at that point. (b) Recovery of shape from texture for a curved surface (white circle and pointer this time). Images courtesy of Jitendra Malik and Ruth Rosenholtz (1994).

3.4.5 Shading

Shading—variation in the intensity of light received from different portions of a surface in a scene—is determined by the geometry of the scene and by the reflectance properties of the surfaces. In computer graphics, the objective is to compute the image brightness I(x, y), given the scene geometry and reflectance properties of the objects in the scene. Computer vision aims to invert the process—that is, to recover the geometry and reflectance properties, given the image brightness I(x, y). This has proved to be difficult to do in anything but the simplest cases.

From previous knowledge, we know that if a surface normal points toward the light source, the surface is brighter, and if it points away, the surface is darker. We cannot conclude that a dark patch has its normal pointing away from the light; instead, it could have low albedo. Generally, albedo changes quite quickly in images, and shading changes rather slowly, and humans seem to be quite good at using this observation to tell whether low illumination, surface orientation, or albedo caused a surface patch to be dark.

To simplify the problem, let us assume that the albedo is known at every surface point. It is still difficult to recover the normal, because the image brightness is one measurement but the normal has two unknown parameters, so we cannot simply solve for the normal. The key to this situation seems to be that nearby normals will be similar, because most surfaces are smooth—they do not have sharp changes.

The real difficulty comes in dealing with interreflections. If we consider a typical indoor scene, such as the objects inside an office, surfaces are illuminated not only by the light sources, but also by the light reflected from other surfaces in the scene that effectively serve as secondary light sources. These mutual illumination effects are quite significant and make it quite difficult to predict the relationship

between the normal and the image brightness. Two surface patches with the same normal might have quite different brightnesses, because one receives light reflected from a large white wall and the other faces only a dark bookcase.

Despite these difficulties, the problem is important. Humans seem to be able to ignore the effects of interreflections and get a useful perception of shape from shading, but we know frustratingly little about algorithms to do this.

3.4.6 Contour

When we look at a line drawing, such as Figure 3.7, we get a vivid perception of three-dimensional shape and layout. How? It is a combination of recognition of familiar objects in the scene and the application of generic constraints such as the following:

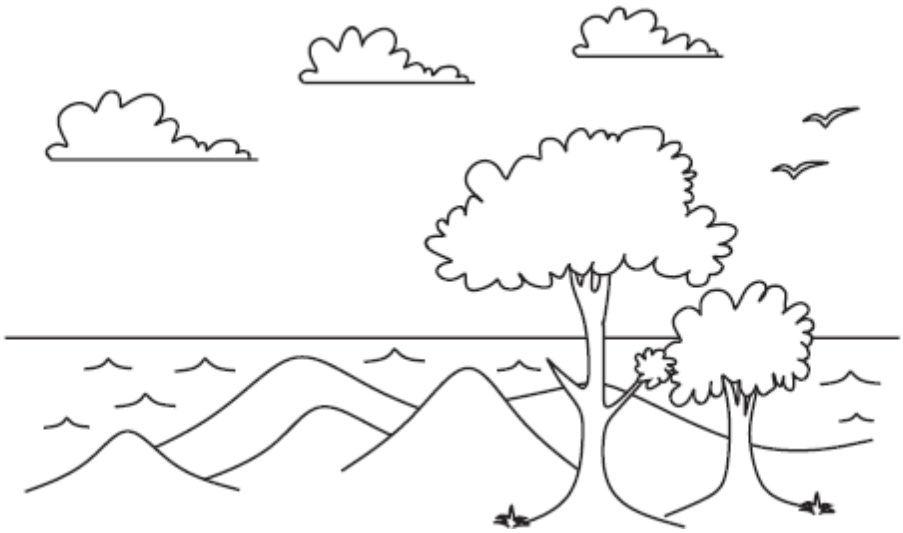


Figure 3.21 An evocative line drawing. (Courtesy of Isha Malik.)

• Occluding contours, such as the outlines of the hills. One side of the contour is nearer to the viewer, the other side is farther away. Features such as local convexity and symmetry provide cues to solving the figure-ground problem—assigning which side of the contour is figure (nearer), and which is ground (farther). At an occluding contour, the line of sight is tangential to the surface in the scene.

- T-junctions. When one object occludes another, the contour of the farther object is interrupted, assuming that the nearer object is opaque. A T-junction results in the image.
- Position on the ground plane. Humans, like many other terrestrial animals, are very often in a scene that contains a ground plane, with various objects at different locations on this plane. Because of gravity, typical objects don't float in air but are supported by this ground plane, and we can exploit the very special geometry of this viewing scenario.

Let us work out the projection of objects of different heights and at different locations on the ground plane. Suppose that the eye, or camera, is at a height h_c above the ground plane. Consider an object of height δY resting on the ground plane, whose bottom is at $(X, -h_c, Z)$ and top is at $(X, \delta Y - h_c, Z)$. The bottom projects to the image point $(f_X/Z, -fh_c/Z)$ and the top to $(f_X/Z, f(\delta Y - h_c)/Z)$. The bottoms of nearer

objects (small Z) project to points lower in the image plane; farther objects have bottoms closer to the horizon.

3.4.7 Objects and the geometric structure of scenes

A typical adult human head is about 9 inches long. This means that for someone who is 43 feet away, the angle subtended by the head at the camera is 1 degree. If we see a person whose head appears to subtend just half a degree, Bayesian inference suggests we are looking at a normal person who is 86 feet away, rather than someone with a half-size head. This line of reasoning supplies us with a method to check the results of a pedestrian detector, as well as a method to estimate the distance to an object. For example, all pedestrians are about the same height, and they tend to stand on a ground plane. If we know where the horizon is in an image, we can rank pedestrians by distance to the camera. This works because we know where their feet are, and pedestrians whose feet are closer to the horizon in the image are farther away from the camera (Figure 3.8). Pedestrians who are farther away from the camera must also be smaller in the image. This means we can rule out some detector responses — if a detector finds a pedestrian who is large in the image and whose feet are close to the horizon, it has found an enormous pedestrian; these don't exist, so the detector is wrong. In fact, many or most image windows are not acceptable pedestrian windows, and need not even be presented to the detector.

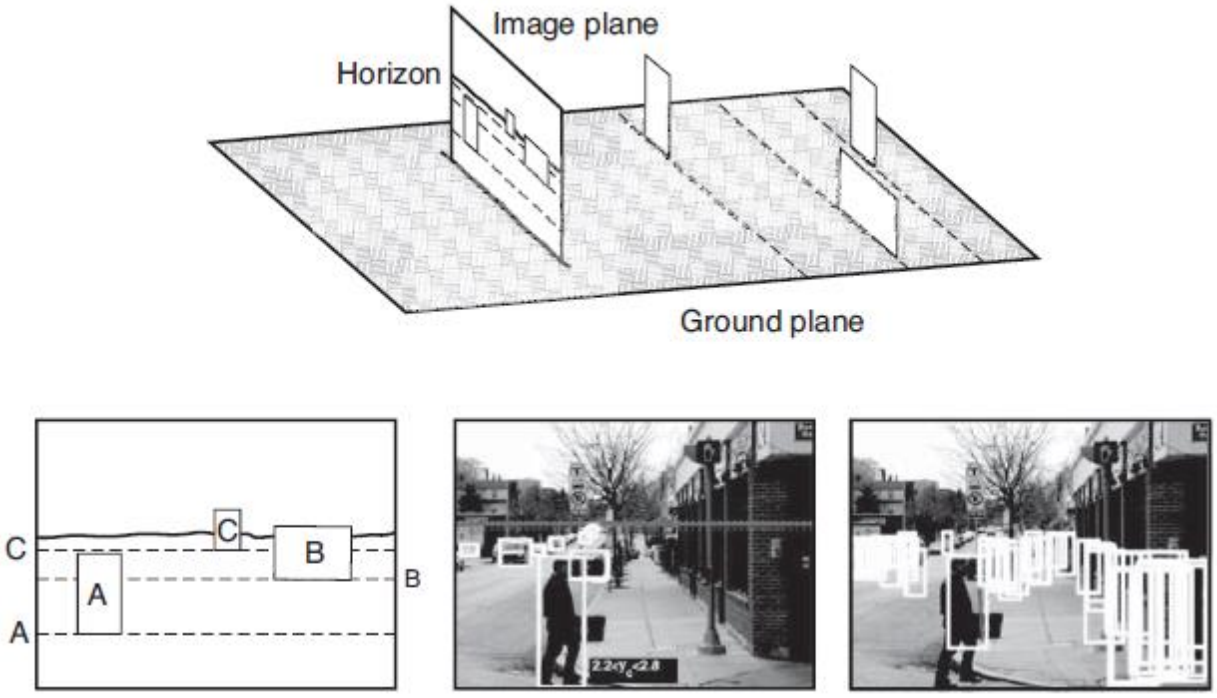


Figure 3.22 In an image of people standing on a ground plane, the people whose feet are closer to the horizon in the image must be farther away (top drawing). This means they must look smaller in the image (left lower drawing). This means that the size and location of real pedestrians in an image depend upon one another and on the location of the horizon. To exploit this, we need to identify the ground plane, which is done using shape-from-texture methods. From this information, and from some likely pedestrians, we can recover a horizon as shown in the center image. On the right, acceptable pedestrian boxes given this geometric context. Notice that pedestrians who are higher in the scene must be smaller. If they are not, then they are false positives. Images from Hoiem et al. (2008) cIEEE.

There are several strategies for finding the horizon, including searching for a roughly horizontal line with a lot of blue above it, and using surface orientation estimates obtained from texture deformation. A more elegant strategy exploits the reverse of our geometric constraints. A reasonably reliable pedestrian detector is capable of producing estimates of the horizon, if there are several pedestrians in the scene at different distances from the camera. This is because the relative scaling of the pedestrians is a cue to where the horizon is. So, we can extract a horizon estimate from the detector, then use this estimate to prune the pedestrian detector's mistakes.

If the object is familiar, we can estimate more than just the distance to it, because what it looks like in the image depends very strongly on its pose, i.e., its position and orientation with respect to the viewer. This has many applications. For instance, in an industrial manipulation task, the robot arm cannot pick up an object until the pose is known. In the case of rigid objects, whether three-dimensional or two-dimensional, this problem has a simple and well-defined solution based on the alignment method, which we now develop.

The object is represented by M features or distinguished points $m_1, m_2, \ldots, m_M$ in three-dimensional space—perhaps the vertices of a polyhedral object. These are measured in some coordinate system that is natural for the object. The points are then subjected to an unknown three-dimensional rotation R, followed by translation by an unknown amount t and then projection to give rise to image feature points $p_1, p_2, \ldots, p_N$ on the image plane.

In general, $N \neq M$, because some model points may be occluded, and the feature detector could miss some features (or invent false ones due to noise). We can express this as

$$p_i = \Pi(Rm_i + t) = Q(m_i) \qquad (14)$$

for a three-dimensional model point mi and the corresponding image point p_i. Here, R is a rotation matrix, t is a translation, and Π denotes perspective projection or one of its approximations, such as scaled orthographic projection. The net result is a transformation Q that will bring the model point mi into alignment with the image point pi. Although we do not know Q initially, we do know (for rigid objects) that Q must be the same for all the model points.

We can solve for Q, given the three-dimensional coordinates of three model points and their two-dimensional projections. The intuition is as follows: we can write down equations relating the coordinates of pi to those of mi. In these equations, the unknown quantities correspond to the parameters of the rotation matrix R and the translation vector t. If we have enough equations, we ought to be able to solve for Q. We will not give a proof here; we merely state the following result:

Given three noncollinear points m_1, m_2, and m_3 in the model, and their scaled orthographic projections p_1, p_2, and p_3 on the image plane, there exist exactly two transformations from the three-dimensional model coordinate frame to a two-dimensional image coordinate frame.

These transformations are related by a reflection around the image plane and can be computed by a simple closed-form solution. If we could identify the corresponding model features for three features in the image, we could compute Q, the pose of the object.

Let us specify position and orientation in mathematical terms. The position of a point P in the scene is characterized by three numbers, the (X, Y, Z) coordinates of P in a coordinate frame with its origin at the pinhole and the Z-axis along the optical axis (Figure 3.9). What we have available is the perspective projection (x, y) of the point in the image. This specifies the ray from the pinhole along which P lies; what we do not know is the distance. The term "orientation" could be used in two senses:

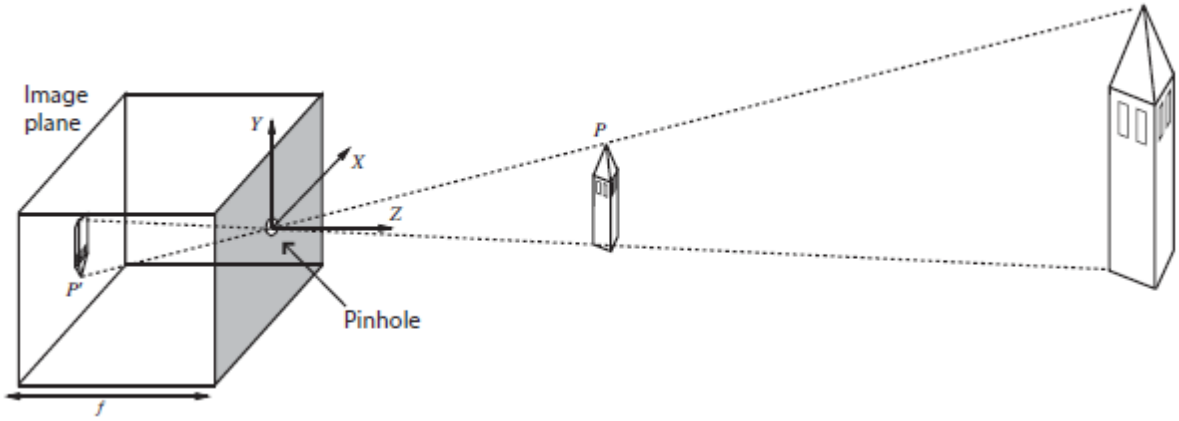


Figure 3.23 Each light-sensitive element in the image plane at the back of a pinhole camera receives light from the small range of directions that passes through the pinhole. If the pinhole is small enough, the result is a focused image at the back of the pinhole. The process of projection means that large, distant objects look the same as smaller, nearby objects. Note that the image is projected upside down.

1. **The orientation of the object as a whole**. This can be specified in terms of a three-dimensional rotation relating its coordinate frame to that of the camera.
2. The orientation of the surface of the object at P. This can be specified by a normal vector, n—which is a vector specifying the direction that is perpendicular to the surface.

Often, we express the surface orientation using the variables slant and tilt. Slant is the angle between the Z-axis and n. Tilt is the angle between the X-axis and the projection of n on the image plane.

When the camera moves relative to an object, both the object's distance and its orientation change. What is preserved is the shape of the object. If the object is a cube, that fact is not changed when the object moves. Geometers have been attempting to formalize shape for centuries, the basic concept being that shape is what remains unchanged under some group of transformations—for example, combinations of rotations and translations. The difficulty lies in finding a representation of global shape that is general enough to deal with the wide variety of objects in the real world—not just simple forms like cylinders, cones, and spheres—and yet can be recovered easily from the visual input. The problem of characterizing the local shape of a surface is much better understood. Essentially, this can be done in terms of curvature: how does the surface normal change as one moves in different directions on the surface? For a plane, there is no change at all. For a cylinder, if one moves parallel to the axis, there is no change, but in the perpendicular direction, the surface normal rotates at a rate inversely proportional to the radius of the cylinder, and so on. All this is studied in the subject called differential geometry.

The shape of an object is relevant for some manipulation tasks (e.g., deciding where to grasp an object), but its most significant role is in object recognition, where geometric shape along with color and texture

provide the most significant cues to enable us to identify objects, classify what is in the image as an example of some class one has seen before, and so on.

There are various cues in the image that enable one to obtain three-dimensional information about the scene: motion, stereopsis, texture, shading, and contour analysis. Each of these cues relies on background assumptions about physical scenes to provide nearly unambiguous interpretations.

4.0 Conclusion

Although perception appears to be an effortless activity for humans, it requires a significant amount of sophisticated computation. The goal of vision is to extract information needed for tasks such as manipulation, navigation, and object recognition.

- The process of image formation is well understood in its geometric and physical aspects. Given a description of a three-dimensional scene, we can easily produce a picture of it from some arbitrary camera position (the graphics problem). Inverting the process by going from an image to a description of the scene is more difficult.
- To extract the visual information necessary for the tasks of manipulation, navigation, and recognition, intermediate representations have to be constructed. Early vision image-processing algorithms extract primitive features from the image, such as edges and regions.
- There are various cues in the image that enable one to obtain three-dimensional information about the scene: motion, stereopsis, texture, shading, and contour analysis. Each of these cues relies on background assumptions about physical scenes to provide nearly unambiguous interpretations.

5.0 Summary

We hope you enjoyed this unit. This unit scene analysis in from one dimension to three-dimension scenarios. Now, let us attempt the questions below.

6.0 Tutor Marked Assignment

i. In the shadow of a tree with a dense, leafy canopy, one sees a number of light spots. Surprisingly, they all appear to be circular. Why? After all, the gaps between the leaves through which the sun shines are not likely to be circular.

ii. Consider an infinitely long cylinder of radius r oriented with its axis along the y-axis. The cylinder has a Lambertian surface and is viewed by a camera along the positive z-axis. What will you expect to see in the image if the cylinder is illuminated by a point source at infinity located on the positive x-axis? Draw the contours of constant brightness in the projected image. Are the contours of equal brightness uniformly spaced?

7.0 References/Further Readings

1) Russell, S., & Norvig, P. (2005). AI a modern approach. Learning, 2(3), 4.
2) Rich E., & Knight K. (1991) Artificial Intelligence, McGraw-Hill, New York, NY
3) Shi, Z. (2011). Advanced artificial intelligence (Vol. 1). World Scientific.
4) Malik, J. and Rosenholtz, R. (1997). Computing local surface orientation and shape from texture for curved surfaces. IJCV, 23(2), 149–168.
5) Shi, J. and Malik, J. (2000). Normalized cuts and image segmentation. PAMI, 22(8), 888–905.

6) Dalal, N. and Triggs, B. (2005). Histograms of oriented gradients for human detection. In CVPR, pp. 886–893.
7) Lowe, D. (1999). Object recognition using local scale invariant feature. In ICCV.
8) Tomasi, C. and Kanade, T. (1992). Shape and motion from image streams under orthography: A factorization method. IJCV, 9, 137–154.
9) Malik, J. and Rosenholtz, R. (1994). Recovering surface curvature and orientation from texture distortion: A least squares algorithm and sensitivity analysis. In ECCV, pp. 353–364.
10) Hoiem, D., Efros, A. A., and Hebert, M. (2008). Putting objects in perspective. IJCV, 80(1).

Unit Three: Expert Systems,

CONTENT

1.0 Introduction

An Expert System is defined as an interactive and reliable computer-based decision-making system which uses both facts and heuristics to solve complex decision-making problems. It is considered at the highest level of human intelligence and expertise. It is a computer application which solves the most complex issues in a specific domain.

The expert system can resolve many issues which generally would require a human expert. It is based on knowledge acquired from an expert. It is also capable of expressing and reasoning about some domain of knowledge. Expert systems were the predecessor of the current day artificial intelligence, deep learning and machine learning systems.

2.0 Objectives

At the end of this unit you should be able to:

- List some examples of expert systems
- Outline the characteristics of an expert system
- Sketch the relationship between the components of an expert system
- Discuss the function of each component of an expert system
- Outline roles in expert system development
- Discuss the six steps involved in development and maintenance of expert systems
- Mention benefits and limitations of expert systems

3.0 Main Content

We can say, the Expert System in AI are computer applications. Also, with the help of this development, we can solve complex problems. It has level of human intelligence and expertise.

3.1 Examples of Expert Systems

Following are examples of Expert Systems

- MYCIN: It was based on backward chaining and could identify various bacteria that could cause acute infections. It could also recommend drugs based on the patient's weight.
- DENDRAL: Expert system used for chemical analysis to predict molecular structure.
- PXDES: Expert system used to predict the degree and type of lung cancer
- CaDet: Expert system that could identify cancer at early stages

3.2 Characteristic of Expert System

Following are Important characteristic of Expert System:

- **The Highest Level of Expertise:** The expert system offers the highest level of expertise. It provides efficiency, accuracy and imaginative problem-solving.
- **Right on Time Reaction:** An Expert System interacts in a very reasonable period of time with the user. The total time must be less than the time taken by an expert to get the most accurate solution for the same problem.
- **Good Reliability**: The expert system needs to be reliable, and it must not make any a mistake.
- **Flexible:** It is vital that it remains flexible as it the is possessed by an Expert system.
- **Effective Mechanism:** Expert System must have an efficient mechanism to administer the compilation of the existing knowledge in it.

- Capable of handling challenging decision & problems: An expert system is capable of handling challenging decision problems and delivering solutions.

3.3 Components of the expert system

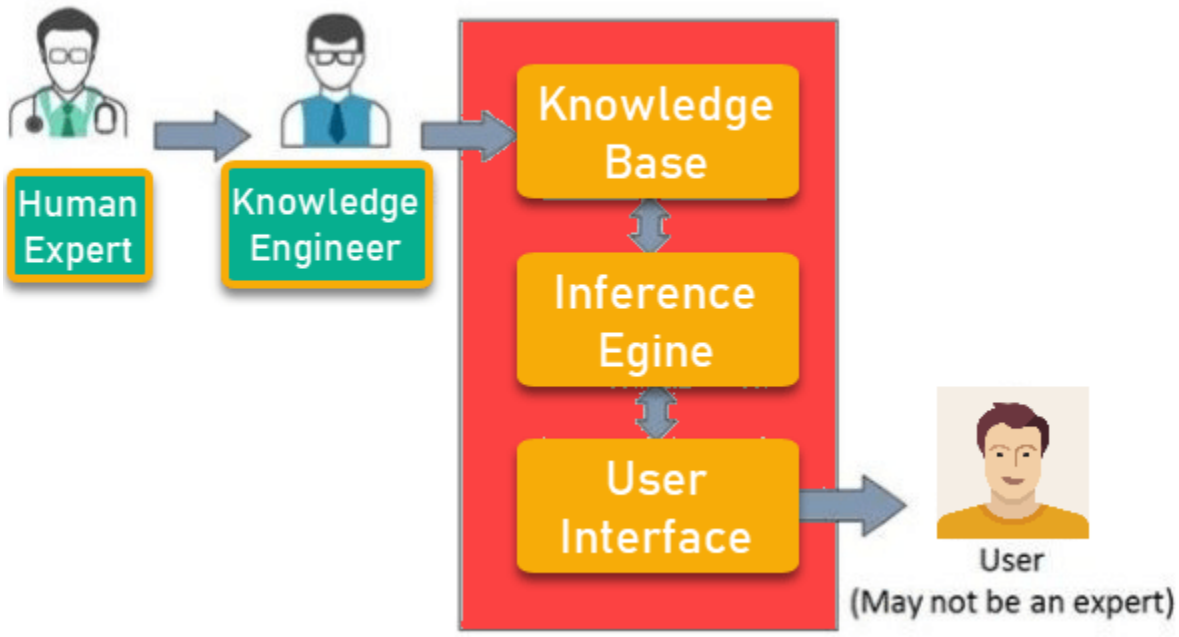


Figure 3.1: Expert System Components

The expert System consists of the following given components:

3.3.1 User Interface

The user interface is the most crucial part of the expert system. This component takes the user's query in a readable form and passes it to the inference engine. After that, it displays the results to the user. In other words, it's an interface that helps the user communicate with the expert system.

Generally, Expert System in AI users and ES itself uses User interface as a medium of interaction between users. Also, the user of the Expert Systems need not be necessarily an expert in Artificial Intelligence.

Although, at a particular recommendation, it explains how the Expert Systm has arrived. Hence, the explanation may appear in the following forms −

- Basically, the natural language displayed on a screen.
- Also, verbal narrations in natural language.

Further, listing of rule numbers displayed on the screen. The user interface makes it easy to trace the credibility of the deductions.

3.3.2 Inference Engine

The inference engine is the brain of the expert system. Inference engine contains rules to solve a specific problem. It refers the knowledge from the Knowledge Base. It selects facts and rules to apply when trying to answer the user's query. It provides reasoning about the information in the knowledge base. It also helps in deducting the problem to find the solution. This component is also helpful for formulating conclusions.

To arrive a particular solution, Inference Engine acquires and manipulates the knowledge. In case of rule-based Expert System in Artificial Intelligence −

- We have to apply rules to the facts. That is obtained from earlier rule application.
- Also, we have to add new knowledge if it's required.

Basically, it can resolve rules conflict. Whenever multiple rules are applicable to a particular case. To recommend a solution, the Inference Engine uses the following strategies −

a. Forward Chaining
b. Backwards Chaining

a. **Forward Chaining:** We can say that it's a type of strategy of an expert system. In this, we have to answers this question, "What can happen next?"

It is a data-driven strategy. The inferencing process moves from the facts of the case to a goal (conclusion). The strategy is thus driven by the facts available in the working memory and by the premises that can be satisfied. The inference engine attempts to match the condition (IF) part of each rule in the knowledge base with the facts currently available in the working memory. If several rules match, a conflict resolution procedure is invoked; for example, the lowest-numbered rule that adds new information to the working memory is fired. The conclusion of the firing rule is added to the working memory.

Forward-chaining systems are commonly used to solve more open-ended problems of a design or planning nature, such as, for example, establishing the configuration of a complex product.

Generally, it follows the chain of conditions and derivations. Also, reduces the outcome. Although, it needs to consider all the facts and rules. Further, have to sort them before concluding to a solution. Moreover, this is followed by working on a result.

For example: Prediction of share market status as an effect of changes in interest rates.

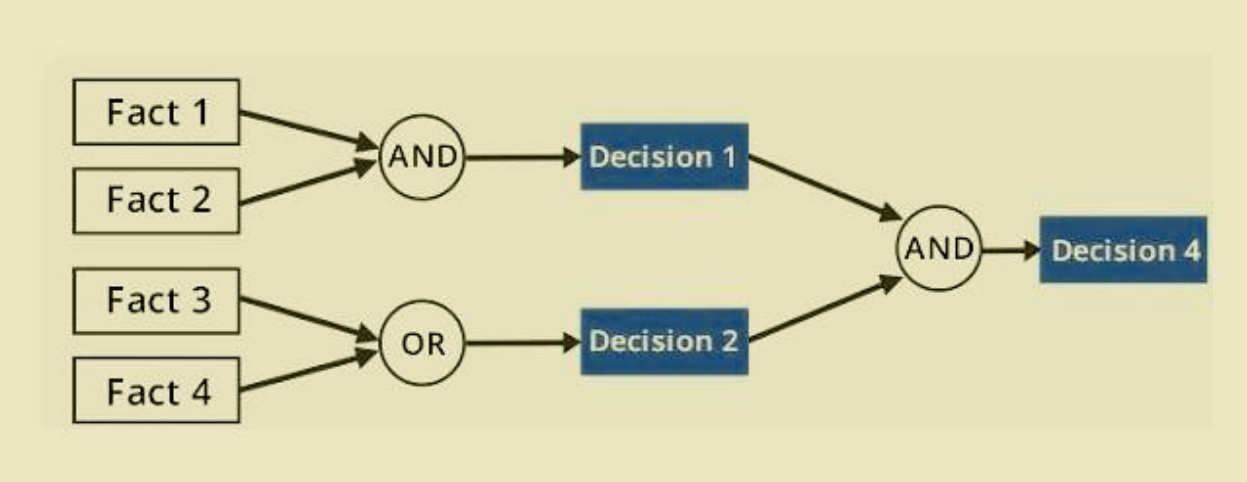


Figure 3.2: Forward Chaining

b. **Backward Chaining:** As with the use of this strategy, an expert system finds out the answer to the question, "Why this happened?"

The inference engine attempts to match the assumed (hypothesized) conclusion - the goal or subgoal state - with the conclusion (THEN) part of the rule. If such a rule is found, its premise becomes the new subgoal. In an ES with few possible goal states, this is a good strategy to pursue.

If a hypothesized goal state cannot be supported by the premises, the system will attempt to prove another goal state. Thus, possible conclusions are review until a goal state that can be supported by the premises is encountered.

Backward chaining is best suited for applications in which the possible conclusions are limited in number and well defined. Classification or diagnosis type systems, in which each of several possible conclusions can be checked to see if it is supported by the data, are typical applications.

Basically, as what has already happened matter a lot. Thus, it tries to find out which conditions could have happened in the past for this result. Hence, this strategy is followed by finding out cause or reason.

For example:

Diagnosis of blood cancer in humans.

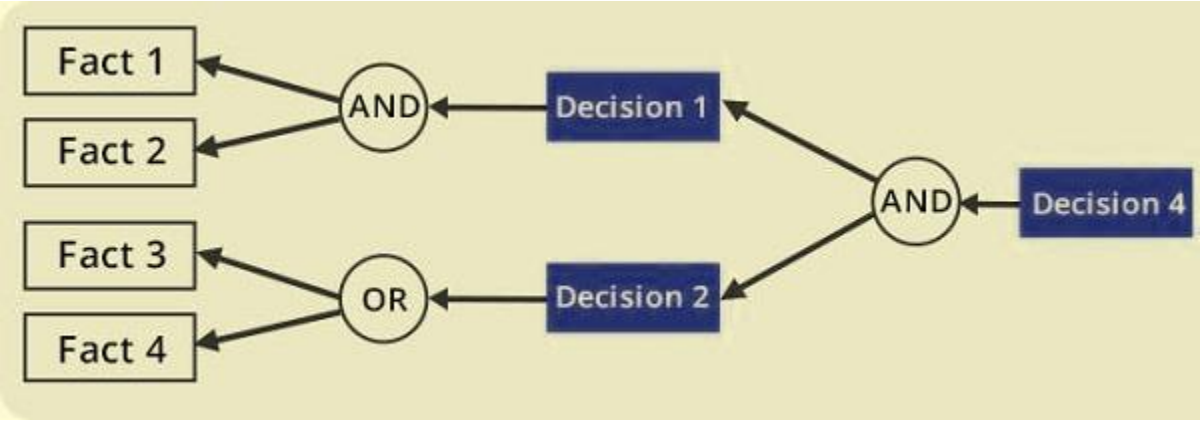


Figure 3.3: Knowledge Base in Expert Systems – Backward Chaining

3.3.3 Knowledge Base

The knowledge base is a repository of facts. It stores all the knowledge about the problem domain. It is like a large container of knowledge which is obtained from different experts of a specific field. Thus, we can say that the success of the Expert System mainly depends on the highly accurate and precise knowledge.

Generally, it contains domain-specific and high-quality knowledge. Also, as to exhibit intelligence required Knowledge. Although, a collection of highly accurate and precise knowledge is the reason for the success of Expert System in Artificial Intelligence.

3.3.3.1 What is Knowledge?

Basically, data is a collection of facts. Also, have to organize the information as data and facts about the task domain. Further, we can define knowledge as the combinations of Data, information, and past experience.

3.3.3.2 Components of Knowledge Base

Basically, there are two components present:

- Factual Knowledge – Generally, knowledge Engineers and Scholar used this in the task domain.
- Heuristic Knowledge – Generally, we can say it's all about practice, accurate judgment and one's ability of evaluation.

3.3.3.3 Knowledge Representation

Basically, it's types of method. We use this to organize and formalize knowledge in a knowledge base. Although, this method is in the form of IF-THEN-ELSE rules.

3.3.3.4 Knowledge Acquisition

Basically, quality and accuracy are the key reasons for the success of an expert system in AI. Also, knowledge engineer acquires exact information. Although, as they collect this information from a subject expert. He has had various ways to collect the information in different ways. Such as by recording, interviewing, and observing him at work, etc. He uses IF-THEN-ELSE rules, to organize and categorize information in a meaningful way. The knowledge engineer also monitors the development of the Expert System.

Figure 3.4: Knowledge Acquisition

Another widely used representation, called the unit (also known as frame, schema, or list structure) is based upon a more passive view of knowledge. The unit is an assemblage of associated symbolic knowledge about an entity to be represented. Typically, a unit consists of a list of properties of the entity and associated values for those properties.

Since every task domain consists of many entities that stand in various relations, the properties can also be used to specify relations, and the values of these properties are the names of other units that are linked according to the relations. One unit can also represent knowledge that is a "special case" of another unit, or some units can be "parts of" another unit.

The problem-solving model, or paradigm, organizes and controls the steps taken to solve the problem. One common but powerful paradigm involves chaining of IF-THEN rules to form a line of reasoning. If the chaining starts from a set of conditions and moves toward some conclusion, the method is called forward chaining. If the conclusion is known (for example, a goal to be achieved) but the path to that conclusion is not known, then reasoning backwards is called for, and the method is backward chaining. These problem-solving methods are built into program modules called inference engines or inference procedures that manipulate and use knowledge in the knowledge base to form a line of reasoning.

The knowledge base an expert uses is what he learned at school, from colleagues, and from years of experience. Presumably the more experience he has, the larger his store of knowledge. Knowledge allows him to interpret the information in his databases to advantage in diagnosis, design, and analysis.

Though an expert system consists primarily of a knowledge base and an inference engine, a couple of other features are worth mentioning: reasoning with uncertainty, and explanation of the line of reasoning.

Knowledge is almost always incomplete and uncertain. To deal with uncertain knowledge, a rule may have associated with it a confidence factor or a weight. The set of methods for using uncertain knowledge in combination with uncertain data in the reasoning process is called reasoning with uncertainty. An important subclass of methods for reasoning with uncertainty is called "fuzzy logic," and the systems that use them are known as "fuzzy systems."

Because an expert system uses uncertain or heuristic knowledge (as we humans do) its credibility is often in question (as is the case with humans). When an answer to a problem is questionable, we tend to want to know the rationale. If the rationale seems plausible, we tend to believe the answer. So it is with expert systems. Most expert systems have the ability to answer questions of the form: "Why is the answer X?" Explanations can be generated by tracing the line of reasoning used by the inference engine (Feigenbaum, McCorduck et al. 1988).

The most important ingredient in any expert system is knowledge. The power of expert systems resides in the specific, high-quality knowledge they contain about task domains. AI researchers will continue to explore and add to the current repertoire of knowledge representation and reasoning methods. But in knowledge resides the power. Because of the importance of knowledge in expert systems and because the current knowledge acquisition method is slow and tedious, much of the future of expert systems depends on breaking the knowledge acquisition bottleneck and in codifying and representing a large knowledge infrastructure.

3.4 Knowledge engineering

It is the art of designing and building expert systems, and knowledge engineers are its practitioners. Gerald M. Weinberg said of programming in The Psychology of Programming: "'Programming,' -- like 'loving,' -- is a single word that encompasses an infinitude of activities" (Weinberg 1971). Knowledge engineering is the same, perhaps more so. We stated earlier that knowledge engineering is an applied part of the science of artificial intelligence which, in turn, is a part of computer science. Theoretically, then, a knowledge engineer is a computer scientist who knows how to design and implement programs that incorporate artificial intelligence techniques. The nature of knowledge engineering is changing, however, and a new breed of knowledge engineers is emerging. We'll discuss the evolving nature of knowledge engineering later.

Today there are two ways to build an expert system. They can be built from scratch, or built using a piece of development software known as a "tool" or a "shell." Before we discuss these tools, let's briefly discuss what knowledge engineers do. Though different styles and methods of knowledge engineering exist, the basic approach is the same: a knowledge engineer interviews and observes a human expert or a group of experts and learns what the experts know, and how they reason with their knowledge. The engineer then translates the knowledge into a computer-usable language, and designs an inference engine, a reasoning structure, that uses the knowledge appropriately. He also determines how to integrate the use of uncertain knowledge in the reasoning process, and what kinds of explanation would be useful to the end user.

Next, the inference engine and facilities for representing knowledge and for explaining are programmed, and the domain knowledge is entered into the program piece by piece. It may be that the inference engine is not just right; the form of knowledge representation is awkward for the kind of knowledge needed for the task; and the expert might decide the pieces of knowledge are wrong. All these are discovered and modified as the expert system gradually gains competence.

The discovery and cumulation of techniques of machine reasoning and knowledge representation is generally the work of artificial intelligence research. The discovery and cumulation of knowledge of a task domain is the province of domain experts. Domain knowledge consists of both formal, textbook knowledge, and experiential knowledge -- the expertise of the experts.

3.5 Tools, Shells, and Skeletons

Compared to the wide variation in domain knowledge, only a small number of AI methods are known that are useful in expert systems. That is, currently there are only a handful of ways in which to represent knowledge, or to make inferences, or to generate explanations. Thus, systems can be built that contain these useful methods without any domain-specific knowledge. Such systems are known as skeletal systems, shells, or simply AI tools.

Building expert systems by using shells offers significant advantages. A system can be built to perform a unique task by entering into a shell all the necessary knowledge about a task domain. The inference engine that applies the knowledge to the task at hand is built into the shell. If the program is not very complicated and if an expert has had some training in the use of a shell, the expert can enter the knowledge himself.

Many commercial shells are available today, ranging in size from shells on PCs, to shells on workstations, to shells on large mainframe computers. They range in price from hundreds to tens of thousands of dollars, and range in complexity from simple, forward-chained, rule-based systems requiring two days of training to those so complex that only highly trained knowledge engineers can use them to advantage. They range from general-purpose shells to shells custom-tailored to a class of tasks, such as financial planning or real-time process control.

Although shells simplify programming, in general they don't help with knowledge acquisition. Knowledge acquisition refers to the task of endowing expert systems with knowledge, a task currently performed by knowledge engineers. The choice of reasoning method, or a shell, is important, but it isn't as important as the accumulation of high-quality knowledge. The power of an expert system lies in its store of knowledge about the task domain -- the more knowledge a system is given, the more competent it becomes.

3.6 Expert System Technology

There are several levels of ES technologies available. Two important things to keep in mind when selecting ES tools include:

1. The tool selected for the project has to match the capability and sophistication of the projected ES, in particular, the need to integrate it with other subsystems such as databases and other components of a larger information system.
2. The tool also has to match the qualifications of the project team.

Expert systems technologies include:

1. Specific expert systems: These expert systems actually provide recommendations in a specific task domain.
2. Expert system shells: are the most common vehicle for the development of specific ESs. A shell is an expert system without a knowledge base. A shell furnishes the ES developer with the inference engine, user interface, and the explanation and knowledge acquisition facilities.
3. Domain-specific shells are actually incomplete specific expert systems, which require much less effort in order to field an actual system.

4. Expert system development environments: These systems expand the capabilities of shells in various directions. They run on engineering workstations, minicomputers, or mainframes; offer tight integration with large databases; and support the building of large expert systems.
5. High-level programming languages: Several ES development environments have been rewritten from LISP into a procedural language more commonly found in the commercial environment, such as C or C++. ESs are now rarely developed in a programming language.

3.7 Roles in Expert System Development

Three fundamental roles in building expert systems are:

1. Expert - Successful ES systems depend on the experience and application of knowledge that the people can bring to it during its development. Large systems generally require multiple experts.
2. Knowledge engineer - The knowledge engineer has a dual task. This person should be able to elicit knowledge from the expert, gradually gaining an understanding of an area of expertise. Intelligence, tact, empathy, and proficiency in specific techniques of knowledge acquisition are all required of a knowledge engineer. Knowledge-acquisition techniques include conducting interviews with varying degrees of structure, protocol analysis, observation of experts at work, and analysis of cases.
3. On the other hand, the knowledge engineer must also select a tool appropriate for the project and use it to represent the knowledge with the application of the knowledge acquisition facility.
4. User - A system developed by an end user with a simple shell, is built rather quickly an inexpensively. Larger systems are built in an organized development effort. A prototype-oriented iterative development strategy is commonly used. ESs lends themselves particularly well to prototyping.

3.8 Development and Maintenance of Expert Systems

Steps in the methodology for the iterative process of ES development and maintenance include:

1. Problem Identification and Feasibility Analysis: the problem must be suitable for an expert system to solve it. It must find an expert for the project. The cost-effectiveness of the system has to be established (feasibility)
2. System Design and ES Technology Identification: The system is being designed. The needed degree of integration with other subsystems and databases is established. The concepts that best represent the domain knowledge are worked out. The best way to represent the knowledge and to perform inferencing should be established with sample cases.
3. Development of Prototype: The knowledge engineer works with the expert to place the initial kernel of knowledge in the knowledge base. The knowledge needs to be expressed in the language of the specific tool chosen for the project.
4. Testing and Refinement of Prototype: By using sample cases, the prototype is tested, and deficiencies in performance are noted. End users test the prototypes of the ES.

5. Complete and Field the ES: The interaction of the ES with all elements of its environment, including users and other information systems, is ensured and tested. ES is documented and user training is conducted
6. Maintain the System: The system is keep current primarily by updating its knowledge base. interfaces with other information systems have to be maintained as well, as those systems evolve.

3.9 Expert Systems in Organizations: Benefits and Limitations

Expert systems offer both tangible and important intangible benefits to owner companies. These benefits should be weighed against the development and exploitation costs of an ES, which are high for large, organizationally important ESs.

3.9.1 Benefits of Expert Systems in Artificial Intelligence

An ES is no substitute for a knowledge worker's overall performance of the problem-solving task. But these systems can dramatically reduce the amount of work the individual must do to solve a problem, and they do leave people with the creative and innovative aspects of problem solving.

Some of the possible organizational benefits of expert systems are:

1. An Es can complete its part of the tasks much faster than a human expert.
2. The error rate of successful systems is low, sometimes much lower than the human error rate for the same task.
3. ESs make consistent recommendations
4. ESs are a convenient vehicle for bringing to the point of application difficult-to-use sources of knowledge.
5. ESs can capture the scarce expertise of a uniquely qualified expert.
6. ESs can become a vehicle for building up organizational knowledge, as opposed to the knowledge of individuals in the organization.
7. When use as training vehicles, ESs result in a faster learning curve for novices.
8. The company can operate an ES in environments hazardous for humans.

3.9.2 Limitations of Expert Systems

No technology offers an easy and total solution. Large systems are costly and require significant development time and computer resources. ESs also have their limitations which include:

1. Limitations of the technology
2. Problems with knowledge acquisition
3. Operational domains as the principal area of ES application
4. Maintaining human expertise in organizations

3.10 AI Expert System with Applications

a. Design Domain: We use expert systems in designing of camera lens and automobile.

b. Monitoring Systems: In this data is compared with an observed system

c. Process Control Systems: We have to control physical process based on monitoring

d. Knowledge Domain: Finding out faults in vehicles, computers

e. Finance Commerce: Expert system is used to detect possible fraud.

4.0 Conclusion

As a result, we have studied Expert Systems in artificial intelligence. Also, learned characteristics, components, types and benefits along with expert systems applications of AI.

5.0 Summary

So, in this unit, the discussion was all about Expert System in Artificial Intelligence. Hope you like our explanation.

6.0 Tutor Marked Assignment

i. Discuss the six steps involved in development and maintenance of expert systems
ii. Write a comprehensive note on knowledge engineering.

7.0 References/Further Readings

1) Lucas, P., & Van Der Gaag, L. (1991). Principles of expert systems. Wokingham: Addison-Wesley.
2) Engelmore, R. S., & Feigenbaum, E. (1993). Expert systems and artificial intelligence. Expert Systems, 100(2). (Chapter 1)
3) DataFlair Team · What is Expert System in Artificial Intelligence – How it Solve Problems. Published January 23, 2018 · Updated November 15, 2018, https://data-flair.training/blogs/expert-system/ retrieved on 07/30/2019
4) Turban, E., & Frenzel, L. E. (1992). Expert systems and applied artificial intelligence. Prentice Hall Professional Technical Reference. Chapter 11, http://www.umsl.edu/~joshik/msis480/chapt11.htm retrieved on 07/30/2019
5) Expert System in Artificial Intelligence: What is, Applications, Example. https://www.guru99.com/expert-systems-with-applications.html retrieved on 07/30/2019
6) Feigenbaum, McCorduck et al. 1988)
7) Weinberg 1971

Unit Four: Robot Planning

CONTENT

7.0 References/Further Readings

1.0 Introduction

All of a robot's deliberations ultimately come down to deciding how to move effectors. The point-to-point motion problem is to deliver the robot or its end effector to a designated target location. A greater challenge is the compliant motion problem, in which a robot moves while being in physical contact with an obstacle. An example of compliant motion is a robot manipulator that screws in a light bulb, or a robot that pushes a box across a table top.

We begin by finding a suitable representation in which motion-planning problems can be described and solved. It turns out that the configuration space—the space of robot states defined by location, orientation, and joint angles—is a better place to work than the original 3D space. The path planning problem is to find a path from one configuration to another in configuration space. We have already encountered various versions of the path-planning problem throughout this material; the complication added by robotics is that path planning involves continuous spaces. There are two main approaches: cell decomposition and skeletonization.

Each reduces the continuous path-planning problem to a discrete graph-search problem. In this section, we assume that motion is deterministic and that localization of the robot is exact. Subsequent sections will relax these assumptions.

2.0 Objectives

At the end of this unit, you should be able to;

- Discuss how cell decomposition methods can be improved
- Mention the problems associated to configuration spaces
- Explain what is meant by potential field
- Using an example, discuss how skeletonization methods are applied to path planning algorithms
- Mention how best to model the uncertainty issues in robots motion
- Able to calculate the forces exerted by robots during motion

3.0 Main Content

3.1 PLANNING TO MOVE

3.1.1 Configuration space

We will start with a simple representation for a simple robot motion problem. Consider the robot arm shown in Figure 3.1(a). It has two joints that move independently. Moving the joints alters the (x, y) coordinates of the elbow and the gripper. (The arm cannot move in the z direction.) This suggests that the robot's configuration can be described by a four-dimensional coordinate: (x_e, y_e) for the location of the elbow relative to the environment and (x_g, y_g) for the location of the gripper. Clearly, these four coordinates characterize the full state of the robot. They constitute what is known as workspace representation, since the coordinates of the robot are specified in the same coordinate system as the objects it seeks to manipulate (or to avoid). Workspace representations are well-suited for collision checking, especially if the robot and all objects are represented by simple polygonal models. The

problem with the workspace representation is that not all workspace coordinates are actually attainable, even in the absence of obstacles. This is because of the linkage constraints on the space of attainable workspace coordinates. For example, the elbow position (x_e, y_e) and the gripper position (x_g, y_g) are always a fixed distance apart, because they are joined by a rigid forearm. A robot motion planner defined over workspace coordinates faces the challenge of generating paths that adhere to these constraints. This is particularly tricky because the state space is continuous and the constraints are nonlinear. It turns out to be easier to plan with a configuration space representation. Instead of representing the state of the robot by the Cartesian coordinates of its elements, we represent the state by a configuration of the robot's joints. Our example robot possesses two joints. Hence, we can represent its state with the two angles φs and φe for the shoulder joint and elbow joint, respectively. In the absence of any obstacles, a robot could freely take on any value in configuration space. In particular, when planning a path one could simply connect the present configuration and the target configuration by a straight line. In following this path, the robot would then move its joints at a constant velocity, until a target location is reached.

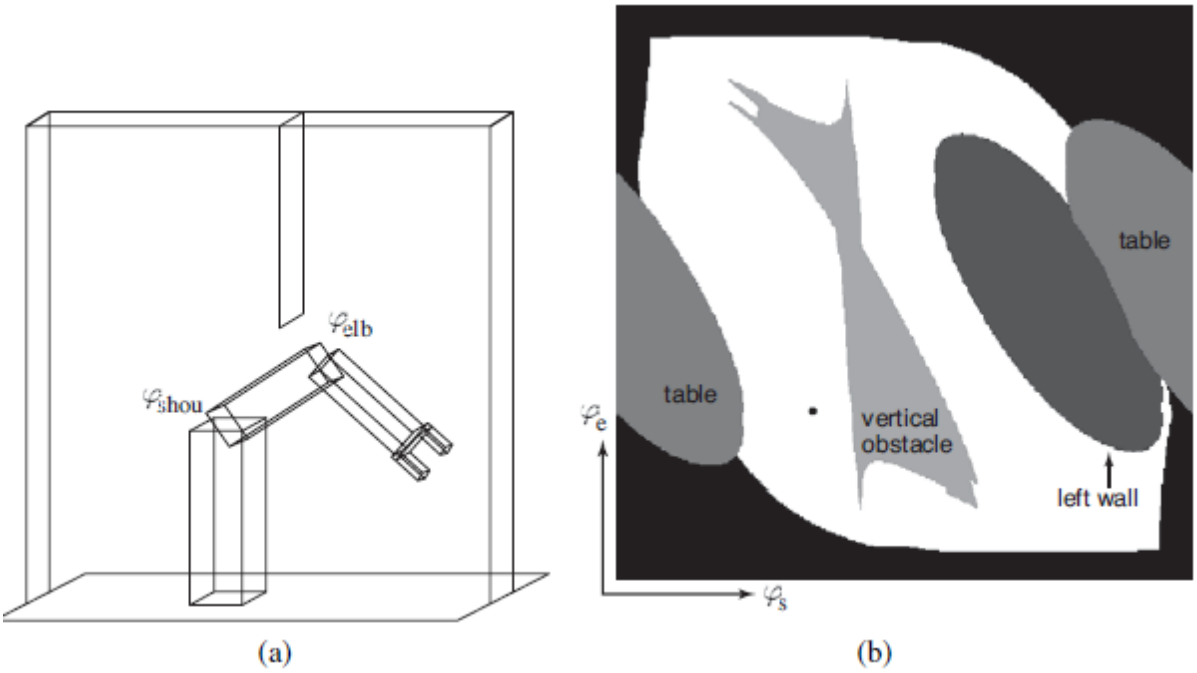


Figure 3.1 (a) Workspace representation of a robot arm with 2 DOFs. The workspace is a box with a flat obstacle hanging from the ceiling. (b) Configuration space of the same robot. Only white regions in the space are configurations that are free of collisions. The dot in this diagram corresponds to the configuration of the robot shown on the left.

Unfortunately, configuration spaces have their own problems. The task of a robot is usually expressed in workspace coordinates, not in configuration space coordinates. This raises the question of how to map between workspace coordinates and configuration space. Transforming configuration space coordinates into workspace coordinates is simple: it involves a series of straightforward coordinate transformations. These transformations are linear for prismatic joints and trigonometric for revolute joints. This chain of coordinate transformation is known as kinematics.

The inverse problem of calculating the configuration of a robot whose effector location is specified in workspace coordinates is known as inverse kinematics. Calculating the inverse kinematics is hard, especially for robots with many DOFs. In particular, the solution is seldom unique. Figure 3.1(a) shows one of two possible configurations that put the gripper in the same location. (The other configuration would has the elbow below the shoulder.)

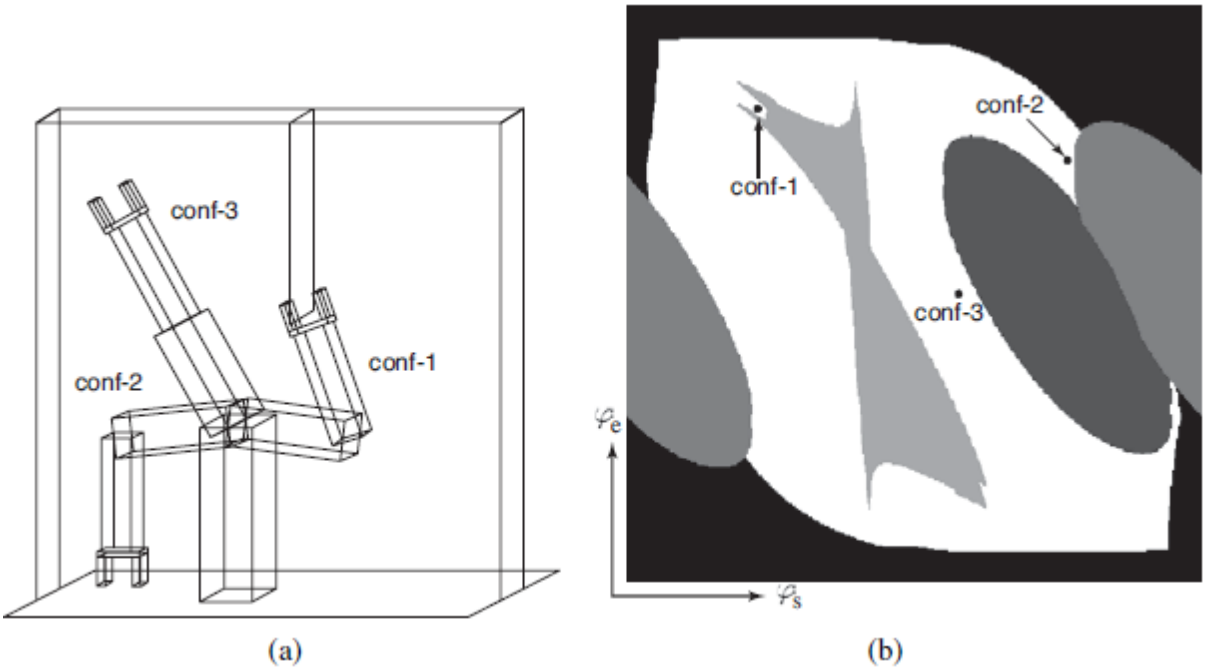


Figure 3.2 Three robot configurations, shown in workspace and configuration space.

In general, this two-link robot arm has between zero and two inverse kinematic solutions for any set of workspace coordinates. Most industrial robots have sufficient degrees of freedom to find infinitely many solutions to motion problems. To see how this is possible, simply imagine that we added a third revolute joint to our example robot, one whose rotational axis is parallel to the ones of the existing joints. In such a case, we can keep the location (but not the orientation!) of the gripper fixed and still freely rotate its internal joints, for most configurations of the robot. With a few more joints (how many?) we can achieve the same effect while keeping the orientation of the gripper constant as well. We have already seen an example of this in the "experiment" of placing your hand on the desk and moving your elbow. The kinematic constraint of your hand position is insufficient to determine the configuration of your elbow. In other words, the inverse kinematics of your shoulder–arm assembly possesses an infinite number of solutions.

The second problem with configuration space representations arises from the obstacles that may exist in the robot's workspace. Our example in Figure 3.1(a) shows several such obstacles, including a free-hanging obstacle that protrudes into the center of the robot's workspace. In workspace, such obstacles

take on simple geometric forms—especially in most robotics textbooks, which tend to focus on polygonal obstacles. But how do they look in configuration space?

Figure 3.1(b) shows the configuration space for our example robot, under the specific obstacle configuration shown in Figure 3.1(a). The configuration space can be decomposed into two subspaces: the space of all configurations that a robot may attain, commonly called free space, and the space of unattainable configurations, called occupied space. The white area in Figure 3.1(b) corresponds to the free space. All other regions correspond to occupied space. The different shadings of the occupied space corresponds to the different objects in the robot's workspace; the black region surrounding the entire free space corresponds to configurations in which the robot collides with itself. It is easy to see that extreme values of the shoulder or elbow angles cause such a violation. The two oval-shaped regions on both sides of the robot correspond to the table on which the robot is mounted. The third oval region corresponds to the left wall. Finally, the most interesting object in configuration space is the vertical obstacle that hangs from the ceiling and impedes the robot's motions. This object has a funny shape in configuration space: it is highly nonlinear and at places even concave. With a little bit of imagination the reader will recognize the shape of the gripper at the upper left end. We encourage the reader to pause for a moment and study this diagram. The shape of this obstacle is not at all obvious! The dot inside Figure 25.14(b) marks the configuration of the robot, as shown in Figure 3.1(a). Figure 3.2 depicts three additional configurations, both in workspace and in configuration space. In configuration conf-1, the gripper encloses the vertical obstacle.

Even if the robot's workspace is represented by flat polygons, the shape of the free space can be very complicated. In practice, therefore, one usually probes a configuration space instead of constructing it explicitly. A planner may generate a configuration and then test to see if it is in free space by applying the robot kinematics and then checking for collisions in workspace coordinates.

3.1.2 Cell decomposition methods

The first CELL approach to path planning uses cell decomposition—that is, it decomposes the free space into a finite number of contiguous regions, called cells. These regions have the important property that the path-planning problem within a single region can be solved by simple means (e.g., moving along a straight line). The path-planning problem then becomes a discrete graph-search problem, very much like the search problems introduced previously.

The simplest cell decomposition consists of a regularly spaced grid. Figure 3.3(a) shows a square grid decomposition of the space and a solution path that is optimal for this grid size. Grayscale shading indicates the value of each free-space grid cell—i.e., the cost of the shortest path from that cell to the goal. (These values can be computed by a deterministic form of the VALUE-ITERATION algorithm.) Figure 25.16(b) shows the corresponding workspace trajectory for the arm. Of course, we can also use the A*algorithm to find a shortest path.

Such a decomposition has the advantage that it is extremely simple to implement, but it also suffers from three limitations. First, it is workable only for low-dimensional configuration spaces, because the number of grid cells increases exponentially with d, the number of dimensions. Sounds familiar? This is the curse!dimensionality@of dimensionality. Second, there is the problem of what to do with cells that are "mixed"—that is, neither entirely within free space nor entirely within occupied space. A solution path that includes such a cell may not be a real solution, because there may be no way to cross the cell

in the desired direction in a straight line. This would make the path planner unsound. On the other hand, if we insist that only completely free cells may be used, the planner will be incomplete, because it might be the case that the only paths to the goal go through mixed cells—especially if the cell size is comparable to that of the passageways and clearances in the space. And third, any path through a discretized state space will not be smooth. It is generally difficult to guarantee that a smooth solution exists near the discrete path. So a robot may not be able to execute the solution found through this decomposition.

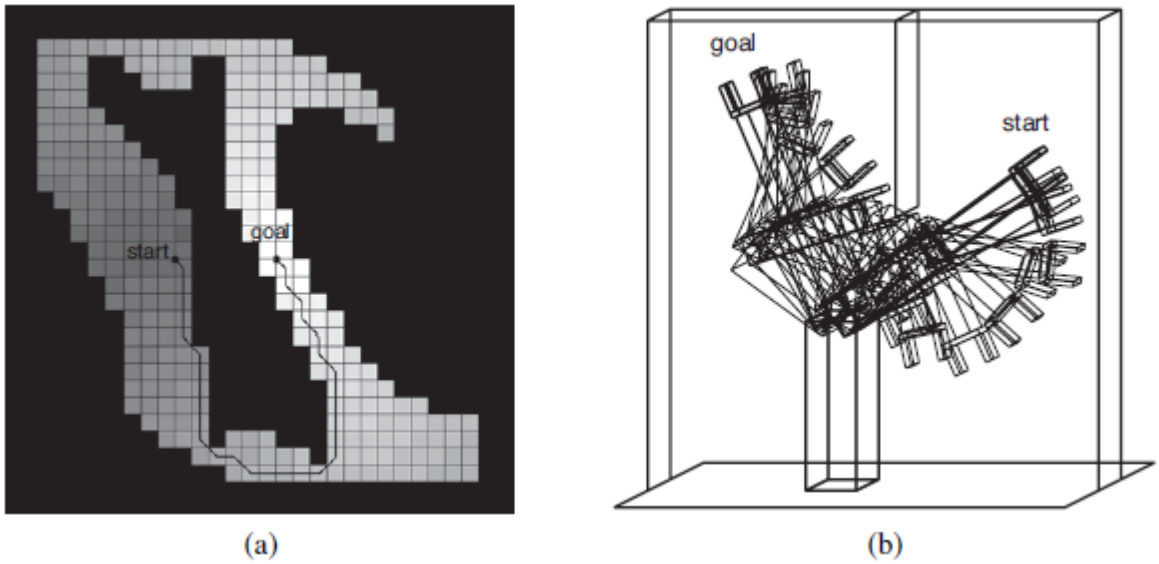


Figure 3.3 (a) Value function and path found for a discrete grid cell approximation of the configuration space. (b) The same path visualized in workspace coordinates. Notice how the robot bends its elbow to avoid a collision with the vertical obstacle.

Cell decomposition methods can be improved in a number of ways, to alleviate some of these problems. The first approach allows further subdivision of the mixed cells—perhaps using cells of half the original size. This can be continued recursively until a path is found that lies entirely within free cells. (Of course, the method only works if there is a way to decide if a given cell is a mixed cell, which is easy only if the configuration space boundaries have relatively simple mathematical descriptions.) This method is complete provided there is a bound on the smallest passageway through which a solution must pass. Although it focuses most of the computational effort on the tricky areas within the configuration space, it still fails to scale well to high-dimensional problems because each recursive splitting of a cell creates 2d smaller cells. A second way to obtain a complete algorithm is to insist on an exact cell decomposition of the free space. This method must allow cells to be irregularly shaped where they meet the boundaries of free space, but the shapes must still be “simple” in the sense that it should be easy to compute a traversal of any free cell. This technique requires some quite advanced geometric ideas, so we shall not pursue it further here.

Examining the solution path shown in Figure 3.3(a), we can see an additional difficulty that will have to be resolved. The path contains arbitrarily sharp corners; a robot moving at any finite speed could not execute such a path. This problem is solved by storing certain continuous values for each grid cell. Consider an algorithm which stores, for each grid cell, the exact, continuous state that was attained with the cell was first expanded in the search.

Assume further, that when propagating information to nearby grid cells, we use this continuous state as a basis, and apply the continuous robot motion model for jumping to nearby cells. In doing so, we can now guarantee that the resulting trajectory is smooth and can indeed be executed by the robot. One HYBRID A* algorithm that implements this is hybrid A*.

3.1.3 Modified cost functions

Notice that in Figure 3.3, the path goes very close to the obstacle. Anyone who has driven a car knows that a parking space with one millimeter of clearance on either side is not really a parking space at all; for the same reason, we would prefer solution paths that are robust with respect to small motion errors. This problem can be solved by introducing a potential field. A potential field is a function defined over state space, whose value grows with the distance to the closest obstacle.

Figure 3.4(a) shows such a potential field—the darker a configuration state, the closer it is to an obstacle. The potential field can be used as an additional cost term in the shortest-path calculation. This induces an interesting tradeoff. On the one hand, the robot seeks to minimize path length to the goal. On the other hand, it tries to stay away from obstacles by virtue of minimizing the potential function. With the appropriate weight balancing the two objectives, a resulting path may look like the one shown in Figure 3.4(b). This figure also displays the value function derived from the combined cost function, again calculated by value iteration. Clearly, the resulting path is longer, but it is also safer.

There exist many other ways to modify the cost function. For example, it may be desirable to smooth the control parameters over time. For example, when driving a car, a smooth path is better than a jerky one. In general, such higher-order constraints are not easy to accommodate in the planning process, unless we make the most recent steering command a part of the state. However, it is often easy to smooth the resulting trajectory after planning, using conjugate gradient methods. Such post-planning smoothing is essential in many real-world applications.

3.1.4 Skeletonization methods

The second major family of path-planning algorithms is based on the idea of skeletonization. These algorithms reduce the robot's free space to a one-dimensional representation, for which the planning problem is easier. This lower-dimensional representation is called a skeleton of the configuration space.

Figure 3.5 shows an example skeletonization: it is a Voronoi graph of the free space—the set of all points that are equidistant to two or more obstacles. To do path planning with a Voronoi graph, the robot first changes its present configuration to a point on the Voronoi graph. It is easy to show that this can always be achieved by a straight-line motion in configuration space. Second, the robot follows the Voronoi graph until it reaches the point nearest to the target configuration. Finally, the robot leaves the Voronoi graph and moves to the target. Again, this final step involves straight-line motion in configuration space.

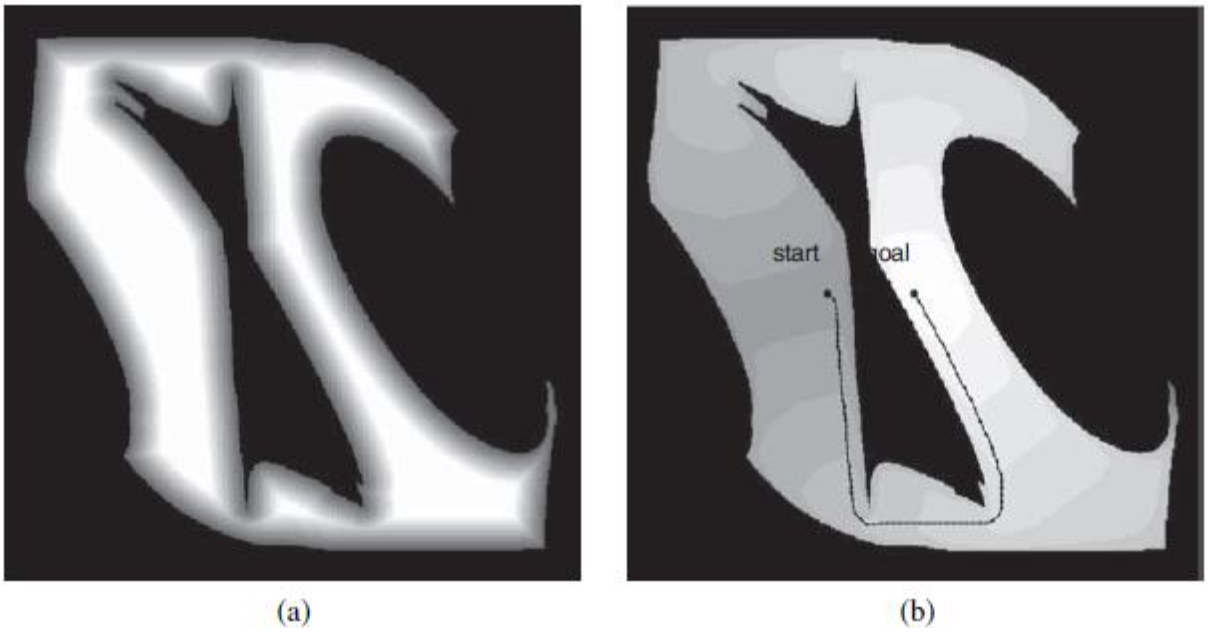


Figure 3.4 (a) A repelling potential field pushes the robot away from obstacles. (b) Path found by simultaneously minimizing path length and the potential.

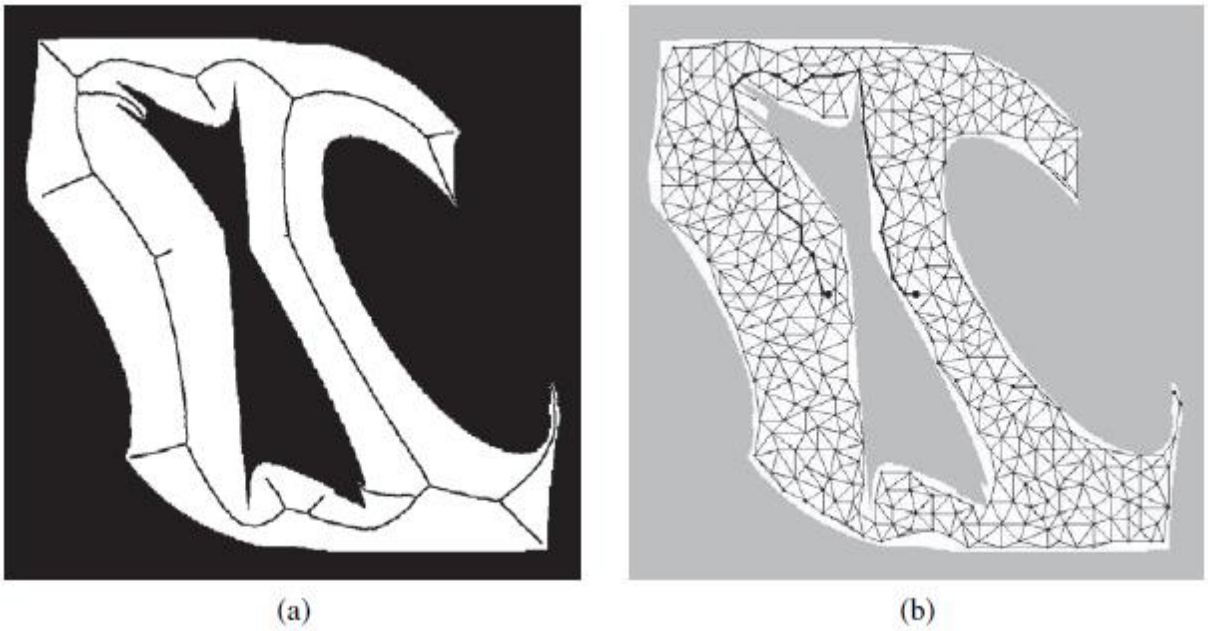


Figure 3.5 (a) The Voronoi graph is the set of points equidistant to two or more obstacles in configuration space. (b) A probabilistic roadmap, composed of 400 randomly chosen points in free space.

In this way, the original path-planning problem is reduced to finding a path on the Voronoi graph, which is generally one-dimensional (except in certain nongeneric cases) and has finitely many points where three or more one-dimensional curves intersect. Thus, finding the shortest path along the Voronoi graph is a discrete graph-search problem. Following the Voronoi graph may not give us the shortest path, but the resulting paths tend to maximize clearance. Disadvantages of Voronoi graph techniques are that they are difficult to apply to higher-dimensional configuration spaces, and that they tend to induce unnecessarily large detours when the configuration space is wide open. Furthermore, computing the Voronoi graph can be difficult, especially in configuration space, where the shapes of obstacles can be complex.

An alternative to the Voronoi graphs is the probabilistic roadmap, a skeletonization approach that offers more possible routes, and thus deals better with wide-open spaces. Figure 3.5(b) shows an example of a probabilistic roadmap. The graph is created by randomly generating a large number of configurations, and discarding those that do not fall into free space. Two nodes are joined by an arc if it is "easy" to reach one node from the other–for example, by a straight line in free space. The result of all this is a randomized graph in the robot's free space. If we add the robot's start and goal configurations to this graph, path planning amounts to a discrete graph search. Theoretically, this approach is incomplete, because a bad choice of random points may leave us without any paths from start to goal. It is possible to bound the probability of failure in terms of the number of points generated and certain geometric properties of the configuration space. It is also possible to direct the generation of sample points towards the areas where a partial search suggests that a good path may be found, working bidirectionally from both the start and the goal positions. With these improvements, probabilistic roadmap planning tends to scale better to high-dimensional configuration spaces than most alternative path-planning techniques.

3.2 PLANNING UNCERTAIN MOVEMENTS

None of the robot motion-planning algorithms discussed thus far addresses a key characteristic of robotics problems: uncertainty. In robotics, uncertainty arises from partial observability of the environment and from the stochastic (or unmodeled) effects of the robot's actions. Errors can also arise from the use of approximation algorithms such as particle filtering, which does not provide the robot with an exact belief state even if the stochastic nature of the environment is modeled perfectly.

Most of today's robots use deterministic algorithms for decision making, such as the path-planning algorithms of the previous section. To do so, it is common practice to extract the most likely state from the probability distribution produced by the state estimation algorithm.

The advantage of this approach is purely computational. Planning paths through configuration space is already a challenging problem; it would be worse if we had to work with a full probability distribution over states. Ignoring uncertainty in this way works when the uncertainty is small. In fact, when the environment model changes over time as the result of incorporating sensor measurements, many robots plan paths online during plan execution. This is the online replanning technique.

Unfortunately, ignoring the uncertainty does not always work. In some problems the robot's uncertainty is simply too massive: How can we use a deterministic path planner to control a mobile robot that has no clue where it is? In general, if the robot's true state is not the one identified by the maximum likelihood rule, the resulting control will be suboptimal.

Depending on the magnitude of the error this can lead to all sorts of unwanted effects, such as collisions with obstacles.

The field of robotics has adopted a range of techniques for accommodating uncertainty. If the robot faces uncertainty only in its state transition, but its state is fully observable, the problem is best modeled as a Markov decision process (MDP). The solution of an MDP is an optimal policy, which tells the robot what to do in every possible state. In this way, it can handle all sorts of motion errors, whereas a single-path solution from a deterministic planner would be much less robust. In robotics, policies are called navigation functions. The value function shown in Figure 3.3(a) can be converted into such a navigation function simply by following the gradient.

Partial observability makes the problem much harder. The resulting robot control problem is a partially observable MDP, or POMDP. In such situations, the robot maintains an internal belief state. The solution to a POMDP is a policy defined over the robot's belief state. Put differently, the input to the policy is an entire probability distribution. This enables the robot to base its decision not only on what it knows, but also on what it does not know. For example, if it is uncertain about a critical state variable, it can rationally invoke an information gathering action. This is impossible in the MDP framework, since MDPs assume full observability. Unfortunately, techniques that solve POMDPs exactly are inapplicable to robotics—there are no known techniques for high-dimensional continuous spaces. Discretization produces POMDPs that are far too large to handle. One remedy is to make the minimization of uncertainty a control objective. For example, the coastal navigation heuristic requires the robot to stay near known landmarks to decrease its uncertainty. Another approach applies variants of the probabilistic roadmap planning method to the belief space representation. Such methods tend to scale better to large discrete POMDPs.

3.2.1 Robust methods

Uncertainty can also be handled using so-called robust control methods rather than probabilistic methods. A robust method is one that assumes a bounded amount of uncertainty in each aspect of a problem, but does not assign probabilities to values within the allowed interval. A robust solution is one that works no matter what actual values occur, provided they are within the assumed interval. An extreme form of robust method is the conformant planning approach. It produces plans that work with no state information at all.

Here, we look at a robust method that is used for fine-motion planning (or FMP) in robotic assembly tasks. Fine-motion planning involves moving a robot arm in very close proximity to a static environment object. The main difficulty with fine-motion planning is that the required motions and the relevant features of the environment are very small. At such small scales, the robot is unable to measure or control its position accurately and may also be uncertain of the shape of the environment itself; we will assume that these uncertainties are all bounded. The solutions to FMP problems will typically be conditional plans or policies that make use of sensor feedback during execution and are guaranteed to work in all situations consistent with the assumed uncertainty bounds.

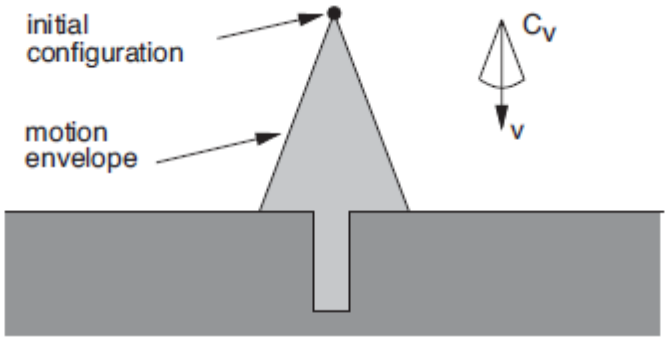


Figure 3.6 A two-dimensional environment, velocity uncertainty cone, and envelope of possible robot motions. The intended velocity is v, but with uncertainty the actual velocity could be anywhere in C_v, resulting in a final configuration somewhere in the motion envelope, which means we wouldn't know if we hit the hole or not.

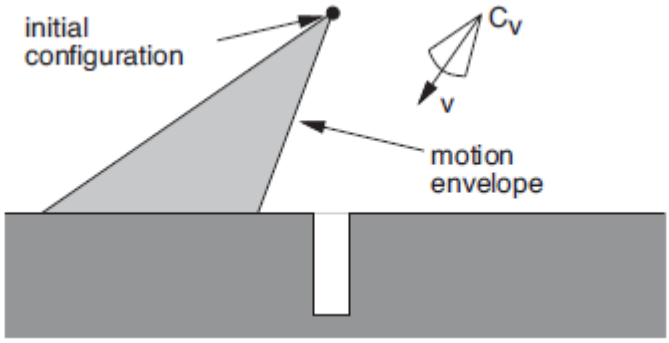


Figure 3.7 The first motion command and the resulting envelope of possible robot motions. No matter what the error, we know the final configuration will be to the left of the hole.

A fine-motion plan consists of a series of guarded motions. Each guarded motion consists of (1) a motion command and (2) a termination condition, which is a predicate on the robot's sensor values, and returns true to indicate the end of the guarded move. The motion commands are typically compliant motions that allow the effector to slide if the motion command would cause collision with an obstacle. As an example, Figure 3.6 shows a two-dimensional configuration space with a narrow vertical hole. It could be the configuration space for insertion of a rectangular peg into a hole or a car key into the ignition. The motion commands are constant velocities. The termination conditions are contact with a surface. To model uncertainty in control, we assume that instead of moving in the commanded direction, the robot's actual motion lies in the cone C_v about it. The figure shows what would happen if we commanded a velocity straight down from the initial configuration. Because of the uncertainty in velocity, the robot could move anywhere in the conical envelope, possibly going into the hole, but more likely landing to one side of it. Because the robot would not then know which side of the hole it was on, it would not know which way to move.

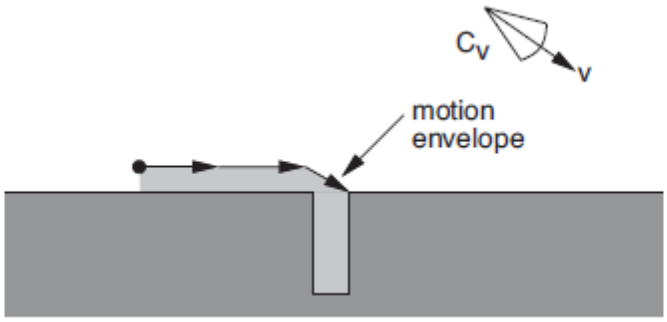


Figure 3.8 The second motion command and the envelope of possible motions. Even with error, we will eventually get into the hole.

A more sensible strategy is shown in Figures 3.7 and 3.8. In Figure 3.7, the robot deliberately moves to one side of the hole. The motion command is shown in the figure, and the termination test is contact with any surface. In Figure 3.8, a motion command is given that causes the robot to slide along the surface and into the hole. Because all possible velocities in the motion envelope are to the right, the robot will slide to the right whenever it is in contact with a horizontal surface. It will slide down the right-hand vertical edge of the hole when it touches it, because all possible velocities are down relative to a vertical surface.

It will keep moving until it reaches the bottom of the hole, because that is its termination condition. In spite of the control uncertainty, all possible trajectories of the robot terminate in contact with the bottom of the hole—that is, unless surface irregularities cause the robot to stick in one place.

As one might imagine, the problem of constructing fine-motion plans is not trivial; in fact, it is a good deal harder than planning with exact motions. One can either choose a fixed number of discrete values for each motion or use the environment geometry to choose directions that give qualitatively different behavior. A fine-motion planner takes as input the configuration-space description, the angle of the velocity uncertainty cone, and a specification of what sensing is possible for termination (surface contact in this case). It should produce a multistep conditional plan or policy that is guaranteed to succeed, if such a plan exists.

Our example assumes that the planner has an exact model of the environment, but it is possible to allow for bounded error in this model as follows. If the error can be described in terms of parameters, those parameters can be added as degrees of freedom to the configuration space. In the last example, if the depth and width of the hole were uncertain, we could add them as two degrees of freedom to the configuration space. It is impossible to move the robot in these directions in the configuration space or to sense its position directly. But both those restrictions can be incorporated when describing this problem as an FMP problem by appropriately specifying control and sensor uncertainties. This gives a complex, four-dimensional planning problem, but exactly the same planning techniques can be applied.

Notice that unlike the decision-theoretic methods, this kind of robust approach results in plans designed for the worst-case outcome, rather than maximizing the expected quality of the plan. Worst-case plans are optimal in the decision-theoretic sense only if failure during execution is much worse than any of the other costs involved in execution.

3.3 MOVING

So far, we have talked about how to plan motions, but not about how to move. Our plans particularly those produced by deterministic path planners—assume that the robot can simply follow any path that the algorithm produces. In the real world, of course, this is not the case. Robots have inertia and cannot execute arbitrary paths except at arbitrarily slow speeds. In most cases, the robot gets to exert forces **rather than specify positions. This section discusses methods for calculating these forces.**

3.3.1 Dynamics and control

We previously introduced the notion of dynamic state, which extends the kinematic state of a robot by its velocity. For example, in addition to the angle of a robot joint, the dynamic state also captures the rate of change of the angle, and possibly even its momentary acceleration.

The transition model for a dynamic state representation includes the effect of forces on this rate of change. Such models are typically expressed via differential equations, which are equations that relate a quantity (e.g., a kinematic state) to the change of the quantity over time (e.g., velocity). In principle, we could have chosen to plan robot motion using dynamic models, instead of our kinematic models. Such a methodology would lead to superior robot performance, if we could generate the plans. However, the dynamic state has higher dimension than the kinematic space, and the curse of dimensionality would render many motion planning algorithms inapplicable for all but the most simple robots. For this reason, practical robot system often rely on simpler kinematic path planners.

A common technique to compensate for the limitations of kinematic plans is to use a separate mechanism, a controller, for keeping the robot on track. Controllers are techniques for generating robot controls in real time using feedback from the environment, so as to achieve a control objective. If the objective is to keep the robot on a preplanned path, it is often referred to as a reference controller and the path is called a reference path. Controllers that optimize a global cost function are known as optimal controllers. Optimal policies for continuous MDPs are, in effect, optimal controllers.

On the surface, the problem of keeping a robot on a prespecified path appears to be relatively straightforward. In practice, however, even this seemingly simple problem has its pitfalls. Figure 3.9(a) illustrates what can go wrong; it shows the path of a robot that attempts to follow a kinematic path. Whenever a deviation occurs—whether due to noise or to constraints on the forces the robot can apply—the robot provides an opposing force whose magnitude is proportional to this deviation. Intuitively, this might appear plausible, since deviations should be compensated by a counterforce to keep the robot on track. However, as Figure 3.9(a) illustrates, our controller causes the robot to vibrate rather violently. The vibration is the result of a natural inertia of the robot arm: once driven back to its reference position the robot then overshoots, which induces a symmetric error with opposite sign. Such overshooting may continue along an entire trajectory, and the resulting robot motion is far from desirable.

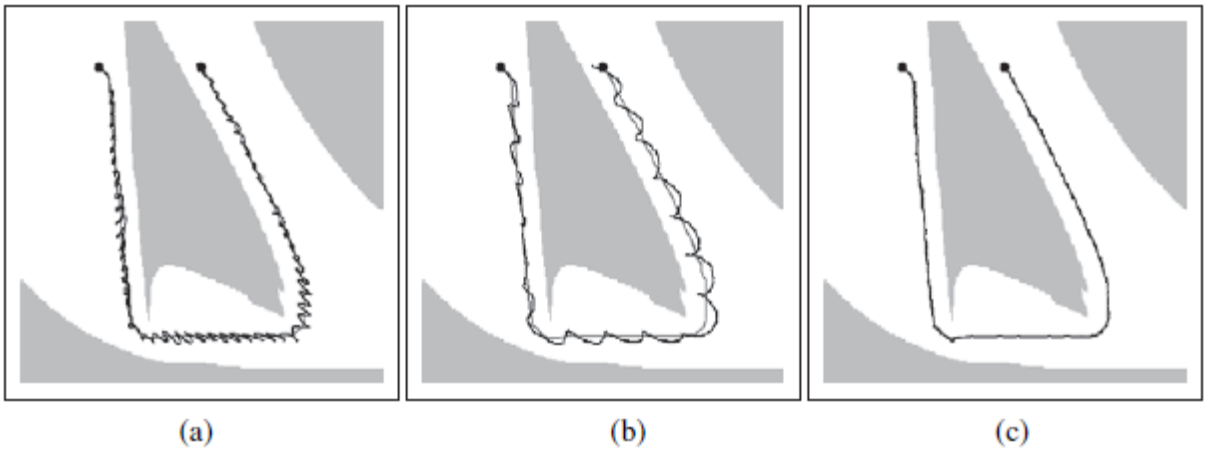


Figure 3.9 Robot arm control using (a) proportional control with gain factor 1.0, (b) proportional control with gain factor 0.1, and (c) PD (proportional derivative) control with gain factors 0.3 for the proportional component and 0.8 for the differential component. In all cases the robot arm tries to follow the path shown in gray.

Before we can define a better controller, let us formally describe what went wrong. Controllers that provide force in negative proportion to the observed error are known as P controllers. The letter 'P' stands for P proportional, indicating that the actual control is proportional to the error of the robot manipulator. More formally, let y(t) be the reference path, parameterized by time index t. The control at generated by a P controller has the form:

$a_t = K_P (y(t) - x_t)$. (1)

Here x_t is the state of the robot at time t and K_P is a constant known as the gain parameter of the controller and its value is called the gain factor); K_p regulates how strongly the controller corrects for deviations between the actual state x_t and the desired one y(t). In our example, KP = 1. At first glance, one might think that choosing a smaller value for KP would remedy the problem. Unfortunately, this is not the case. Figure 25.22(b) shows a trajectory for KP = .1, still exhibiting oscillatory behavior. Lower values of the gain parameter may simply slow down the oscillation, but do not solve the problem. In fact, in the absence of friction, the P controller is essentially a spring law; so it will oscillate indefinitely around a fixed target location.

Traditionally, problems of this type fall into the realm of control theory, a field of increasing importance to researchers in AI. Decades of research in this field have led to a large number of controllers that are superior to the simple control law given above. In particular, a reference controller is said to be stable if small perturbations lead to a bounded error between the robot and the reference signal. It is said to be strictly stable if it is able to return to and then stay on its reference path upon such perturbations. Our P controller appears to be stable but not strictly stable, since it fails to stay anywhere near its reference trajectory.

The simplest controller that achieves strict stability in our domain is a PD controller. The letter 'P' stands again for proportional, and 'D' stands for derivative. PD controllers are described by the following equation:

$$a_t = K_P(y(t) - x_t) + K_D \frac{\partial(y(t) - x_t)}{\partial t} \quad (2)$$

As this equation suggests, PD controllers extend P controllers by a differential component, which adds to the value of at a term that is proportional to the first derivative of the error $y(t) - x_t$ over time. What is the effect of such a term? In general, a derivative term dampens the system that is being controlled. To see this, consider a situation where the error $(y(t)-x_t)$ is changing rapidly over time, as is the case for our P controller above. The derivative of this error will then counteract the proportional term, which will reduce the overall response to the perturbation. However, if the same error persists and does not change, the derivative will vanish and the proportional term dominates the choice of control.

Figure 3.9(c) shows the result of applying this PD controller to our robot arm, using as gain parameters $K_P = .3$ and $K_D = .8$. Clearly, the resulting path is much smoother, and does not exhibit any obvious oscillations. PD controllers do have failure modes, however. In particular, PD controllers may fail to regulate an error down to zero, even in the absence of external perturbations. Often such a situation is the result of a systematic external force that is not part of the model. An autonomous car driving on a banked surface, for example, may find itself systematically pulled to one side. Wear and tear in robot arms cause similar systematic errors. In such situations, an over-proportional feedback is required to drive the error closer to zero. The solution to this problem lies in adding a third term to the control law, based on the integrated error over time:

$$a_t = K_P(y(t) - x_t) + K_I \int (y(t) - x_t)dt + K_D \frac{\partial(y(t) - x_t)}{\partial t} \quad (3)$$

Here K_I is yet another gain parameter. The term $\int (y(t) - x_t)dt$ calculates the integral of the error over time. The effect of this term is that long-lasting deviations between the reference signal and the actual state are corrected. If, for example, x_t is smaller than $y(t)$ for a long period of time, this integral will grow until the resulting control at forces this error to shrink.

Integral terms, then, ensure that a controller does not exhibit systematic error, at the expense of increased danger of oscillatory behavior. A controller with all three terms is called a PID controller (for proportional integral derivative). PID controllers are widely used in industry, for a variety of control problems.

3.3.2 Potential-field control

We introduced potential fields as an additional cost function in robot motion planning, but they can also be used for generating robot motion directly, dispensing with the path planning phase altogether. To achieve this, we have to define an attractive force that pulls the robot towards its goal configuration and a repellent potential field that pushes the robot away from obstacles. Such a potential field is shown in Figure 3.10. Its single global minimum is the goal configuration, and the value is the sum of the distance to this goal configuration and the proximity to obstacles. No planning was involved in generating the potential field shown in the figure. Because of this, potential fields are well suited to real-time control.

Figure 3.10(a) shows a trajectory of a robot that performs hill climbing in the potential field. In many applications, the potential field can be calculated efficiently for any given configuration. Moreover, optimizing the potential amounts to calculating the gradient of the potential for the present robot configuration. These calculations can be extremely efficient, especially when compared to path-planning algorithms, all of which are exponential in the dimensionality of the configuration space (the DOFs) in the worst case.

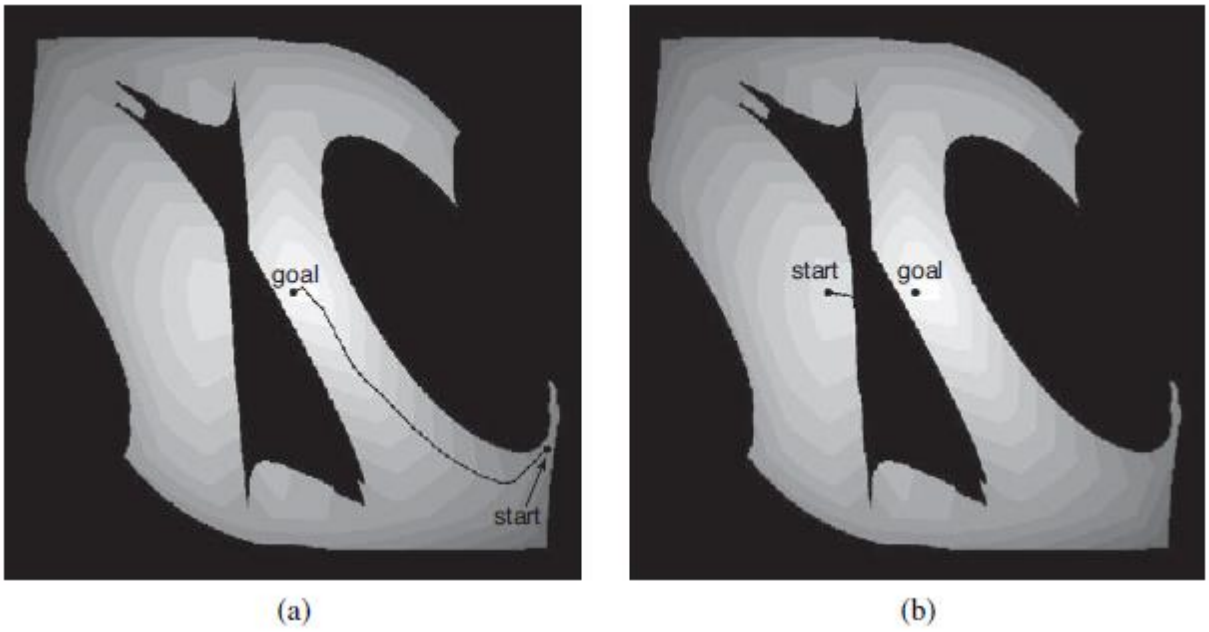


Figure 3.10 Potential field control. The robot ascends a potential field composed of repelling forces asserted from the obstacles and an attracting force that corresponds to the goal configuration. (a) Successful path. (b) Local optimum.

The fact that the potential field approach manages to find a path to the goal in such an efficient manner, even over long distances in configuration space, raises the question as to whether there is a need for planning in robotics at all. Are potential field techniques sufficient, or were we just lucky in our example? The answer is that we were indeed lucky.

Potential fields have many local minima that can trap the robot. In Figure 3.10(b), the robot approaches the obstacle by simply rotating its shoulder joint, until it gets stuck on the wrong side of the obstacle. The potential field is not rich enough to make the robot bend its elbow so that the arm fits under the obstacle. In other words, potential field control is great for local robot motion but sometimes we still need global planning. Another important drawback with potential fields is that the forces they generate depend only on the obstacle and robot positions, not on the robot's velocity. Thus, potential field control is really a kinematic method and may fail if the robot is moving quickly.

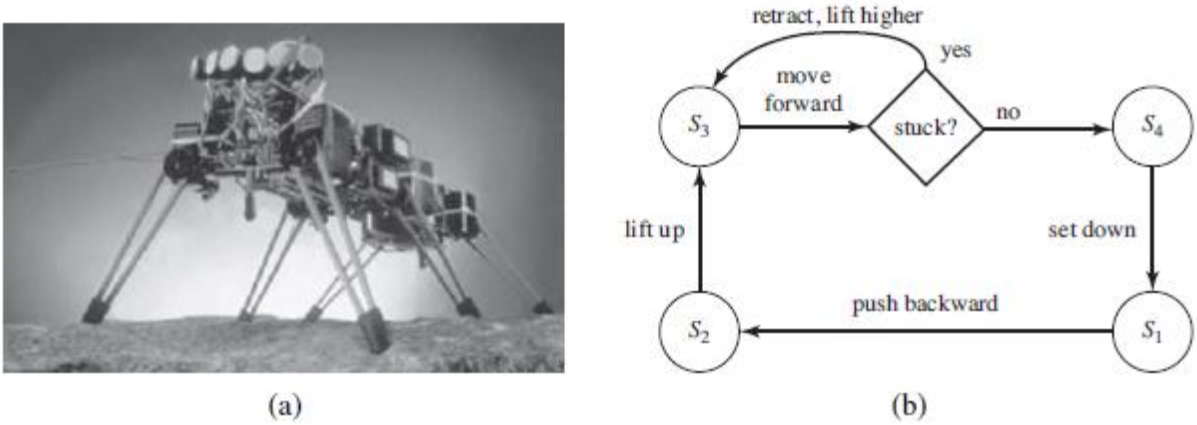


Figure 3.11 (a) Genghis, a hexapod robot. (b) An augmented finite state machine (AFSM) for the control of a single leg. Notice that this AFSM reacts to sensor feedback: if a leg is stuck during the forward swinging phase, it will be lifted increasingly higher.

3.3.3 Reactive control

So far we have considered control decisions that require some model of the environment for constructing either a reference path or a potential field. There are some difficulties with this approach. First, models that are sufficiently accurate are often difficult to obtain, especially in complex or remote environments, such as the surface of Mars, or for robots that have few sensors. Second, even in cases where we can devise a model with sufficient accuracy, computational difficulties and localization error might render these techniques impractical. In some cases, a reflex agent architecture using reactive control is more appropriate.

For example, picture a legged robot that attempts to lift a leg over an obstacle. We could give this robot a rule that says lift the leg a small height h and move it forward, and if the leg encounters an obstacle, move it back and start again at a higher height. You could say that his modeling an aspect of the world, but we can also think of h as an auxiliary variable of the robot controller, devoid of direct physical meaning.

One such example is the six-legged (hexapod) robot, shown in Figure 3.11(a), designed for walking through rough terrain. The robot's sensors are inadequate to obtain models of the terrain for path planning. Moreover, even if we added sufficiently accurate sensors, the twelve degrees of freedom (two for each leg) would render the resulting path planning problem computationally intractable. It is possible, nonetheless, to specify a controller directly without an explicit environmental model. (We have already seen this with the PD controller, which was able to keep a complex robot arm on target without an explicit model of the robot dynamics; it did, however, require a reference path generated from a kinematic model.) For the hexapod robot we first GAIT choose a gait, or pattern of movement of the limbs. One statically stable gait is to first move the right front, right rear, and left center legs forward (keeping the other three fixed), and then move the other three. This gait works well on flat terrain. On rugged terrain, obstacles may prevent a leg from swinging forward. This problem can be overcome by a remarkably simple control rule: when a leg's forward motion is blocked, simply retract it, lift it higher, and try again. The resulting controller is shown in Figure 3.11(b) as a finite state machine; it constitutes a

reflex agent with state, where the internal state is represented by the index of the current machine state (s1 through s4).

Figure 3.12 Multiple exposures of an RC helicopter executing a flip based on a policy learned with reinforcement learning. Images courtesy of Andrew Ng, Stanford University.

Variants of this simple feedback-driven controller have been found to generate remarkably robust walking patterns, capable of maneuvering the robot over rugged terrain. Clearly, such a controller is model-free, and it does not deliberate or use search for generating controls.

Environmental feedback plays a crucial role in the controller's execution. The software alone does not specify what will actually happen when the robot is placed in an environment. Behavior that emerges through the interplay of a (simple) controller and a (complex) environment is often referred to as emergent behavior. Strictly speaking, all robots discussed in this unit exhibit emergent behavior, due to the fact that no model is perfect. Historically, however, the term has been reserved for control techniques that do not utilize explicit environmental models. Emergent behavior is also characteristic of biological organisms.

3.3.4 Reinforcement learning control

One particularly exciting form of control is based on the policy search form of reinforcement learning. This work has been enormously influential in recent years, at is has solved challenging robotics problems for which previously no solution existed. An example is acrobatic autonomous helicopter flight. Figure 3.12 shows an autonomous flip of a small RC (radio-controlled) helicopter. This maneuver is challenging due to the highly nonlinear nature of the aerodynamics involved. Only the most experienced of human pilots are able to perform it. Yet a policy search method, using only a few minutes of computation, learned a policy that can safely execute a flip every time.

Policy search needs an accurate model of the domain before it can find a policy. The input to this model is the state of the helicopter at time t, the controls at time t, and the resulting state at time t+Δt. The state of a helicopter can be described by the 3D coordinates of the vehicle, its yaw, pitch, and roll angles, and the rate of change of these six variables.

The controls are the manual controls of the helicopter: throttle, pitch, elevator, aileron, and rudder. All that remains is the resulting state—how are we going to define a model that accurately says how the helicopter responds to each control? The answer is simple: Let an expert human pilot fly the helicopter, and record the controls that the expert transmits over the radio and the state variables of the helicopter. About four minutes of human-controlled flight suffices to build a predictive model that is sufficiently accurate to simulate the vehicle.

What is remarkable about this example is the ease with which this learning approach solves a challenging robotics problem. This is one of the many successes of machine learning in scientific fields previously dominated by careful mathematical analysis and modeling.

4.0 Conclusion

Robotics concerns itself with intelligent agents that manipulate the physical world. In this chapter, we have learned the following basics of robot planning:

- The planning of robot motion is usually done in configuration space, where each point specifies the location and orientation of the robot and its joint angles.
- Configuration space search algorithms include cell decomposition techniques, which decompose the space of all configurations into finitely many cells, and skeletonization techniques, which project configuration spaces onto lower-dimensional manifolds. The motion planning problem is then solved using search in these simpler structures.
- A path found by a search algorithm can be executed by using the path as the reference trajectory for a PID controller. Controllers are necessary in robotics to accommodate small perturbations; path planning alone is usually insufficient.

5.0 Summary

We hope you enjoyed this unit. This unit discourses the robot motion planning. Now, let us attempt the questions below.

6.0 Tutor Marked Assignment

i. Consider a mobile robot moving on a horizontal surface. Suppose that the robot can execute two kinds of motions:

- Rolling forward a specified distance.
- Rotating in place through a specified angle.

The state of such a robot can be characterized in terms of three parameters ‹ x, y, ϕ, the xcoordinate and y-coordinate of the robot (more precisely, of its center of rotation) and the robot's orientation expressed as the angle from the positive x direction. The action "Roll(D)" has the effect of changing state ‹ x, y, ϕ to ‹x + Dcos(ϕ), y + Dsin(ϕ), ϕ› , and the action Rotate(θ) has the effect of changing state ‹ x, y, ϕ to x, y, ϕ + θ ›.

a. Suppose that the robot is initially at 0, 0, 0 and then executes the actions Rotate(60◦), Roll(1), Rotate(25◦), Roll(2). What is the final state of the robot?

b. Now suppose that the robot has imperfect control of its own rotation, and that, if it attempts to rotate by θ, it may actually rotate by any angle between θ−10◦ and θ+10◦. In that case, if the robot attempts to carry out the sequence of actions in (A), there is a range of possible ending

states. What are the minimal and maximal values of the x-coordinate, the y-coordinate and the orientation in the final state?

c. Let us modify the model in (B) to a probabilistic model in which, when the robot attempts to rotate by θ, its actual angle of rotation follows a Gaussian distribution with mean θ and standard deviation 10◦. Suppose that the robot executes the actions Rotate(90◦), Roll(1). Give a simple argument that (a) the expected value of the location at the end is not equal to the result of rotating exactly 90◦ and then rolling forward 1 unit, and (b) that the distribution of locations at the end does not follow a Gaussian.

(Do not attempt to calculate the true mean or the true distribution.)

The point of this exercise is that rotational uncertainty quickly gives rise to a lot of positional uncertainty and that dealing with rotational uncertainty is painful, whether uncertainty is treated in terms of hard intervals or probabilistically, due to the fact that the relation between orientation and position is both non-linear and non-monotonic.

7.0 References/Further Readings

1) Russell, S., & Norvig, P. (2005). AI a modern approach. Learning, 2(3), 4.
2) Rich E., & Knight K. (1991) Artificial Intelligence, McGraw-Hill, New York, NY
3) Shi, Z. (2011). Advanced artificial intelligence (Vol. 1). World Scientific.
12. Rothman, D. (2018). Artificial Intelligence By Example: Develop machine intelligence Russell, S., & Norvig, P. (2005). AI a modern approach. Learning, 2(3), 4.
13. Rich E., & Knight K. (1991) Artificial Intelligence, McGraw-Hill, New York, NY
14. Shi, Z. (2011). Advanced artificial intelligence (Vol. 1). World Scientific.
4) Rothman, D. (2018). Artificial Intelligence By Example: Develop machine intelligence

MODULE 4: LAB. EXERCISES IN AI LANG.

Unit One: Getting Started
Unit Two: Searching for Solutions
Unit Three: Multiagent Systems
Unit Four: Scene Processing

Unit One: Getting Started

CONTENT

1.0 Introduction

We use Python because Python programs can be close to pseudo-code. It is designed for humans to read. Python is reasonably efficient. Efficiency is usually not a problem for small examples. If your Python code is not efficient enough, a general procedure to improve it is to find out what is taking most the time, and implement just that part more efficiently in some lower-level language. Most of these lowerlevel languages interoperate with Python nicely. This will result in much less programming and more efficient code (because you will have more time to optimize) than writing everything in a low-level language. You will not have to do that for the code here if you are using it for course projects.

2.0 Objectives

At the end of this unit, you should be able to:

- Install python and get it running

3.0 Main Content

3.1 Getting Python

You need Python 3 (http://python.org/) and matplotlib (http://matplotlib. org/) that runs with Python 3. This code is not compatible with Python 2 (e.g., with Python 2.7). Download and istall the latest Python 3 release from http://python.org/. This should also install pip3. You can install matplotlib using

pip3 install matplotlib

in a terminal shell (not in Python). That should "just work". If not, try using pip instead of pip3.

The command python or python3 should then start the interactive python shell. You can quit Python with a control-D or with quit().

To upgrade matplotlib to the latest version (which you should do if you

install a new version of Python) do:

pip3 install --upgrade matplotlib

We recommend using the enhanced interactive python ipython (http://ipython.org/). To install ipython after you have installed python do:

pip3 install ipython

3.2 Running Python

We assume that everything is done with an interactive Python shell. You can either do this with an IDE, such as IDLE that comes with standard Python distributions, or just running ipython3 (or perhaps just ipython) from a shell.

Here we describe the most simple version that uses no IDE. If you download the zip file, and cd to the "aipython" folder where the .py files are, you should be able to do the following, with user input following : . The first ipython3 command is in the operating system shell (note that the -i is important to enter interactive mode):

$ ipython3 -i searchGeneric.py
Python 3.6.5 (v3.6.5:f59c0932b4, Mar 28 2018, 05:52:31)
Type 'copyright', 'credits' or 'license' for more information
IPython 6.2.1 -- An enhanced Interactive Python. Type '?' for help.
Testing problem 1:
7 paths have been expanded and 4 paths remain in the frontier
Path found: a --> b --> c --> d --> g
Passed unit test

*In [1]: searcher2 = AStarSearcher(searchProblem.acyclic_delivery_problem) #A**

In [2]: searcher2.search() # find first path

16 paths have been expanded and 5 paths remain in the frontier

Out[2]: o103 --> o109 --> o119 --> o123 --> r123

In [3]: searcher2.search() # find next path

21 paths have been expanded and 6 paths remain in the frontier

Out[3]: o103 --> b3 --> b4 --> o109 --> o119 --> o123 --> r123

In [4]: searcher2.search() # find next path

28 paths have been expanded and 5 paths remain in the frontier

Out[4]: o103 --> b3 --> b1 --> b2 --> b4 --> o109 --> o119 --> o123 --> r123

In [5]: searcher2.search() # find next path

No (more) solutions. Total of 33 paths expanded.

In [6]:

You can then interact at the last prompt.

There are many textbooks for Python. The best source of information about python is https://www.python.org/. We will be using Python 3; please download the latest release. The documentation is at https://docs.python.org/3/.

This unit is about what is special about the code for AI tools. We will only use the Standard Python Library and matplotlib. All of the exercises can be done (and should be done) without using other libraries; the aim is for you to spend your time thinking about how to solve the problem rather than searching for pre-existing solutions.

3.3 Pitfalls

It is important to know when side effects occur. Often AI programs consider what would happen or what may have happened. In many such cases, we don't want side effects. When an agent acts in the world, side effects are appropriate.

In Python, you need to be careful to understand side effects. For example, the inexpensive function to add an element to a list, namely append, changes the list. In a functional language like Lisp, adding a new element to a list, without changing the original list, is a cheap operation. For example if x is a list containing n elements, adding an extra element to the list in Python (using append) is fast, but it has the side effect of changing the list x. To construct a new list that contains the elements of x plus a new element, without changing the value of x, entails copying the list, or using a different representation for lists. In the searching code, we will use a different representation for lists for this reason.

3.4 Features of Python

3.4.1 Lists, Tuples, Sets, Dictionaries and Comprehensions

We make extensive uses of lists, tuples, sets and dictionaries (dicts). See https://docs.python.org/3/library/stdtypes.html

One of the nice features of Python is the use of list comprehensions (and also tuple, set and dictionary comprehensions).

(fe for e in iter if cond)

enumerates the values fe for each e in iter for which cond is true. The “if cond” part is optional, but the “for” and “in” are not optional. Here e has to be a variable, iter is an iterator, which can generate a stream of data, such as a list, a set, a range object (to enumerate integers between ranges) or a file. Cond is an expression that evaluates to either True or False for each e, and fe is an expression that will be evaluated for each value of e for which cond returns

True.

The result can go in a list or used in another iteration, or can be called directly using next. The procedure next takes an iterator returns the next element (advancing the iterator) and raises a StopIteration exception if there is no next element. The following shows a simple example, where user input is prepended with >>>

```
>>> [e*e for e in range(20) if e%2==0]
[0, 4, 16, 36, 64, 100, 144, 196, 256, 324]
>>> a = (e*e for e in range(20) if e%2==0)
>>> next(a)
0
>>> next(a)
4
>>> next(a)
16
>>> list(a)
[36, 64, 100, 144, 196, 256, 324]
>>> next(a)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
StopIteration
```

Notice how list(a) continued on the enumeration, and got to the end of it.

Comprehensions can also be used for dictionaries. The following code creates an index for list a:

```
>>> a = ["a","f","bar","b","a","aaaaa"]
>>> ind = {a[i]:i for i in range(len(a))}
>>> ind
{'a': 4, 'f': 1, 'bar': 2, 'b': 3, 'aaaaa': 5}
>>> ind['b']
3
```

which means that 'b' is the 3rd element of the list.

The assignment of *ind* could have also be written as:

>>> ind = {val:i for (i,val) in enumerate(a)}

where enumerate returns an iterator of (index, value) pairs.

3.4.2 Functions as rst-class objects

Python can create lists and other data structures that contain functions. There is an issue that tricks many newcomers to Python. For a local variable in a function, the function uses the last value of the variable when the function is called, not the value of the variable when the function was defined (this is called “late binding”). This means if you want to use the value a variable has when the function is created, you need to save the current value of that variable.

Whereas Python uses “late binding” by default, the alternative that newcomers often expect is “early binding”, where a function uses the value a variable had when the function was defined, can be easily implemented.

Consider the following programs designed to create a list of 5 functions, where the ith function in the list is meant to add i to its argument:

pythonDemo.py — Some tricky examples

```
11 fun_list1 = []
12 for i in range(5):
13     def fun1(e):
14         return e+i
15     fun_list1.append(fun1)
16
17 fun_list2 = []
18 for i in range(5):
19     def fun2(e,iv=i):
20         return e+iv
21     fun_list2.append(fun2)
22
23 fun_list3 = [lambda e: e+i for i in range(5)]
24
25 fun_list4 = [lambda e,iv=i: e+iv for i in range(5)]
26
27 i=56
```

Try to predict, and then test to see the output, of the output of the following calls, remembering that the function uses the latest value of any variable that is not bound in the function call:

pythonDemo.py — (continued)

```
29 # in Shell do
30 ## ipython -i pythonDemo.py
31 # Try these (copy text after the comment symbol and paste in the Python prompt):
32 # print([f(10) for f in fun_list1])
33 # print([f(10) for f in fun_list2])
34 # print([f(10) for f in fun_list3])
35 # print([f(10) for f in fun_list4])
```

In the first for-loop, the function fun uses i, whose value is the last value it was assigned. In the second loop, the function fun2 uses iv. There is a separate iv variable for each function, and its value is the value of i when the function was defined. Thus fun1 uses late binding, and fun2 uses early binding. fun list3 and fun list4 are equivalent to the first two (except fun list4 uses a different I variable).

One of the advantages of using the embedded definitions (as in fun1 and fun2 above) over the lambda is that is it possible to add a __doc__ string, which is the standard for documenting functions in Python, to the embedded definitions.

3.4.3 Generators and Coroutines

Python has generators which can be used for a form of coroutines. The yield command returns a value that is obtained with next. It is typically used to enumerate the values for a for loop or in generators.

A version of the built-in range, with 2 or 3 arguments (and positive steps) can be implemented as:

pythonDemo.py — (continued)

```
37 def myrange(start, stop, step=1):
38     """enumerates the values from start in steps of size step that are
39     less than stop.
40     """
41     assert step>0, "only positive steps implemented in myrange"
42     i = start
43     while i<stop:
44         yield i
45         i += step
46
47 print("myrange(2,30,3):",list(myrange(2,30,3)))
```

Note that the built-in range is unconventional in how it handles a single argument, as the single argument acts as the second argument of the function.

Note also that the built-in range also allows for indexing (e.g., range(2, 30, 3)[2] returns 8), which the above implementation does not. However myrange also works for floats, which the built-in range does not.

Exercise 3.1 Implement a version of myrange that acts like the built-in version when there is a single argument. (Hint: make the second argument have a default value that can be recognized in the function.)

Yield can be used to generate the same sequence of values as in the example of Section 3.4.1:

pythonDemo.py — (continued)

```
49 def ga(n):
50     """generates square of even nonnegative integers less than n"""
51     for e in range(n):
52         if e%2==0:
53             yield e*e
54 a = ga(20)
```

The sequence of next(a), and list(a) gives exactly the same results as the comprehension in Section 1.5.1.

It is straightforward to write a version of the built-in enumerate. Let's call it myenumerate:

pythonDemo.py — (continued)

```
56 def myenumerate(enum):
57     for i in range(len(enum)):
58         yield i,enum[i]
```

Exercise 3.2 Write a version of enumerate where the only iteration is "for val in enum". Hint: keep track of the index.

4.0 Conclusion

In this unit how to install python and how to get it running is introduced.

5.0 Summary

We hope you enjoyed this unit. This unit treats searching for solutions with python. Now, let us attempt the questions below.

6.0 Tutor Marked Assignment

i. The exercises 3.1 and 3.2 represent the tutor marked assignment for this unit.

7.0 References/Further Readings

1) Artificial intelligence with Python Tutorials
2) Python code for Artificial Intelligence: Foundations of Computational Agents
3) Artificial Intelligence Foundations of Computational Agents

Unit Two: Searching for Solutions

CONTENT

1.0 Introduction

A search problem consists of:

- a start node
- a neighbor's function that given a node, returns an enumeration of the arcs from the node
- a specification of a goal in terms of a Boolean function that takes a node and returns true if the node is a goal
- a (optional) heuristic function that, given a node, returns a non-negative real number. The heuristic function defaults to zero.

As far as the searcher is concerned a node can be anything. If multiple-path pruning is used, a node must be hashable. In the simple examples, it is a string, but in more complicated examples (in later chapters) it can be a tuple, a frozen set, or a Python object.

2.0 Objectives

At the end of this unit you should be able to

- Represent search problems to return paths
- Implement a searcher
- Use priority queues to implement search algorithms
- Implement multiple path pruning

- implement branch-and-bound searches

3.0 Main Content

3.1 Representing Search Problems

In the following code raise NotImplementedError() is a way to specify that this is an abstract method that needs to be overridden to define an actual search problem.

searchProblem.py — representations of search problems

```
11 class Search_problem(object):
12     """A search problem consists of:
13     * a start node
14     * a neighbors function that gives the neighbors of a node
15     * a specification of a goal
16     * a (optional) heuristic function.
17     The methods must be overridden to define a search problem."""
18
19     def start_node(self):
20         """returns start node"""
21         raise NotImplementedError("start_node") # abstract method
22
23     def is_goal(self,node):
24         """is True if node is a goal"""
25         raise NotImplementedError("is_goal") # abstract method
26
27     def neighbors(self,node):
28         """returns a list of the arcs for the neighbors of node"""
29         raise NotImplementedError("neighbors") # abstract method
30
31     def heuristic(self,n):
32         """Gives the heuristic value of node n.
33         Returns 0 if not overridden."""
34         return 0
```

The neighbors is a list of arcs. A (directed) arc consists of a from node node and a to node node. The arc is the pair ⟨from node, to_node⟩, but can also contain a non-negative cost (which defaults to 1) and can be labeled with an action.

searchProblem.py — (continued)

```
class Arc(object):
    """An arc has a from_node and a to_node node and a (non-negative) cost"""
    def __init__(self, from_node, to_node, cost=1, action=None):
        assert cost >= 0, ("Cost cannot be negative for"+
                           str(from_node)+"->"+str(to_node)+", cost: "+str(cost))
        self.from_node = from_node
        self.to_node = to_node
        self.action = action
        self.cost=cost

    def __repr__(self):
        """string representation of an arc"""
        if self.action:
            return str(self.from_node)+" --"+str(self.action)+"--> "+str(self.to_node)
        else:
            return str(self.from_node)+" --> "+str(self.to_node)
```

3.1.1 Explicit Representation of Search Graph

The first representation of a search problem is from an explicit graph (as opposed to one that is generated as needed).

An explicit graph consists of

- a list or set of nodes
- a list or set of arcs
- a start node
- a list or set of goal nodes
- (optionally) a dictionary that maps a node to a heuristic value for that node

To define a search problem, we need to define the start node, the goal predicate, the neighbors function and the heuristic function.

searchProblem.py — (continued)

```
53 class Search_problem_from_explicit_graph(Search_problem):
54     """A search problem consists of:
55     * a list or set of nodes
56     * a list or set of arcs
57     * a start node
58     * a list or set of goal nodes
59     * a dictionary that maps each node into its heuristic value.
60     """
61
62     def __init__(self, nodes, arcs, start=None, goals=set(), hmap={}):
63         self.neighs = {}
64         self.nodes = nodes
65         for node in nodes:
66             self.neighs[node]=[]
67         self.arcs = arcs
68         for arc in arcs:
69             self.neighs[arc.from_node].append(arc)
70         self.start = start
71         self.goals = goals
72         self.hmap = hmap
73
74     def start_node(self):
75         """returns start node"""
76         return self.start
77
78     def is_goal(self,node):
79         """is True if node is a goal"""
80         return node in self.goals
81
82     def neighbors(self,node):
83         """returns the neighbors of node"""
84         return self.neighs[node]
85
86     def heuristic(self,node):
87         """Gives the heuristic value of node n.
88         Returns 0 if not overridden in the hmap."""
89         if node in self.hmap:
90             return self.hmap[node]
91         else:
92             return 0
```

```
93
94      def __repr__(self):
95          """returns a string representation of the search problem"""
96          res=""
97          for arc in self.arcs:
98              res += str(arc)+". "
99          return res
```

The following is used for the depth-first search implementation below.

searchProblem.py — (continued)

```
101     def neighbor_nodes(self,node):
102         """returns an iterator over the neighbors of node"""
103         return (path.to_node for path in self.neighs[node])
```

3.1.2 Paths

A searcher will return a path from the start node to a goal node. A Python list is not a suitable representation for a path, as many search algorithms consider multiple paths at once, and these paths should share initial parts of the path.

If we wanted to do this with Python lists, we would need to keep copying the list, which can be expensive if the list is long. An alternative representation is used here in terms of a recursive data structure that can share subparts.

A path is either:

- a node (representing a path of length 0) or
- a path, initial and an arc, where the from node of the arc is the node at the end of initial.

These cases are distinguished in the following code by having arc = None if the path has length 0, in which case initial is the node of the path.

searchProblem.py — (continued)

```
105 class Path(object):
106     """A path is either a node or a path followed by an arc"""
107
108     def __init__(self,initial,arc=None):
109         """initial is either a node (in which case arc is None) or
110         a path (in which case arc is an object of type Arc)"""
111         self.initial = initial
112         self.arc=arc
113         if arc is None:
114             self.cost=0
115         else:
116             self.cost = initial.cost+arc.cost
117
118     def end(self):
119         """returns the node at the end of the path"""
120         if self.arc is None:
121             return self.initial
122         else:
123             return self.arc.to_node
124
125     def nodes(self):
126         """enumerates the nodes for the path.
127         This starts at the end and enumerates nodes in the path backwards."""
128         current = self
129         while current.arc is not None:
130             yield current.arc.to_node
131             current = current.initial
132         yield current.initial
133
134     def initial_nodes(self):
135         """enumerates the nodes for the path before the end node.
136         This starts at the end and enumerates nodes in the path backwards."""
137         if self.arc is not None:
138             for nd in self.initial.nodes(): yield nd # could be "yield from"
139
140     def __repr__(self):
141         """returns a string representation of a path"""
142         if self.arc is None:
143             return str(self.initial)
144         elif self.arc.action:
145             return (str(self.initial)+"\n --"+str(self.arc.action)
146                     +"--> "+str(self.arc.to_node))
147         else:
148             return str(self.initial)+" --> "+str(self.arc.to_node)
```

3.1.3 Example Search Problems

The first search problem is one with 5 nodes where the least-cost path is one with many arcs. See Figure 3.1. Note that this example is used for the unit tests, so the test (in searchGeneric) will need to be changed if this is changed.

searchProblem.py — (continued)

```
150 problem1 = Search_problem_from_explicit_graph(
151     {'a','b','c','d','g'},
152     [Arc('a','b',1), Arc('a','c',3), Arc('b','d',3), Arc('b','c',1),
153         Arc('c','d',1), Arc('c','g',3), Arc('d','g',1)],
154     start = 'a',
155     goals = {'g'})
```

The second search problem is one with 8 nodes where many paths do not lead to the goal. See Figure 3.2.

searchProblem.py — (continued)

```
157 problem2 = Search_problem_from_explicit_graph(
158     {'a','b','c','d','e','g','h','j'},
159     [Arc('a','b',1), Arc('b','c',3), Arc('b','d',1), Arc('d','e',3),
```

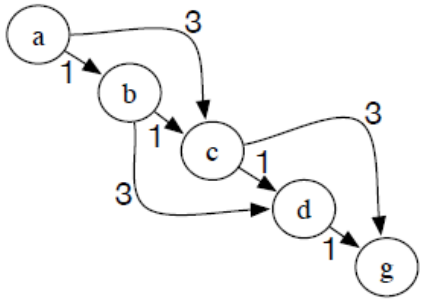


Figure 3.1: problem1

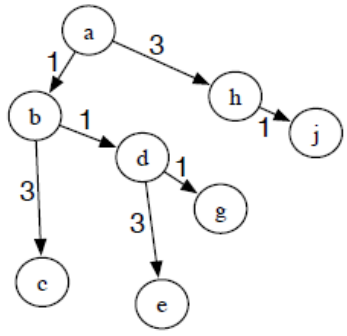


Figure 3.2: problem2

```
160         Arc('d','g',1), Arc('a','h',3), Arc('h','j',1)],
161     start = 'a',
162     goals = {'g'})
```

The third search problem is a disconnected graph (contains no arcs), where the start node is a goal node. This is a boundary case to make sure that weird cases work.

searchProblem.py — (continued)

```
164 problem3 = Search_problem_from_explicit_graph(
165     {'a','b','c','d','e','g','h','j'},
166     [],
167     start = 'g',
168     goals = {'k','g'})
```

3.2 Generic Searcher and Variants

To run the search demos, in folder “aipython”, load “searchGeneric.py” , using e.g., ipython -i searchGeneric.py, and copy and paste the example queries at the bottom of that file. This requires Python 3.

3.2.1 Searcher

A Searcher for a problem can be asked repeatedly for the next path. To solve a problem, we can construct a Searcher object for the problem and then repeatedly ask for the next path using search. If there are no more paths, None is returned.

searchGeneric.py — Generic Searcher, including depth-first and A*

```
11 from display import Displayable, visualize
12
13 class Searcher(Displayable):
14     """returns a searcher for a problem.
15     Paths can be found by repeatedly calling search().
16     This does depth-first search unless overridden
17     """
18     def __init__(self, problem):
19         """creates a searcher from a problem
20         """
21         self.problem = problem
22         self.initialize_frontier()
23         self.num_expanded = 0
24         self.add_to_frontier(Path(problem.start_node()))
25         super().__init__()
26
27     def initialize_frontier(self):
28         self.frontier = []
29
30     def empty_frontier(self):
31         return self.frontier == []
32
33     def add_to_frontier(self,path):
34         self.frontier.append(path)
35
36     @visualize
```

```
37     def search(self):
38         """returns (next) path from the problem's start node
39         to a goal node.
40         Returns None if no path exists.
41         """
42         while not self.empty_frontier():
43             path = self.frontier.pop()
44             self.display(2, "Expanding:",path,"(cost:",path.cost,")")
45             self.num_expanded += 1
46             if self.problem.is_goal(path.end()): # solution found
47                 self.display(1, self.num_expanded, "paths have been expanded and",
48                              len(self.frontier), "paths remain in the frontier")
49                 self.solution = path # store the solution found
50                 return path
51             else:
52                 neighs = self.problem.neighbors(path.end())
53                 self.display(3,"Neighbors are", neighs)
54                 for arc in reversed(neighs):
```

```
55                  self.add_to_frontier(Path(path,arc))
56              self.display(3,"Frontier:",self.frontier)
57          self.display(1,"No (more) solutions. Total of",
58                       self.num_expanded,"paths expanded.")
```

Note that this reverses the neigbours so that it implements depth-first search in an intutive manner (expanding the first neighbor first). This might not be required for other methods.

Exercise 3.1 When it returns a path, the algorithm can be used to find another path by calling search() again. However, it does not find other paths that go through one goal node to another. Explain why, and change the code so that it can find such paths when search() is called again.

3.2.2 Frontier as a Priority Queue

In many of the search algorithms, such as A* and other best-first searchers, the frontier is implemented as a priority queue. Here we use the Python's built-in priority queue implementations, heapq.

Following the lead of the Python documentation, http://docs.python.org/3.3/library/heapq.html, a frontier is a list of triples. The first element of each triple is the value to be minimized. The second element is a unique index which specifies the order when the first elements are the same, and the third element is the path that is on the queue. The use of the unique index ensures that the priority queue implementation does not compare paths; whether one path is less than another is not defined. It also lets us control what sort of search (e.g., depth-first or breadth-first) occurs when the value to be minimized does not give a unique next path.

The variable frontier index is the total number of elements of the frontier that have been created. As well as being used as a unique index, it is useful for statistics, particularly in conjunction with the current size of the frontier.

searchGeneric.py — (continued)

```
60  import heapq      # part of the Python standard library
61  from searchProblem import Path
62
63  class FrontierPQ(object):
64      """A frontier consists of a priority queue (heap), frontierpq, of
65          (value, index, path) triples, where
66      * value is the value we want to minimize (e.g., path cost + h).
67      * index is a unique index for each element
68      * path is the path on the queue
69      Note that the priority queue always returns the smallest element.
70      """
71
72      def __init__(self):
73          """constructs the frontier, initially an empty priority queue
74          """
75          self.frontier_index = 0 # the number of items ever added to the frontier
```

```
76          self.frontierpq = [] # the frontier priority queue
77
78      def empty(self):
79          """is True if the priority queue is empty"""
80          return self.frontierpq == []
81
82      def add(self, path, value):
83          """add a path to the priority queue
84          value is the value to be minimized"""
85          self.frontier_index += 1 # get a new unique index
86          heapq.heappush(self.frontierpq,(value, -self.frontier_index, path))
87
88      def pop(self):
89          """returns and removes the path of the frontier with minimum value.
90          """
91          (_,_,path) = heapq.heappop(self.frontierpq)
92          return path
```

The following methods are used for finding and printing information about the frontier.

searchGeneric.py — (continued)

```
94      def count(self,val):
95          """returns the number of elements of the frontier with value=val"""
96          return sum(1 for e in self.frontierpq if e[0]==val)
97
98      def __repr__(self):
99          """string representation of the frontier"""
100         return str([(n,c,str(p)) for (n,c,p) in self.frontierpq])
101
102     def __len__(self):
103         """length of the frontier"""
104         return len(self.frontierpq)
105
106     def __iter__(self):
107         """iterate through the paths in the frontier"""
108         for (_,_,path) in self.frontierpq:
109             yield path
```

3.2.3 A* Search

For an A* Search the frontier is implemented using the FrontierPQ class.

searchGeneric.py — (continued)

```
111 class AStarSearcher(Searcher):
112     """returns a searcher for a problem.
113     Paths can be found by repeatedly calling search().
114     """
115
116     def __init__(self, problem):
```

```
117         super().__init__(problem)
118
119     def initialize_frontier(self):
120         self.frontier = FrontierPQ()
121
122     def empty_frontier(self):
123         return self.frontier.empty()
124
125     def add_to_frontier(self,path):
126         """add path to the frontier with the appropriate cost"""
127         value = path.cost+self.problem.heuristic(path.end())
128         self.frontier.add(path, value)
```

Testing:

searchGeneric.py — (continued)

```
130 import searchProblem as searchProblem
131
132 def test(SearchClass):
133     print("Testing problem 1:")
134     schr1 = SearchClass(searchProblem.problem1)
135     path1 = schr1.search()
136     print("Path found:",path1)
137     assert list(path1.nodes()) == ['g','d','c','b','a'], "Shortest path not found in problem1"
138     print("Passed unit test")
139
140 if __name__ == "__main__":
141     #test(Searcher)
142     test(AStarSearcher)
143
144 # example queries:
145 # searcher1 = Searcher(searchProblem.acyclic_delivery_problem) # DFS
146 # searcher1.search() # find first path
147 # searcher1.search() # find next path
148 # searcher2 = AStarSearcher(searchProblem.acyclic_delivery_problem) # A*
149 # searcher2.search() # find first path
150 # searcher2.search() # find next path
151 # searcher3 = Searcher(searchProblem.cyclic_delivery_problem) # DFS
152 # searcher3.search() # find first path with DFS. What do you expect to happen?
153 # searcher4 = AStarSearcher(searchProblem.cyclic_delivery_problem) # A*
154 # searcher4.search() # find first path
```

Exercise 3.2 Change the code so that it implements (i) best-first search and (ii) lowest-cost-first search. For each of these methods compare it to A* in terms of the number of paths expanded, and the path found.

Exercise 3.3 In the add method in FrontierPQ what does the ”-” in front of frontier index do? When there are multiple paths with the same f -value, which search method does this act like? What happens if the ”-” is removed? When there are multiple paths with the same value, which search method does

this act like? Does it work better with or without the ”-”? What evidence did you base your conclusion on?

3.2.4 Multiple Path Pruning

To run the multiple-path pruning demo, in folder “aipython”, load “searchMPP.py” , using e.g., ipython -i searchMPP.py, and copy and paste the example queries at the bottom of that file. The following implements A* with multiple-path pruning. It overrides search() in Searcher.

searchMPP.py — Searcher with multiple-path pruning

```
11 from searchGeneric import AStarSearcher, visualize
12 from searchProblem import Path
13
14 class SearcherMPP(AStarSearcher):
15     """returns a searcher for a problem.
16     Paths can be found by repeatedly calling search().
17     """
18     def __init__(self, problem):
19         super().__init__(problem)
20         self.explored = set()
21
22     @visualize
23     def search(self):
24         """returns next path from an element of problem's start nodes
25         to a goal node.
26         Returns None if no path exists.
27         """
28         while not self.empty_frontier():
29             path = self.frontier.pop()
30             if path.end() not in self.explored:
31                 self.display(2, "Expanding:",path,"(cost:",path.cost,")")
32                 self.explored.add(path.end())
33                 self.num_expanded += 1
34                 if self.problem.is_goal(path.end()):
35                     self.display(1, self.num_expanded, "paths have been expanded and",
36                                  len(self.frontier), "paths remain in the frontier")
37                     self.solution = path # store the solution found
38                     return path
39                 else:
40                     neighs = self.problem.neighbors(path.end())
41                     self.display(3,"Neighbors are", neighs)
42                     for arc in neighs:
43                         self.add_to_frontier(Path(path,arc))
44                     self.display(3,"Frontier:",self.frontier)
```

```
45          self.display(1,"No (more) solutions. Total of",
46                       self.num_expanded,"paths expanded.")
47
48  from searchGeneric import test
49  if __name__ == "__main__":
50      test(SearcherMPP)
51
52  import searchProblem
53  # searcherMPPcdp = SearcherMPP(searchProblem.cyclic_delivery_problem)
54  # print(searcherMPPcdp.search()) # find first path
```

Exercise 3.5 Implement a searcher that implements cycle pruning instead of multiple-path pruning. You need to decide whether to check for cycles when paths are added to the frontier or when they are removed. (Hint: either method can be implemented by only changing one or two lines in SearcherMPP.) Compare no pruning, multiple path pruning and cycle pruning for the cyclic delivery problem. Which works better in terms of number of paths expanded, computational time or space?

3.3 Branch-and-bound Search

To run the demo, in folder “aipython”, load “searchBranchAndBound.py”, and copy and paste the example queries at the bottom of that file.

Depth-first search methods do not need an a priority queue, but can use a list as a stack. In this implementation of branch-and-bound search, we call search to find an optimal solution with cost less than bound. This uses depthfirst search to find a path to a goal that extends path with cost less than the bound. Once a path to a goal has been found, that path is remembered as the best path, the bound is reduced, and the search continues.

searchBranchAndBound.py — Branch and Bound Search

```
11  from searchProblem import Path
12  from searchGeneric import Searcher
13  from display import Displayable, visualize
14
15  class DF_branch_and_bound(Searcher):
16      """returns a branch and bound searcher for a problem.
17      An optimal path with cost less than bound can be found by calling search()
18      """
19      def __init__(self, problem, bound=float("inf")):
20          """creates a searcher than can be used with search() to find an optimal path.
21          bound gives the initial bound. By default this is infinite - meaning there
22          is no initial pruning due to depth bound
23          """
24          super().__init__(problem)
25          self.best_path = None
```

```
        self.bound = bound

    @visualize
    def search(self):
        """returns an optimal solution to a problem with cost less than bound.
        returns None if there is no solution with cost less than bound."""
        self.frontier = [Path(self.problem.start_node())]
        self.num_expanded = 0
        while self.frontier:
            path = self.frontier.pop()
            if path.cost+self.problem.heuristic(path.end()) < self.bound:
                self.display(3,"Expanding:",path,"cost:",path.cost)
                self.num_expanded += 1
                if self.problem.is_goal(path.end()):
                    self.best_path = path
                    self.bound = path.cost
                    self.display(2,"New best path:",path," cost:",path.cost)
                else:
                    neighs = self.problem.neighbors(path.end())
                    self.display(3,"Neighbors are", neighs)
                    for arc in reversed(list(neighs)):
                        self.add_to_frontier(Path(path, arc))
        self.display(1,"Number of paths expanded:",self.num_expanded)
        self.solution = self.best_path
        return self.best_path
```

Note that this code used reversed in order to expand the neighbors of a node in the left-to-right order one might expect. It does this because pop() removes the rightmost element of the list. Note that reversed only works on lists and tuples, but the neighbours can be generated. Here is a unit test and some queries:

searchBranchAndBound.py — (continued)

```
from searchGeneric import test
if __name__ == "__main__":
    test(DF_branch_and_bound)

# Example queries:
import searchProblem
# searcherb1 = DF_branch_and_bound(searchProblem.acyclic_delivery_problem)
# print(searcherb1.search())    # find optimal path
# searcherb2 = DF_branch_and_bound(searchProblem.cyclic_delivery_problem, bound=100)
# print(searcherb2.search())    # find optimal path
```

Exercise 3.6 Implement a branch-and-bound search uses recursion. Hint: you don't need an explicit frontier, but can do a recursive call for the children.

Exercise 3.7 After the branch-and-bound search found a solution, Sam ran search again, and noticed a different count. Sam hypothesized that this count was related to the number of nodes that an A* search

would use (either expand or be added to the frontier). Or maybe, Sam thought, the count for a number of nodes when the bound is slightly above the optimal path case is related to how A* would work.

Is there relationship between these counts? Are there different things that it could count so they are related? Try to find the most specific statement that is true, and explain why it is true.

To test the hypothesis, Sam wrote the following code, but isn't sure it is helpful:

searchTest.py — code that may be useful to compare A* and branch-and-bound

```
from searchGeneric import Searcher, AStarSearcher
from searchBranchAndBound import DF_branch_and_bound
from searchMPP import SearcherMPP

DF_branch_and_bound.max_display_level = 1
Searcher.max_display_level = 1

def run(problem,name):
    print("\n\n*******",name)

    print("\nA*:")
    asearcher = AStarSearcher(problem)
    print("Path found:",asearcher.search()," cost=",asearcher.solution.cost)
    print("there are",asearcher.frontier.count(asearcher.solution.cost),
          "elements remaining on the queue with f-value=",asearcher.solution.cost)

    print("\nA* with MPP:"),
    msearcher = SearcherMPP(problem)
    print("Path found:",msearcher.search()," cost=",msearcher.solution.cost)
    print("there are",msearcher.frontier.count(msearcher.solution.cost),
          "elements remaining on the queue with f-value=",msearcher.solution.cost)


   bound = asearcher.solution.cost+0.01
   print("\nBranch and bound (with too-good initial bound of", bound,")")
   tbb = DF_branch_and_bound(problem,bound) # cheating!!!!
   print("Path found:",tbb.search()," cost=",tbb.solution.cost)
   print("Rerunning B&B")
   print("Path found:",tbb.search())

   bbound = asearcher.solution.cost*2+10
   print("\nBranch and bound (with not-very-good initial bound of", bbound, ")")
   tbb2 = DF_branch_and_bound(problem,bbound) # cheating!!!!
   print("Path found:",tbb2.search()," cost=",tbb2.solution.cost)
   print("Rerunning B&B")
   print("Path found:",tbb2.search())

   print("\nDepth-first search: (Use ^C if it goes on forever)")
   tsearcher = Searcher(problem)
   print("Path found:",tsearcher.search()," cost=",tsearcher.solution.cost)


import searchProblem
```

```
53 from searchTest import run
54 if __name__ == "__main__":
55     run(searchProblem.problem1,"Problem 1")
56 #   run(searchProblem.acyclic_delivery_problem,"Acyclic Delivery")
57 #   run(searchProblem.cyclic_delivery_problem,"Cyclic Delivery")
58 # also test some graphs with cycles, and some with multiple least-cost paths
```

4.0 Conclusion

In this unit we have covered how search problems are represented how paths are returned. We also explained how a searcher is implemented, how priority queues are used to implement search algorithms. You will have to do more hands on in order to get acquainted with the language and do more advanced projects.

5.0 Summary

We hope you enjoyed this unit. This unit treats searching for solutions with python. Now, let us attempt the questions below.

6.0 Tutor Marked Assignment

i. The exercises 3.1 to 3.7 represent the tutor marked assignment for this unit.

7.0 References/Further Readings

1) Joshi, P. (2017). Artificial intelligence with python. Packt Publishing Ltd.
2) Poole, D. L., & Mackworth, A. K. (2017). Python code for Artificial Intelligence: Foundations of Computational Agents.
3) Poole, D. L., & Mackworth, A. K. (2010). Artificial Intelligence: foundations of computational agents. Cambridge University Press.

Unit Three: Multiagent Systems

CONTENT

1.0 Introduction

A multiagent system is a computerized system composed of multiple interacting intelligent agents. They can solve problems that are difficult or impossible for an individual agent or a monolithic system.

2.0 Objectives

At the end of this unit, you should be able to:

- Create a multiagent scenario in python
- Implement a naïve depth-first minimax algorithm
- Implement the depth-first minimax algorithm with ∝-β Pruning

3.0 Main Content

3.1 Minimax

Here we consider two-player zero-sum games. Here a player only wins when another player loses. This can be modeled as where there is a single utility which one agent (the maximizing agent) is trying to minimize and the other agent (the minimizing agent) is trying to minimize.

3.1.1 Creating a two-player game

masProblem.py — A Multiagent Problem

```
from display import Displayable

class Node(Displayable):
    """A node in a search tree. It has a
    name a string
    isMax is True if it is a maximizing node, otherwise it is minimizing node
    children is the list of children
    value is what it evaluates to if it is a leaf.
    """
    def __init__(self, name, isMax, value, children):
        self.name = name
        self.isMax = isMax
        self.value = value
        self.allchildren = children

    def isLeaf(self):
        """returns true of this is a leaf node"""
        return self.allchildren is None


    def children(self):
        """returns the list of all children."""
        return self.allchildren

    def evaluate(self):
        """returns the evaluation for this node if it is a leaf"""
        return self.value
```

The following is a representation of a magic-sum game, where players take turns picking a number in the range [1, 9], and the first player to have 3 numbers that sum to 15 wins. Note that this is a syntactic variant of tic-tac-toe or naughts and crosses. To see this, consider the numbers on a magic square (Figure 3.1); 3 numbers that add to 15 correspond exactly to the winning positions of tic-tac-toe played on the magic square. Note that we do not remove symmetries. (What are the symmetries? How do the symmetries of tic-tac-toe translate here?)

6	1	8
7	5	3
2	9	4

Figure 3.1: Magic Square

masProblem.py — (continued)

```
70
71  class Magic_sum(Node):
72      def __init__(self, xmove=True, last_move=None,
73                   available=[1,2,3,4,5,6,7,8,9], x=[], o=[]):
74          """This is a node in the search for the magic-sum game.
75          xmove is True if the next move belongs to X.
76          last_move is the number selected in the last move
77          available is the list of numbers that are available to be chosen
78          x is the list of numbers already chosen by x
79          o is the list of numbers already chosen by o
80          """
81          self.isMax = self.xmove = xmove
82          self.last_move = last_move
83          self.available = available
84          self.x = x
85          self.o = o
86          self.allchildren = None #computed on demand
87          lm = str(last_move)
88          self.name = "start" if not last_move else "o="+lm if xmove else "x="+lm
89
90      def children(self):
91          if self.allchildren is None:
92              if self.xmove:
93                  self.allchildren = [
94                      Magic_sum(xmove = not self.xmove,
95                                last_move = sel,
96                                available = [e for e in self.available if e is not sel],
97                                x = self.x+[sel],
98                                o = self.o)
99                               for sel in self.available]
100             else:
101                 self.allchildren = [
102                     Magic_sum(xmove = not self.xmove,
103                               last_move = sel,
104                               available = [e for e in self.available if e is not sel],
105                               x = self.x,
106                               o = self.o+[sel])
107                              for sel in self.available]
108         return self.allchildren
109
```

```
110     def isLeaf(self):
111         """A leaf has no numbers available or is a win for one of the players.
112         We only need to check for a win for o if it is currently x's turn,
113         and only check for a win for x if it is o's turn (otherwise it would
114         have been a win earlier).
115         """
116         return (self.available == [] or
117                 (sum_to_15(self.last_move,self.o)
118                  if self.xmove
119                  else sum_to_15(self.last_move,self.x)))
120 
121     def evaluate(self):
122         if self.xmove and sum_to_15(self.last_move,self.o):
123             return -1
124         elif not self.xmove and sum_to_15(self.last_move,self.x):
125             return 1
126         else:
127             return 0
128 
129 def sum_to_15(last,selected):
130     """is true if last, toegether with two other elements of selected sum to 15.
131     """
132     return any(last+a+b == 15
133                for a in selected if a != last
134                for b in selected if b != last and b != a)
```

3.1.2 Minimax and ∝-β Pruning

This is a naive depth-first minimax algorithm:

masMiniMax.py — Minimax search with alpha-beta pruning

```
11 def minimax(node):
12     """returns the value of node, and a best path for the agents
13     """
14     if node.isLeaf():
15         return node.evaluate(),None
16     elif node.isMax:
17         max_score = -999
18         max_path = None
19         for C in node.children():
20             score,path = minimax(C,depth+1)
21             if score > max_score:
22                 max_score = score
23                 max_path = C.name,path
24         return max_score,max_path
25     else:
26         min_score = 999
27         min_path = None
28         for C in node.children():
29             score,path = minimax(C,depth+1)
30             if score < min_score:
31                 min_score = score
32                 min_path = C.name,path
33         return min_score,min_path
```

The following is a depth-first minimax with ∝-β Pruning. It returns the value for a node as well as a best path for the agents.

masMiniMax.py — (continued)

```
35 def minimax_alpha_beta(node,alpha,beta,depth=0):
36     """node is a Node, alpha and beta are cutoffs, depth is the depth
37     returns value, path
38     where path is a sequence of nodes that results in the value"""
39     node.display(2," "*depth,"minimax_alpha_beta(",node.name,", ",alpha, ", ", beta,")")
40     best=None     # only used if it will be pruned
41     if node.isLeaf():
42         node.display(2," "*depth,"returning leaf value",node.evaluate())
43         return node.evaluate(),None
44     elif node.isMax:
45         for C in node.children():
46             score,path = minimax_alpha_beta(C,alpha,beta,depth+1)
47             if score >= beta:  # beta pruning
48                 node.display(2," "*depth,"pruned due to beta=",beta,"C=",C.name)
49                 return score, None
50             if score > alpha:
51                 alpha = score
52                 best = C.name, path
53         node.display(2," "*depth,"returning max alpha",alpha,"best",best)
54         return alpha,best
55     else:
56         for C in node.children():
57             score,path = minimax_alpha_beta(C,alpha,beta,depth+1)
58             if score <= alpha:   # alpha pruning
59                 node.display(2," "*depth,"pruned due to alpha=",alpha,"C=",C.name)
60                 return score, None
61             if score < beta:
62                 beta=score
63                 best = C.name,path
64         node.display(2," "*depth,"returning min beta",beta,"best=",best)
65         return beta,best
```

Testing:

masMiniMax.py — (continued)

```
67 from masProblem import fig10_5, Magic_sum, Node
68
69 # Node.max_display_level=2 # print detailed trace
70 # minimax_alpha_beta(fig10_5, -9999, 9999,0)
71 # minimax_alpha_beta(Magic_sum(), -9999, 9999,0)
72
73 #To see how much time alpha-beta pruning can save over minimax, uncomment the following:
74 ## import timeit
75 ## timeit.Timer("minimax(Magic_sum())",setup="from __main__ import minimax, Magic_sum"
```

```
76 ##            ).timeit(number=1)
77 ## trace=False
78 ## timeit.Timer("minimax_alpha_beta(Magic_sum(), -9999, 9999,0)",
79 ##            setup="from __main__ import minimax_alpha_beta, Magic_sum"
80 ##            ).timeit(number=1)
```

4.0 Conclusion

In this unit we have introduced how a maximizing agent can be implemented, naïve depth-first minimax with ∝-β Pruning is also implemented. You will have to do more hands on in order to get acquainted with the language and do more advanced projects.

5.0 Summary

We hope you enjoyed this unit. This unit treats multiagent system implantation with python. Now, let us attempt the questions below.

6.0 Tutor Marked Assignment

i. Write a python code to Play a Tic Tac Toe game.

7.0 References/Further Readings

1) Joshi, P. (2017). Artificial intelligence with python. Packt Publishing Ltd.
2) Poole, D. L., & Mackworth, A. K. (2017). Python code for Artificial Intelligence: Foundations of Computational Agents.
3) Poole, D. L., & Mackworth, A. K. (2010). Artificial Intelligence: foundations of computational agents. Cambridge University Press.

Unit Four: Scene Processing

CONTENT

1.0 Introduction

Computer vision is concerned with modeling and replicating human vision using computer software and hardware. In this chapter, you will learn in detail about this.

2.0 Objectives

At the end of this unit you should be able to:

- Read, write and display an image using python
- Implement Edge detection in python
- Implement Face detection in python
- Implement Eye detection in python

3.0 Main Content

3.1 Computer Vision

Computer vision is a discipline that studies how to reconstruct, interrupt and understand a 3d scene from its 2d images, in terms of the properties of the structure present in the scene.

3.1.1 Computer Vision Hierarchy

Computer vision is divided into three basic categories as following:

- Low-level vision: It includes process image for feature extraction.
- Intermediate-level vision: It includes object recognition and 3D scene interpretation
- High-level vision: It includes conceptual description of a scene like activity, intention and behavior.

3.1.2 Computer Vision Vs Image Processing

Image processing studies image to image transformation. The input and output of image processing are both images.

Computer vision is the construction of explicit, meaningful descriptions of physical objects from their image. The output of computer vision is a description or an interpretation of structures in 3D scene.

3.2 Installing Useful Packages

For Computer vision with Python, you can use a popular library called OpenCV (Open Source Computer Vision). It is a library of programming functions mainly aimed at the real-time computer vision. It is written in C++ and its primary interface is in C++. You can install this package with the help of the following command:

pip install opencv_python-X.X-cp36-cp36m-winX.whl

Here X represents the version of Python installed on your machine as well as the win32 or 64 bit you are having.

If you are using the anaconda environment, then use the following command to install OpenCV:

conda install -c conda-forge opencv

3.3 Reading, Writing and Displaying an Image

Most of the CV applications need to get the images as input and produce the images as output. In this section, you will learn how to read and write image file with the help of functions provided by OpenCV.

3.3.1 OpenCV functions for Reading, Showing, Writing an Image File

OpenCV provides the following functions for this purpose:

- *imread()* function: This is the function for reading an image. OpenCV imread() supports various image formats like PNG, JPEG, JPG, TIFF, etc.
- *imshow()* function: This is the function for showing an image in a window. The window automatically fits to the image size. OpenCV imshow() supports various image formats like PNG, JPEG, JPG, TIFF, etc.
- *imwrite()* function: This is the function for writing an image. OpenCV imwrite() supports various image formats like PNG, JPEG, JPG, TIFF, etc.

Example

This example shows the Python code for reading an image in one format: showing it in a window and writing the same image in other format. Consider the steps shown below:

Import the OpenCV package as shown:

import cv2

Now, for reading a particular image, use the imread() function:

image = cv2.imread('image_flower.jpg')

For showing the image, use the imshow() function. The name of the window in which you can see the image would be image_flower.

cv2.imshow('image_flower',image)

cv2.destroyAllwindows()

Now, we can write the same image into the other format, say .png by using the imwrite() function:

cv2.imwrite('image_flower.png',image)

The output True means that the image has been successfully written as .png file also in the same folder.

True

Note: The function destroyallWindows() simply destroys all the windows we created.

3.4 Color Space Conversion

In OpenCV, the images are not stored by using the conventional RGB color, rather they are stored in the reverse order i.e. in the BGR order. Hence the default color code while reading an image is BGR. The cvtColor() color conversion function in for converting the image from one color code to other.

Example

Consider this example to convert image from BGR to grayscale.

Import the OpenCV package as shown:

import cv2

Now, for reading a particular image, use the imread() function:

image = cv2.imread('image_flower.jpg')

Now, if we see this image using imshow() function, then we can see that this image is in BGR.

cv2.imshow('BGR_Penguins',image)

Now, use cvtColor() function to convert this image to grayscale.

image = cv2.cvtColor(image,cv2.COLOR_BGR2GRAY)

cv2.imshow('gray_penguins',image)

3.5 Edge Detection

Humans, after seeing a rough sketch, can easily recognize many object types and their poses. That is why edges play an important role in the life of humans as well as in the applications of computer vision. OpenCV provides very simple and useful function called Canny()for detecting the edges.

Example

The following example shows clear identification of the edges.

Import OpenCV package as shown:

import cv2

import numpy as np

Now, for reading a particular image, use the imread() function.

image = cv2.imread('Penguins.jpg')

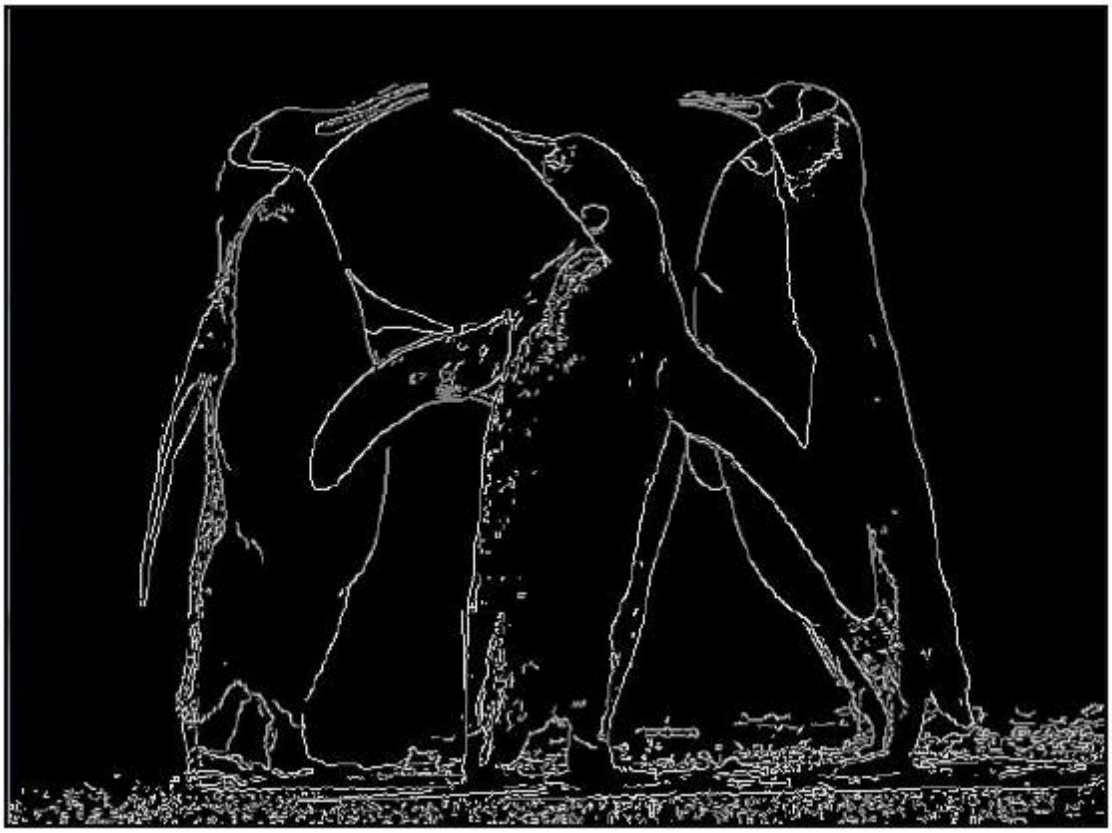
Now, use the Canny () function for detecting the edges of the already read image.

cv2.imwrite('edges_Penguins.jpg',cv2.Canny(image,200,300))

Now, for showing the image with edges, use the imshow() function.

cv2.imshow('edges', cv2.imread(''edges_Penguins.jpg'))

This Python program will create an image named edges_penguins.jpg with edge detection.



3.6 Face Detection

Face detection is one of the fascinating applications of computer vision which makes it more realistic as well as futuristic. OpenCV has a built-in facility to perform face detection. We are going to use the Haar cascade classifier for face detection.

Haar Cascade Data

We need data to use the Haar cascade classifier. You can find this data in our OpenCV package. After installing OpenCv, you can see the folder name haarcascades. There would be .xml files for different application. Now, copy all of them for different use and paste then in a new folder under the current project.

Example

The following is the Python code using Haar Cascade to detect the face of Amitabh Bachan shown in the following image:

Import the OpenCV package as shown:

import cv2

import numpy as np

Now, use the HaarCascadeClassifier for detecting face:

face_detection = cv2.CascadeClassifier('D:/ProgramData/cascadeclassifier/haarcascade_frontalface_default.xml')

Now, for reading a particular image, use the imread() function:

img = cv2.imread('AB.jpg')

Now, convert it into grayscale because it would accept gray images:

gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

Now, using face_detection.detectMultiScale, perform actual face detection

faces = face_detection.detectMultiScale(gray, 1.3, 5)

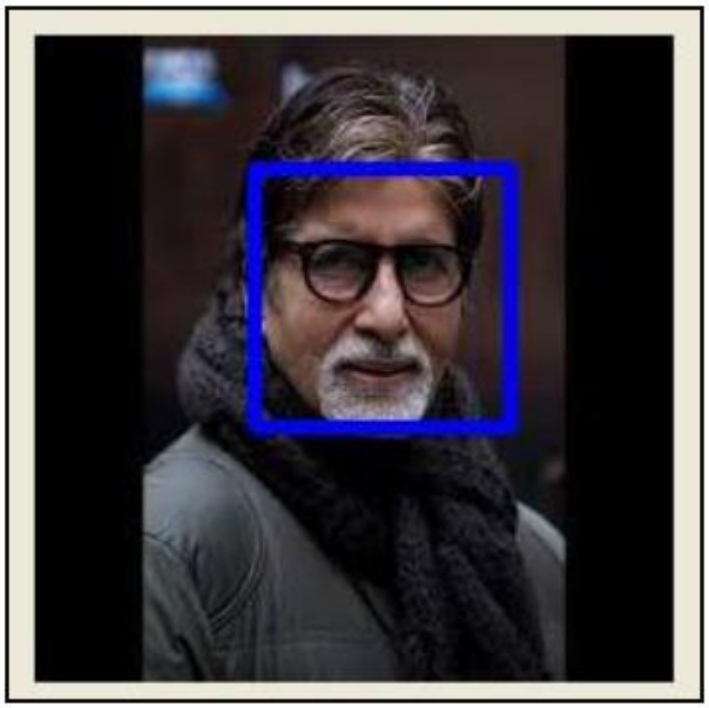
Now, draw a rectangle around the whole face:

for (x,y,w,h) in faces:

img = cv2.rectangle(img,(x,y),(x+w, y+h),(255,0,0),3)

cv2.imwrite('Face_AB.jpg',img)

This Python program will create an image named Face_AB.jpg with face detection as shown:



3.7 Eye Detection

Eye detection is another fascinating application of computer vision which makes it more realistic as well as futuristic. OpenCV has a built-in facility to perform eye detection. We are going to use the Haar cascade classifier for eye detection.

Example

The following example gives the Python code using Haar Cascade to detect the face of Amitabh Bachan given in the following image:

Import OpenCV package as shown:

import cv2

import numpy as np

Now, use the HaarCascadeClassifier for detecting face:

eye_cascade = cv2.CascadeClassifier('D:/ProgramData/cascadeclassifier/haarcascade_eye.xml')

Now, for reading a particular image, use the imread() function:

img = cv2.imread('AB_Eye.jpg')

Now, convert it into grayscale because it would accept grey images:

gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

Now with the help of eye_cascade.detectMultiScale, perform actual face detection:

eyes = eye_cascade.detectMultiScale(gray, 1.03, 5)

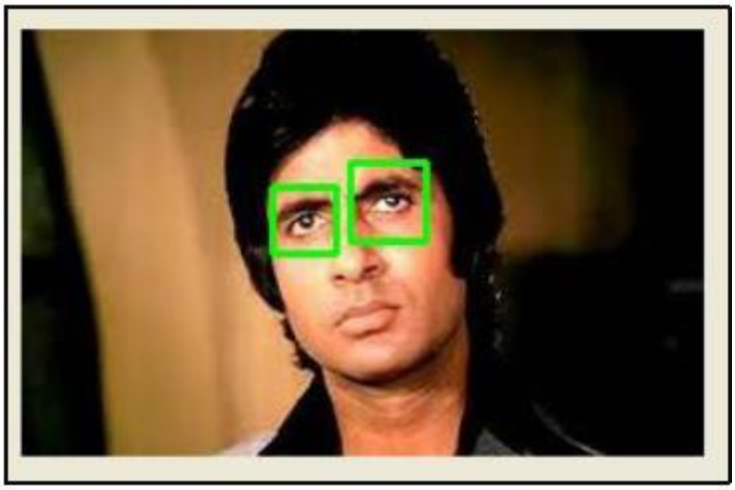
Now, draw a rectangle around the whole face:

for (ex,ey,ew,eh) in eyes:

img = cv2.rectangle(img,(ex,ey),(ex+ew, ey+eh),(0,255,0),2)

cv2.imwrite('Eye_AB.jpg',img)

This Python program will create an image named Eye_AB.jpg with eye detection as shown:



4.0 Conclusion

In this unit we have introduced the basics of computer vision with python. You will have to do more hands on in order to get acquainted with the language and do more advanced projects.

5.0 Summary

We hope you enjoyed this unit. This unit treats computer vision with python. Now, let us attempt the questions below.

6.0 Tutor Marked Assignment

i. Write a python code that is able to perform Color Space Conversion, Edge detection and face detection on the picture of you and your friend.

7.0 References/Further Readings

1) Joshi, P. (2017). Artificial intelligence with python. Packt Publishing Ltd.
2) Poole, D. L., & Mackworth, A. K. (2017). Python code for Artificial Intelligence: Foundations of Computational Agents.
3) Poole, D. L., & Mackworth, A. K. (2010). Artificial Intelligence: foundations of computational agents. Cambridge University Press.

MODULE 5: STUDY OF DIFFERENT CLASSES OF EXPERT SYSTEMS

Unit One: Rule Based: MYCIN
Unit Two: Blackboard; HEARSAY
Unit Three: Frame Based; KEE
Unit Four: Extensive independent study

Unit One: Rule Based: MYCIN,

CONTENT

1.0 Introduction

As the name suggests, rule based expert system consists of set of rules. Rule is an expressive, straight forward and flexible way of expressing knowledge. In a rule based expert system, knowledge is represented as a set of rules. Knowledge is a theoretical or practical understanding of a subject or a domain. (Negnevitsky, Artificial Intelligence: A guide to Intelligent systems, 2008) Expert possesses deep knowledge and practical experience over the years which results into expertise. Expert has an ability to code the knowledge in form of rules. Any rule consists of two parts: The IF part, called and antecedent

(premise or condition) and THEN part, called the consequent (conclusion or action). The basic syntax or rule base is:

IF {antecedent}
THEN {Consequent}

A rule can have multiple antecedents joined by the keywords AND, OR, or combination of both. The antecedent of a rule consists of two parts. They are object and its value. Object and value are linked by an operator. The operator can be mathematical or may be logical.

2.0 Objectives

At the end of this unit you should be able to …….

- Outline the main components of a rule based expert system
- Outline the major components of the MYCIN system
- Sketch the structure of MYCIN system components relationship
- Discuss the functions of each component of the MYCIN system
- Discuss how knowledge acquisition is implemented in the MYCIN system

3.0 Main Content

3.1 Structure of rule based expert system

A rule based expert system has five components: The knowledge base, the database, the inference engine, the explanation facility and the user interface. The knowledge base contains the knowledge about the domain. The database has set of facts, which is used to match the against the IF- THEN rules. The inference engine provides reasoning, so that expert system can reach a solution. The explanation facility provides the answer to user about why the particular solution is reached. The user interface enables user to interact with the other components of the expert systems.

The other additional components include the external interface, the developer interface, text editor, book keeping facilities, debugging aids, and run time knowledge acquisition. The external interface allows an expert system to interact with other database and programs. The developer interface allows developer to edit with knowledge base, rules and facts. Text editor provides notepad kind facility to enter inputs. Book keeping facility is provided to monitor the changes made by the knowledge engineer in knowledge base or inference engine. Debugging aids provides tracing of all rules fired during the program execution. Run time knowledge acquisition facility enables to add knowledge or facts, which are not available in knowledge base or database.

3.2 The Structure of the MYCIN System

A number of constraints influenced the design of the MYCIN system. In order to be useful, the system had to be easy to use and had to provide consistently reliable advice. It needed to be able to accommodate the large body of task-specific knowledge required for high performance, a knowledge base that is subject to change over time. The system also had to be able to use inexact or incomplete information. This applies not only to the absence of definitive laboratory data, but also to the medical domain itself (which is characterized by much judgmental knowledge). Finally, to be useful interactive

system, MYCIN needed to be capable of supplying explanations for its decisions and responding to physicians' questions, rather than simply printing orders.

The MYCIN system comprises three major subprograms, as depicted in Figure 3.1. The Consultation Program is the core of the system; it interacts with the physician to obtain information about the patient, generating diagnoses and therapy recommendations. The Explanation Program provides explanations and justifications for the program's actions. The Knowledge-Acquisition Program is used by experts to update the system's knowledge base.

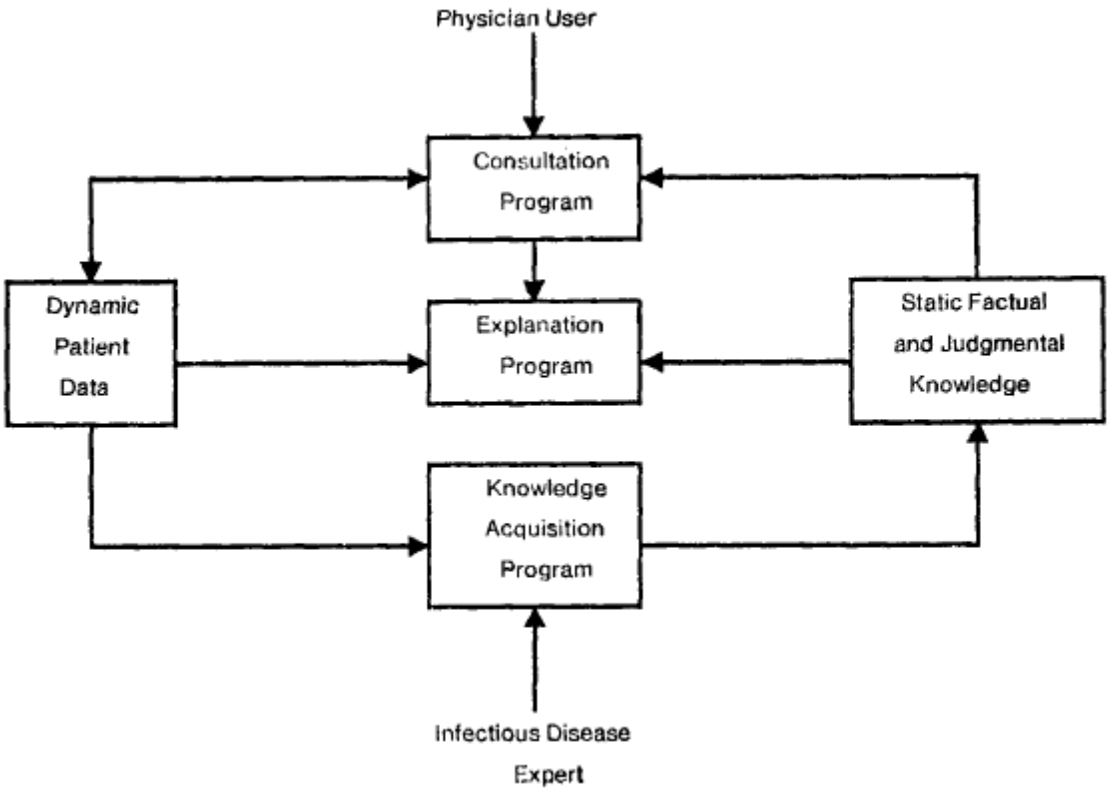


FIGURE 3.1 Organization of the MYCIN system. Arrows denote information flow between modules, knowledge base, and users.

3.2.1 The Consultation Program

Excerpts from a sample consultation are shown in Figure 3-2. The doctor's responses are in upper-case letters following the double-asterisk prompt. Considerable effort has gone into the human engineering aspects of the system, to make the interaction as smooth as possible. If the user finds a question unclear, he or she can request a rephrasing with a list of sample responses. A large dictionary of synonyms allows the user great latitude in choice of responses, and simple typing or spelling errors are automatically corrected.

3.2.2 Knowledge Organization

Information about the patient and conclusions drawn during the consultation are represented as associative (object-attribute-value) triples. The objects, known as contexts in MYCINa,r e such things as individual cultures taken from the patient, organisms that grew out of them, and drugs the patient is currently receiving. Various attributes, termed clinical parameters, characterize these objects. Questions asked during the consultation attempt to fill in the values for relevant attributes of these objects. To represent the uncertainty of data or competing hypotheses, attached to each triple is a certainty factor (CF), a number between - 1 and 1 indicating the strength of the belief in (or a measure of the importance of) that fact. A CF of represents total certainty of the truth of the fact, while a CF of - 1 represents certainty regarding the negation of the fact. While certainty factors are not conditional probabilities, they are informally based on probability theory (see Part Four). Some triples (with CF's) from a typical consultation might be as follows:

(IDENTITY ORGANISM-P1S EUDOMON0A.8S)
(IDENTITYO RGANISME-1. COLI0 .15)
(SITE CULTURE-T2H ROA1T.0)
(BURNEPDA TIENT-29Y8E S- 1.0)

1) Patient's name: (first-last)
** **FRED BRAUN**
2) Sex:
** **M**
3) Age:
** **55**
4) Are there any cultures for Fred Braun which may be related to the present illness, and from which organisms have been grown successfully in the microbiology laboratory?
** **Y**

(Questions 1-3 request background patient data.)

--------CULTURE-1--------
5) From what site was the specimen for CULTURE-1 taken?
** **BLOD**
= **BLOOD**
6) Please give the date and time when CULTURE-1 was obtained. (mo/da/yr time)
** **JUN 20, 1977**
The first organism isolated from the blood culture of 20-JUN-77 (CULTURE-1) will be referred to as:

(System corrects typing error.)

--------ORGANISM-1--------
7) Enter the laboratory-reported identity of ORGANISM-1:
**UNKNOWN
8) The stain (gram or Ziehl-Neelsen acid-fast) of ORGANISM-1:
** **NEG**
9) Is ORGANISM-1 rod or coccus (etc.):
** **ROD**
10) What is the form of the individual organisms (e.g. lancet-shaped for cocci, fusiform for rods, etc.)?
** **FUSIFORM**

(Typically identity is not yet known.)

(Preliminary lab results give some clues.)

(. . . more questions follow in order to gather sufficient information to infer the identity and significance of the infecting organisms . . .)

23) What is the suspected portal of entry of the gramneg rod from the blood culture of 20-JUN-77 (ORGANISM-1) into this sterile site?
** **GI (6)**
24) Is Fred Braun a burn patient?
** **N**
25) Is Fred Braun a compromised host (e.g. alcoholic, sickle-cell-disease, immunosuppressed)?
** **N**
26) Is Fred Braun febrile?
** **Y**

(User indicates the uncertainty of his response.)

(. . . after about 50–60 questions, MYCIN prints the diagnostic hypotheses on which therapy will be based . . .)

My therapy recommendations will be designed to treat for organisms that are either very likely or, although less likely, would have a significant effect on therapy selection if they were present. It is important to cover for the following probable infection(s) and associated organism(s):

INFECTION-1 is BACTEREMIA

<ITEM-1> E.COLI [ORGANISM-1]
<ITEM-2> KLEBSIELLA [ORGANISM-1]
<ITEM-3> ENTEROBACTER [ORGANISM-1]
<ITEM-4> KLEBSIELLA-PNEUMONIAE [ORGANISM-1]

(. . . questions follow to evaluate possible therapy choices, and finally MYCIN prints its therapy recommendations . . .)

[REC-1] My preferred therapy recommendation is as follows:
In order to cover for items <1 2 3 4>:
Give: GENTAMICIN *(One drug covers 4 possible identities.)*
Dose: 119 mg (6.0 ml) q8h IV for 10 days [calculated on basis of 1.7 mg/kg]
Comments: Modify dose in renal failure.

FIGURE 3.2 Excerpts from a MYCIN consultation. (Comments in italics are not part of the actual interaction.)

Here ORGANISM-I is probably Pseudomonas, but there is some evidence to believe it is E. coli; the site of CULTURE is-2 (without doubt) the throat; and PATIENT-298 is known not to be a burn patient.

3.2.3 Production Rules

MYCIN reasons about its domain using judgmental knowledge encoded as production rules. Each rule has a premise, which is a conjunction of predicates regarding triples in the knowledge base. If the premise is true, the conclusion in the action part of the rule is drawn. If the premise is known with less than certainty, the strength of the conclusion is modified accordingly.

A typical rule is shown in Figure 3.3. The predicates (such as SAME) are simple LISP functions operating on associative triples, which match the declared facts in the premise clause of the rule against the dynamic data known so far about the patient. SAND, the multi-valued analogue of the Boolean AND function, performs a minimization operation on CF's.

RULE035

PREMISE: ($AND (SAME CNTXT GRAM GRAMNEG)
(SAME CNTXT MORPH ROD)
(SAME CNTXT AIR ANAEROBIC))
ACTION: (CONCLUDE CNTXT IDENTITY BACTEROIDES TALLY .6)

IF: 1) The gram stain of the organism is gramneg, and
2) The morphology of the organism is rod, and
3) The aerobicity of the organism is anaerobic
THEN: There is suggestive evidence (.6) that the identity of the organism is bacteroides

FIGURE 3.3 A MYCIN rule, in both its internal (LISP) form and English translation. The term CNTXT appearing in every clause is a variable in MYCIN that is bound to the current context, in this case a specific organism (ORGANISM-t2o), which the rule may be applied.

The body of the rule is actually an executable piece of LISP code, and "evaluating" a rule entails little more than the LISP function EVAL. However, the highly stylized nature of the rules permits the system to examine and manipulate them, enabling many of the system's capabilities discussed below. One of these is the ability to produce an English translation of the LISP rule, as shown in the example. This is possible because each of the predicate functions has associated with it a translation pattern indicating the logical roles of the function's arguments.

It is intended that each rule be a single, modular chunk of medical knowledge. The number of rules in the MYCIN system grew to about 500.

3.2.4 Application of Rules---The Rule Interpreter

The control structure is a goal-directed backward chaining of rules. At any given time, MYCIN is working to establish the value of some clinical parameter.

To this end, the system retrieves the (precomputed) list of rules whose conclusions bear on this goal. The rule in Figure 3.3, for example, would be retrieved in the attempt to establish the identity of an organism. If, in the course of evaluating the premise of one of these rules, some other piece of information that is not yet known is needed, MYCIN sets up a subgoal to find out that information; this in turn causes other rules to be tried. Questions are asked during the consultation when rules fail to deduce the necessary information. If the user cannot supply the requested information, the rule is simply ignored. This control structure results in a highly focused search through the rule base.

3.3 Advantages of the Rule Methodology

The modularity of rules simplifies the task of updating the knowledge base. Individual rules can be added, deleted, or modified without drastically affecting the overall performance of the system. And because each rule is a coherent chunk of knowledge, it is a convenient unit for explanation purposes. For example, to explain why the system is asking a question during the consultation, a first approximation is simply to display the rule currently under consideration.

The stylized nature of the rules is useful for many operations. While the syntax of the rules permits the use of any LISP function, there is a small set of standard predicates that make up the vast majority of the rules. The system contains information about the use of these predicates in the form of function templates. For example, the predicate SAME is described as follows:

function template:
sample function call:
(SAME CNTXT PARM VALUE)
(SAME CNTXT SITE BLOOD)

The system can use these templates to "read" its own rules. For example, the template shown here contains the standard tokens CNTXT, PARM, and VALUE (for context, parameter, and corresponding value), indicating the components of the associative triple that SAME tests. If the clause above appears in the premise of a given rule, the system can determine that the rule needs to know the site of the

culture, and that the rule can only succeed if that site is, in fact, blood. When asked to display rules that are relevant to blood cultures, MYCIN will be able to choose that rule.

An important function of the templates is to permit MYCIN to precompute automatically (at system generation time) the set of rules that conclude about a particular parameter; it is this set that the rule monitor retrieves when the system needs to deduce the value of that parameter. The system can also read rules to eliminate obviously inappropriate ones. It is often the case that, of a large set of rules under consideration, several are provably false by information already known. That is, the information needed to evaluate one of the clauses in the premise has already been determined, and that clause is false, thereby making the entire premise false. By reading the rules before actually invoking them, many can be immediately discarded, thereby avoiding the deductive work necessary in evaluating the premise clauses that precede the false one (this is called the preview mechanism). In some cases, this means the system avoids the useless search of one or more subgoal trees, when the information thereby deduced would simply be overridden by the demonstrably false premise.

Another more dramatic case occurs when it is possible, on the basis of information currently available, to deduce with certainty the value of some parameter that is needed by a rule. This is the case when there exists a chain of one or more rules whose premises are known (or provable, as above) with certainty and that ultimately conclude the desired value with certainty. Since each rule in this chain must have a certainty factor of 1.0, we term such a chain a unity path; and since a value known with certainty excludes all other potential values, no other rules need be tried. MYCIN always seeks a unity path before trying a set of rules or asking a question; typically, this means "commonsense" deductions are made directly, without asking the user "silly" questions or blindly invoking all the rules pertaining to the goal. Since there are usually few rules on any potential unity path, the search tends to be small.

The ability to read rules opens the way to the writing of rules that manipulate other rules. We term such rules meta-rules (see Part Nine); they are used to make deductions not about the medical entities of the domain but about strategies to be used by the system. Whenever the rule interpreter is about to invoke a list of rules to establish some goal, it first applies any meta-rules associated with that goal. These meta-rules can reorder or prune the rule list to make the search more suitable for the given case.

3.4 Explanation Capability

A major subprogram of MYCIN is a general Question-Answering (QA) Module, which answers simple English-language questions concerning the system’s decisions in a particular consultation or about the system’s general knowledge. A limited set of commonly desired explanations is also provided in a command style during the consultation by the Reasoning Status Checker. This special module explains the system’s current line of reasoning and motivation for the questions asked. The QA Module is entered automatically at the end of each consultation and is also available to the user during the consultation if the need arises. The user is encouraged to ask questions about any aspect of MYCIN’s performance.

Because of the relatively unambiguous vocabulary used in the medical domain, and because of our desire to minimize the delay in answering questions, we have forgone any complicated parsing techniques in the QA Module. Rather, we use a simple pattern-matching approach, relying on the strong implications of certain keywords in the user’s questions. The words in the translation of a clinical parameter, and values of that parameter, are tagged in the system dictionary as implying that

parameter. By suitably weighing the implications of each word in the sentence, the system determines one or more parameter/value pairs being asked about. Pattern matching is used to determine the general type of the question, which indicates what is being asked about those parameter/value pairs. Typical questions (see Figure 3.4) seek the value of a certain parameter, how this value was concluded, and how this parameter was used.

**** IS ORGANISM-1 A STREPTOCOCCUS?**
No. The identity of ORGANISM-1 is e.coli (.74) bacteroides (.7)
klebsiella-pneumoniae (.4) pseudomonas-aeruginosa (.22) enterobacter (.21) proteus-non-mirabilis (.19) serratia (.03) salmonella (.02)

****WHY DID YOU RULE OUT STREPTOCOCCUS AS A POSSIBILITY FOR ORGANISM-1?**
RULE033 could have been used to determine that the identity of ORGANISM-1 is streptococcus. However, clause 2 of RULE033 ["the morphology of the organism is coccus"] was already known to be false for ORGANISM-1, so the rule was never tried.

**** HOW DO YOU KNOW THAT CULTURE-1 WAS FROM A STERILE SOURCE?**
I used RULE189 to conclude that this blood culture was taken from a sterile source. This gave a cumulative CF of (1.0). The last question asked before the conclusion was made was 7.

**** WHAT DOES GU MANIPULATION TELL YOU ABOUT THE IDENTITY OF AN ORGANISM?**
The rules listed below use:
whether the patient has had a genito-urinary manipulative procedure
to conclude about:
the identity of the organism

156, 163, 190
Which do you wish to see?
**** 156**

RULE156
IF: 1) The site of the culture is blood, and
2) The gram stain of the organism is gramneg, and
3) The morphology of the organism is rod, and
4) The portal of entry of the organism is urine, and
5) The patient has not had a genito-urinary manipulative procedure, and
6) Cystitis is not a problem for which the patient has been treated
THEN: *There is suggestive evidence (.6) that the identity of the* organism is e.coli

**** IS BLOOD A STERILE SITE?**
Yes. Blood is one of those sites that is normally sterile.

**** WHAT ARE THE NONSTERILE SITES?**
Those sites that are normally nonsterile are: CERVIX, CUTANEOUS-ULCER, LOCHIA, NOSE, SKIN, STOOL, THROAT, URETHRA, VAGINA.

**** IS BACTEROIDES AEROBIC?**
No, the aerobicity of bacteroides is anaerobic.

FIGURE 3.4 Interaction with the QA Module. The user's questions follow the double-asterisk prompt.

Questions about the value of a given parameter of an object are answered by simply displaying the current hypothesis regarding it. To explain how the value of a parameter was concluded, MYCIN

retrieves the list of rules that were successfully applied and prints them, along with the conclusions drawn. Alternatively, if the user supplied the value by answering a previous question, this is noted. More general questions about how information is used or concluded are answered by retrieving the relevant rules from the rule base.

34) Was penicillinase added to this blood culture (CULTURE-1)?
**** WHY**

[i.e. WHY is it important to determine whether penicillinase was added to CULTURE-1?]

[3.0] This will aid in determining whether ORGANISM-1 is a contaminant.
It has already been established that
[3.1] the site of CULTURE-1 is blood, and
[3.2] the gram stain of ORGANISM-1 is grampos
Therefore, if
[3.3] penicillinase was added to this blood culture
then
there is weakly suggestive evidence (.4) that ORGANISM-1 is a contaminant [RULE039]

**** WHY**

[i.e. WHY is it important to determine whether ORGANISM-1 is a contaminant?]

[4.0] . . . In order to determine whether there is significant disease associated with this occurrence of ORGANISM-1.
It has already been established that
[4.1] this blood culture was taken from a sterile source
Therefore, if
[4.2] ORGANISM-1 is not a contaminant
then
there is strongly suggestive evidence (.9) that there is significant disease associated with this occurrence of ORGANISM-1 [RULE032]

**** WHY**

[i.e. WHY is it important to determine whether there is significant disease associated with this occurrence of ORGANISM-1]

[5.0] . . . In order to determine the organisms which require therapy.
It has already been established that
[5.1] the infection is known, and
[5.2] the identity of ORGANISM-1 is known
Therefore, if
[5.3] there is significant disease associated with this occurrence of ORGANISM-1
then
it is definite (1.0) that ORGANISM-1 is one of the organisms which require therapy
[RULE090]

**** HOW 5.2**

[i.e. HOW was it established that the identity of ORGANISM-1 is known?]

Question 9 was asked, but you weren't sure; so the following was used:

[5.5] RULE084 was used to conclude that the identity of ORGANISM-1 is streptococcus-group-a (.29).

FIGURE 3.5 Use of the Reasoning Status Checker during the consultation to explain MYCIN's line of reasoning.

As shown in Figure 3.5, the Reasoning Status Checker is invoked by the HOW and WHY commands. At any time during the consultation, when the user is asked a question, he or she can delay answering it and instead ask why the question was asked. Since questions are asked in order to establish the truth of the premise of some rule, a simple answer to WHY is "because I'm trying to apply the following rule." Successive WHY questions unwind the chain of subgoals, citing the rules that led to the current rule being tried.

Besides examining the current line of reasoning, the user can also ask about previous decisions, or about how future decisions might be made, by giving the HOW command. Explaining how the truth of a certain clause was established is accomplished as described above for the general QA Module. To explain how a presently unknown clause might be established, MYCIN retrieves the set of rules that the rule interpreter would select to establish that clause and selects the relevant rules from among them by "reading" the premises for applicability and the conclusions for relevance to the goal.

3.5 Knowledge Acquisition

The knowledge base is expanded and improved by acquiring new rules, or modifications to old rules, from experts. Ordinarily, this process involves having the medical expert supply a piece of medical knowledge in English, which a system programmer converts into the intended LISP rule. This mode of operation is suitable when the expert and the skilled programmer can work together. Ideally, however, the expert should be able to convey his or her knowledge directly to the system.

Work has been undertaken to allow experts to update the rule base directly. A rule-acquisition routine parses an English-language rule by methods similar to those used in parsing questions in the QA Module. Each clause is broken down into one or more object-attribute value triples, which are fitted into the slots of the appropriate predicate function template. This process is further guided by rule models, which supply expectations about the structure of rules and the interrelationships of the clinical parameters.

One mode of acquisition that has received special attention is acquiring new rules in the context of an error. In this case, the user is trying to correct a localized deficiency in the rule base; if a new rule is to correct the program's faulty behavior, it must at the very least apply to the consultation at hand. In particular, each of the premises must evaluate to TRUE for the given case. These expectations greatly simplify the task of the acquisition program, and also aid the expert in formulating new rules.

One difficult aspect of rule acquisition is the actual formulation of medical knowledge into decision rules. Our desire to keep the rule format simple is occasionally at odds with the need to encode the many aspects of medical decision making. The backward chaining of rules by the deductive system is also often a stumbling block for experts who are new to the system. However, they soon learn to structure their knowledge appropriately. In fact, some experts have felt that encoding their knowledge

into rules has helped them formalize their own view of the domain, leading to greater consistency in their decisions.

4.0 Conclusion

This unit introduces the general structure of rule based expert systems. A case study of the MYCIN system was used to explain how rule-based systems are implemented. Advantages of the rule methodology were also discussed.

5.0 Summary

We hope you enjoyed this unit. This unit discussed rule based expert systems. Now, let us attempt the questions below.

6.0 Tutor Marked Assignment

i. Give a detailed discussion on the production rules and the function of the rule interpreter of the MYCIN system.

7.0 References/Further Readings

1) Nagori, V., & Trivedi, B. (2014). Types of Expert System: Comparative Study. Asian Journal of Computer and Information Systems (ISSN: 2321–5658), 2(02).
2) Van Melle, W. (1984). The structure of the MYCIN system. Rule-Based Expert Systems: The MYCIN Experiments of the Stanford Heuristic Programming Project, cap, 4, 67-77.
3) Davis, R., & King, J. J. (1984). The origin of rule-based systems in AI. Rule-based expert systems: The MYCIN experiments of the Stanford Heuristic Programming Project.
4) Liebowitz, J. (Ed.). (1997). The handbook of applied expert systems. Crc Press.

Unit Two: Blackboard; HEARSAY,

CONTENT

1.0 Introduction

Blackboard Model of Problem Solving Historically, the blackboard model arose from abstracting features of the HEARSAY-II speech understanding system developed between 1971 and 1976. HEARSAY-II understood a spoken speech query about computer science abstracts stored in a database. It "understood" in the sense that it was able to respond to spoken commands and queries about the data-base. From an informal summary description of the HEARSAY-II program, the HASP system was designed

and implemented between 1973 and 1975. The domain of HASP was ocean surveillance, and its task was the interpretation of continuous passive sonar data. HASP, as the second example of a blackboard system, not only added credibility to the claim that a blackboard approach to problem solving was general, but it also demonstrated that it could be abstracted into a robust model of problem solving. Subsequently, many application programs have been implemented whose solutions were formulated using the blackboard model. Because of the different characteristics of the application problems and because the interpretation of the blackboard model varied, the design of these programs differed considerably. However, the blackboard model of problem solving has not undergone any substantial changes in the last ten years.

2.0 Objectives

At the end of this unit, you should be able to:

- Outline the three major components consisting the blackboard model
- Define the functions of the three major components consisting the blackboard model
- Outline the iterative sequence used for problem solving in the blackboard model
- Discuss the HEARSAY-II system as an expert system
- Explain the knowledge-application strategy of HEARSAY-II

3.0 Main content

A problem-solving model is a scheme for organizing reasoning steps and domain knowledge to construct a solution to a problem. For example, in a backward-reasoning model, problem solving begins by reasoning backwards from a goal to be achieved towards an initial state (data). More specifically, in a rule-based backward-reasoning model knowledge is organized as “if-then” rules and modus ponens inference steps are applied to the rules from a goal rule back to an “initial-state rule” (a rule that looks at the input data). An excellent example of this approach to problem solving is the MYCIN program. In a forward-reasoning model, however, the inference steps are applied from an initial state toward a goal. The OPS system exemplifies such a system. In an opportunistic-reasoning model, pieces of knowledge are applied either backward or forward at the most “opportune” time. Put another way, the central issue of problem-solving deals with the question of: “What pieces of knowledge should be applied when and how?” A problem-solving model provides a conceptual framework for organizing knowledge and a strategy for applying that knowledge.

The blackboard model of problem solving is a highly structured, special case of opportunistic problem solving. In addition to opportunistic reasoning as a knowledge-application strategy, the blackboard model prescribes the organization of the domain knowledge and all the input and intermediate and partial solutions needed to solve the problem. ‘We refer to all possible partial and full solutions to a problem as its solution space.

In the blackboard model the solution space is organized into one or more application- dependent hierarchies. Information at each level in the hierarchy represents partial solutions and is associated with a unique vocabulary that describes the information. The domain knowledge is partitioned into independent modules of knowledge that transform information on one level, possibly using information at other levels, of the hierarchy into information on the same or other levels. The knowledge modules perform the transformation using algorithmic procedures or heuristic rules that generate actual or hypothetical transformations.

Opportunistic reasoning is applied within this overall organization of the solution space and task-specific knowledge: that is, which module of knowledge to apply is determined dynamically, one step at a time, resulting in the incremental generation of partial solutions.

The choice of a knowledge module is based on the solution state (particularly, the latest additions and modifications to the data structure containing pieces of the solution) and on the existence of knowledge

modules capable of improving the current state of the solution. At each step of knowledge application, either forward- or backward-reasoning methods may be app1ied.

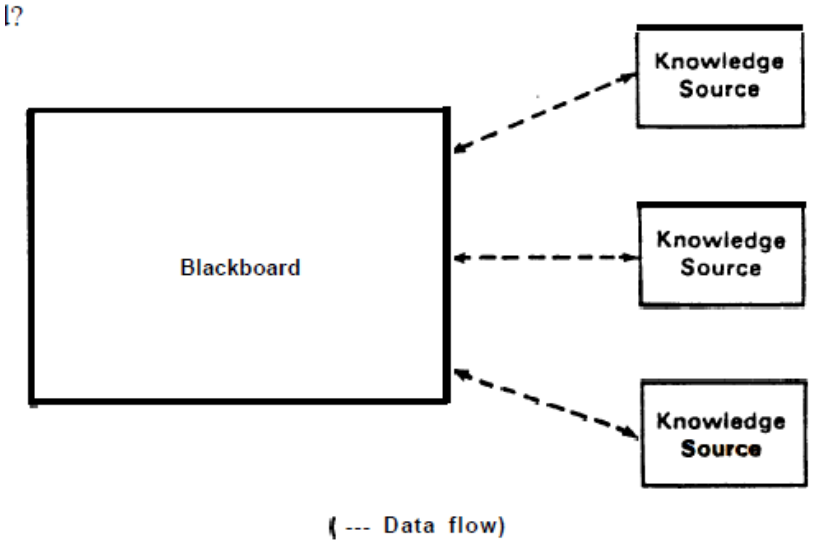
The blackboard model is a relatively complex problem-solving model prescribing the organization of knowledge and data and the problem-solving behavior within the overall organization. This section contains a description of the basic blackboard model. Variations and extensions will be discussed in subsequent sections.

3.1 The Blackboard Model

The blackboard model is usually described as consisting of three major components:

1. **The knowledge sources:** The knowledge needed to solve the problem is partitioned into knowledge sources, which are kept separate and independent.
2. **The blackboard data structure:** The problem-solving state data are kept in a global data base, the blackboard. Knowledge sources produce changes to the blackboard which lead incrementally to a solution to the problem. Communication and interaction among the knowledge sources take place solely through the blackboard.
3. **Control:** The knowledge sources respond opportunistically to changes in the blackboard?

The difficulty with this description of the blackboard model is that it only outlines the organizational principles. For those who want to build a blackboard system, the model does not specify how it is to be realized as a computational entity, that is, the blackboard model is a conceptual entity, not a computational specification. Given a problem to be solved, the blackboard model provides enough guidelines for sketching a solution, but a sketch is a long way from a working system. To design and build a system, a detailed model is needed. Before moving on to adding details to the blackboard model, we explore the implied behavior of this abstract model.



There is a global database called the blackboard, and there are logically independent sources of knowledge called the knowledge sources. The knowledge sources respond to changes on the blackboard. Note that there is no control flow; the knowledge sources are self-activating.

Figure 3.1: The Blackboard Model

Let us consider a hypothetical problem of a group of people trying to put together a jigsaw puzzle. Imagine a room with a large blackboard and around it a group of people each holding over-size jigsaw pieces. We start with volunteers who put on the blackboard (assume it's sticky) their most "promising" pieces. Each member of the group looks at his pieces and sees if any of them fit into the pieces already on the blackboard. Those with the appropriate pieces go up to the blackboard and update the evolving solution. The new updates cause other pieces to fall into place, and other people go to the blackboard to add their pieces. It does not matter whether one person holds more pieces than another. The whole puzzle can be solved in complete silence: that is, there need be no direct communication among the group. Each person is self-activating, knowing when his pieces will contribute to the solution. No a priori established order exists for people to go up to the blackboard. The apparent cooperative behavior is mediated by the state of the solution on the blackboard. If one watches the task being performed, the solution is built incrementally (one piece at a time) and opportunistically (as an opportunity for adding a piece arises), as opposed to starting, say, systematically from the left top corner and trying each piece.

This analogy illustrates quite well the blackboard problem-solving behavior implied in the model and is fine for a starter. Now, let's change the layout of the room in such a way that there is only one center aisle wide enough for one person to get through to the blackboard.

Now, no more than one person can go up to the blackboard at one time, and a monitor is needed, someone who can see the group and can choose the order in which a person is to go up to the blackboard. The monitor can ask all people who have pieces to add to raise their hands. The monitor can then choose one person from those with their hands raised. To select one person, criteria for making the choice is needed, for example, a person who raises a hand first, a person with a piece that bridges two solution islands (that is, two clusters of completed pieces) and so forth. The monitor needs a strategy or a set of strategies for solving the puzzle.

The monitor can choose a strategy before the puzzle solving begins or can develop strategies as the solution begins to unfold. In any case; it should be noted that the monitor has a broad executive power. The monitor has so much power that the monitor could, for 'example, force the puzzle to be solved systematically from left to right; that is, the monitor has the power to violate one essential characteristic of the original blackboard model, that of opportunistic problem solving.

The last analogy, though slightly removed from the original model, is a useful one for computer programmers interested in building blackboard systems. Given the serial nature of most current computers, the conceptual distance between the model and a running blackboard system is a bit far, and the mapping from the model to a system is prone to misinterpretation.

By adding the constraint that solution building physically occur one step at a time in some order determined by the monitor (when multiple steps are possible and desirable), the blackboard model is brought closer to the realities inherent in serial computing environments.

3.1.1 The Blackboard Framework

Applications are implemented with different combinations of knowledge representations, reasoning schemes, and control mechanisms. The variability in the design of blackboard systems is due to many factors, the most influential one being the nature of the application problem itself. It can be seen, however, that blackboard architectures which underly application programs have many similar features and constructs. The blackboard framework is created by abstracting these constructs. The blackboard framework, therefore, contains descriptions of the blackboard system components that are grounded in actual computational constructs. The purpose of the framework is to provide design guidelines appropriate for blackboard systems in a serial-computing environment. Figure 3.2 shows some modifications to Figure 3.1 to reflect the addition of system-oriented details.

1. **The knowledge sources:** The domain knowledge needed to solve a problem is partitioned into knowledge sources that are kept separate and independent.

The objective of each knowledge source is to contribute information that will lead to a solution to the problem. A knowledge source takes a set of current information on the blackboard and updates it as encoded in its specialized knowledge.

The knowledge sources are represented as procedures, sets of rules, or logic assertions. To date most of the knowledge sources have been represented as either procedures or as sets of rules. However, systems that deal with signal processing either make liberal use of procedures in their rules, or use both rule sets and procedurally encoded knowledge sources.

The knowledge sources modify only the blackboard or control data structures (that also might be on the blackboard), and only the knowledge sources modify the blackboard. All modifications to the solution state are explicit and visible.

Each knowledge source is responsible for knowing the conditions under which it can contribute to a solution. Each knowledge source has preconditions that indicate the condition on the blackboard which must exist before the body of the knowledge source is activated?'

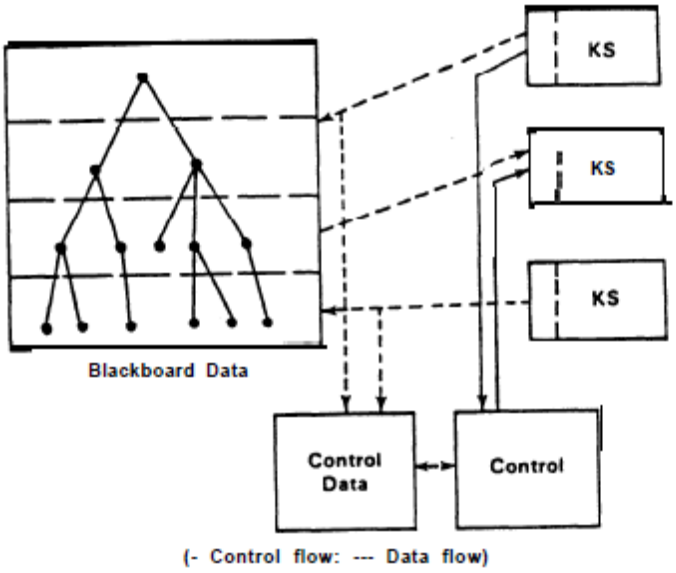
2. **The blackboard data structure:** The problem-solving state data are kept in a global database, the blackboard. Knowledge sources produce changes to the blackboard that lead incrementally to a solution, or a set of acceptable solutions, to the problem. Interaction among the knowledge sources takes place solely through changes on the blackboard.

The purpose of the blackboard is to hold computational and solution-state data needed by and produced by the knowledge sources. The knowledge sources use the blackboard data to interact with each other indirectly.

The blackboard consists of objects from the solution space. These objects can be input data, partial solutions, alternatives, and final solutions (and, possibly, control data).

The objects on the blackboard are hierarchically organized into levels of analysis. Information associated with objects (that is, their properties) on one level serves as input to a set of knowledge sources, which, in turn, place new information on the same or other levels.

The objects and their properties define the vocabulary of the solution space. The properties are represented as attribute-value pairs. Each level uses a distinct subset of the vocabulary.



The data on the blackboard are hierarchically organized. The knowledge sources are logically independent, self-selecting modules. Only the knowledge sources are allowed to make changes to the blackboard. Based on the latest changes to the information on the blackboard, a control module selects and executes the next knowledge source.

Figure 3.2: The Blackboard Framework

The relationships between the objects are denoted by named links. The relationship can be between objects on different levels. such as “part-of” or “in support of,“ or between objects on the same level, such as “next-to” or “follows.”

The blackboard can have multiple blackboard panels; that is, a solution space can be partitioned into multiple hierarchies.

3. **Control:** The knowledge sources respond opportunistically to changes on the blackboard. There is a set of control modules that monitor the changes on the blackboard and decide what actions to take next.

Various kinds of information are made globally available to the control modules. The information can be on the blackboard or kept separately. The control information is used by the control modules to determine the focus of attention.

The focus of attention indicates the next thing to be processed. The focus of attention can be either the knowledge sources (that is, which knowledge sources to activate next) or the blackboard objects (i.e., which solution islands to pursue next), or a combination of both (i.e., which knowledge sources to apply to which objects).

The solution is built one step at a time. Any type of reasoning step (data driven, goal driven, model driven, and so on) can be applied at each stage of solution formation. As a result, the sequence of knowledge source invocation is dynamic and opportunistic rather than fixed and preprogrammed.

Pieces of problem-solving activities occur in the following iterative sequence:

1. A knowledge source makes change(s) to blackboard object(s). As these changes are made, a record is kept in a global data structure that holds the control information.

2. Each knowledge source indicates the contribution it can make to the new solution state. (This can be defined a priori for an application, or dynamically determined.)

3. Using the information from points 1 and 2 a control module selects a focus of attention.

4. Depending on the information contained in the focus of attention, an appropriate control module prepares it for execution as follows:

a. If the focus of attention is a knowledge source, then a blackboard object (or sometimes, a set of blackboard objects) is chosen to serve as the context of its invocation (knowledge-scheduling approach).

b. If the focus of attention is a blackboard object, then a knowledge source is chosen which will process that object (event-scheduling approach).

c. If the focus of attention is a knowledge source and an object, then that knowledge source is ready for execution. The knowledge source is executed together with the context, thus described.

Criteria are provided to determine when to terminate the process. Usually, one of the knowledge sources indicates when the problem-solving process is terminated, either because an acceptable solution has been found or because the system cannot continue further for lack of knowledge or data.

3.1.2 Problem-Solving Behavior and Knowledge Application

The problem-solving behavior of a system is determined by the knowledge-application strategy encoded in the control modules. The choice of the most appropriate knowledge-application strategy is dependent on the characteristics of the application task and on the quality and quantity of domain knowledge relevant to the task. Basically, the acts of choosing a particular blackboard region and choosing a particular knowledge source to operate on that region determine the problem-solving behavior. Generally, a knowledge source uses information on one level as its input and produces output information on another level. Thus, if the input level of a particular knowledge source is on the level' lower (closer to data) than its output level, then the application of this knowledge source is an application of bottom-up, forward reasoning.

Conversely, a commitment to a particular type of reasoning step is a commitment to a particular knowledge-application method. For example, if we are interested in applying a data directed, forward-reasoning step, then we would select a knowledge source whose input level is lower than its output level. If we are interested in goal-directed reasoning, we would select a knowledge source that put information needed to satisfy a goal on a lower level. Using the constructs in the control component one can make any type of reasoning step happen at each step of knowledge application.

How a piece of knowledge is stated often presupposes how it is to be used. Given a piece of knowledge about a relationship between information on two levels, that knowledge can be expressed in top-down or bottom-up application forms. These can further be refined. The top-down form can be written as a goal, an expectation, or as an abstract model of the lower level information. For example, a piece of knowledge can be expressed as a conjunction of information on a lower level needed to generate a

hypothesis at a higher level (a goal), Or, it can be expressed as information on a lower level needed to confirm a hypothesis at a higher level (an expectation), and so on. The framework does not presuppose nor does it prescribe the knowledge-application, or reasoning, methods. It merely provides constructs within which any reasoning methods can be used. Many interesting problem-solving behaviors have been implemented using these constructs.

3.1.3 Perspectives

The organizational underpinnings of blackboard systems have been the primary focus. The blackboard framework is a system-oriented interpretation of the blackboard model. It is a mechanistic formulation intended to serve as a foundation for system specifications. In problem-solving programs, we are usually interested in their performance and problem-solving behavior, not their organization. We have found, however, that some classes of complex problems become manageable when they are formulated along the lines of the blackboard model. Also, interesting problem-solving behavior can be programmed using the blackboard framework as a foundation. Even though the blackboard framework still falls short of being a computational specification, given an application task and the necessary knowledge, it provides enough information so that a suitable blackboard system can be designed, specified, and built.

Some examples of complex problems with interesting problem-solving behavior are discussed in Section 3. The examples show that new constructs can be added to the blackboard framework as the application problems demand, without violating the guidelines contained in it.

There are other perspectives on the blackboard model. The blackboard model is sometimes viewed as a model of general problem solving [16]. It has been used to structure cognitive models [28]. [41], [20]; the OPM system (described in Section 3.5.1) simulates ‘the human planning process. Sometimes the blackboard model is used as an organizing principle for large, complex systems built by many programmers. The ALVan project [49] takes this approach.

3.2 HEARSAY-II

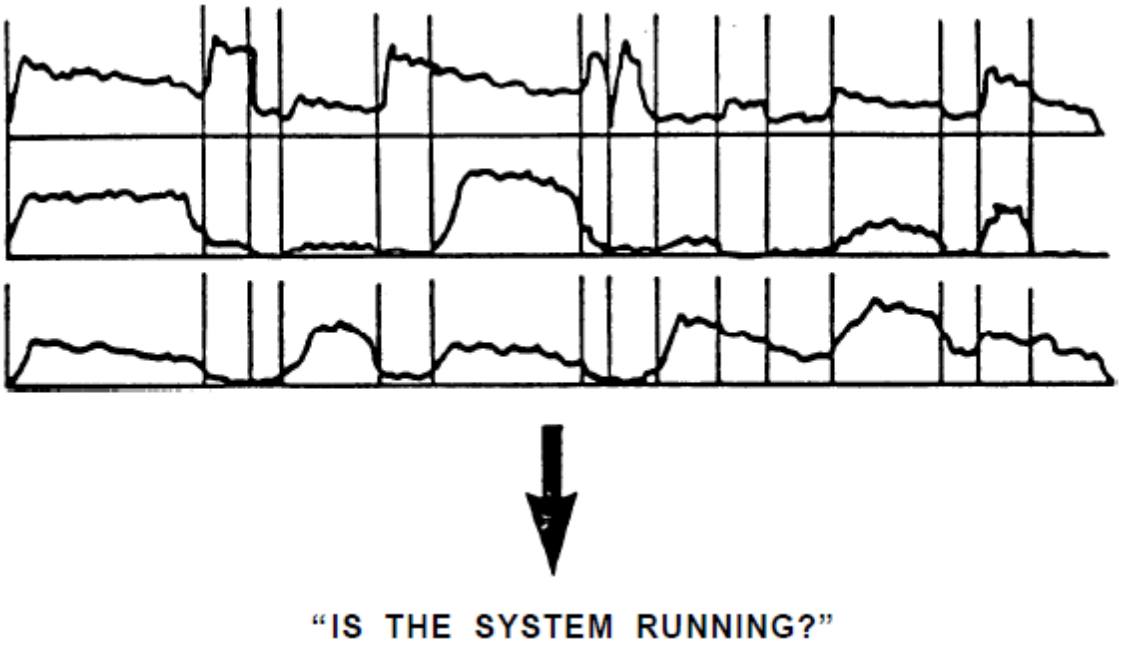


Figure 3.3: The HEARSAY-II Task

One additional item of historical context is worth noting, however. Various continuous speech-understanding projects were brought under one umbrella in the Defense Advanced Research Projects Agency (DARPA) Speech Understanding Project, a five-year project that began in 1971. The goals of the Speech Understanding Project were to design and implement systems that "accept continuous speech from many cooperative speakers of the general American dialect in a quiet room over a good quality microphone, allowing a slight tuning of the system per speaker, by requiring only natural adaptation by the user, permitting a slightly selected vocabulary of 1,000 words, with a highly artificial syntax...in a few times real time..." Hearsay-II was developed at Carnegie-Mellon University for the Speech Understanding Project and successfully met most of these goals.

3.2.1 The Task

The goal of the HEARSAY-II system was to understand speech utterances. To prove that it understood a sentence, it performed the spoken commands. In the earlier HEARSAY-1 period, the domain of discourse was chess (for example, bishop moves to king knight five). In the HEARSAY-II era, the task was to answer queries about, and to retrieve documents from, a collection of computer science abstracts in the area of artificial intelligence. For example, the system understood the following types of command:

"Which abstract refer to the theory of computation?"

"List those articles."

"What has McCarthy written since nineteen seventy-four?"

The HEARSAY-II system was not restricted to any particular task domain. “Given the syntax and the vocabulary of a language and the semantics of the task, it attempts recognition of the utterance in that language.” The vocabulary for the document retrieval task consisted of 1011 words in which each extended form of a root, for example, the plural of a noun, was counted separately. The grammar defining a legal sentence was context-free and included recursion, and imbedded semantics and pragmatic constraints. For example, in the place of noun in conventional grammars, this grammar included such non-terminals as topic, author, year, and publisher. The grammar allowed each word to be followed, on the average, by seventeen other words in the vocabulary.

The problem of speech understanding is characterized by error and variability in both the input and the knowledge. “The first source of error is due to deviation between ideal and spoken messages due to inexact production [input], and the second source of error is due to imprecise rules of comprehension [knowledge].” Because of these uncertainties, a direct mapping between the speech signals and a sequence of words making up the uttered sentence is not possible. The HEARSAY designers structured the understanding problem as a search in a space consisting of complete and partial interpretations. These interpretations were organized within an abstraction hierarchy containing signal parameters, segments, phones, phonemes, syllables, words, phrases, and sentence levels. This approach required the use of a diverse set of knowledge that produced large numbers of partial solutions on the many levels.

Furthermore, the uncertainties in the knowledge generated many competing, alternative hypothetical interpretations. To avoid a combinatorial explosion, the knowledge sources had to construct partial interpretations by applying constraints at each level of abstraction. For example, one kind of constraint is imposed when an adjacent word is predicted, and the prediction is used to limit subsequent search. The constraints also had to be added in such a way that their accrual reduced the uncertainty inherent in the data and the knowledge sources.

In order to control the combinatorial explosion and to meet the requirement for near real-time understanding, the interpretation process had to be selective in exploiting the most promising hypotheses, both in terms of combining them (for example, combining syllables into words) and in terms of predicting neighboring hypotheses around them (for example, a possible adjective to precede a noun). Thus, the need for incremental problem solving and flexible, opportunistic control were inherent in HEARSAY’s task.

3.2.2 The Blackboard Structure

The blackboard was partitioned into six to eight (depending on the configuration) levels of analysis corresponding to the intermediate levels of the decoding process. These levels formed a hierarchy in which the solution-space elements on each level could be described loosely as forming an abstraction of information on its adjacent lower level. One such hierarchy was comprised of, from the lowest to the highest level: parametric, segmental, phonetic, phonemic, syllabic, lexical, phrasal, and conceptual levels (see Figure 3.4). A blackboard element represented a hypothesis. An element at the lexical level, for example, represented a hypothesized word whose validity was supported by a group of syllables on the syllable level. The blackboard could be viewed as a three-dimensional problem space with time

(utterance sequence) on the x-axis, information levels containing a hypothesized solution on the y-axis, and alternative solutions on the z-axis.

Each hypothesis, no matter which level it belonged to, was constructed using a uniform structure of attribute-value pairs. Some attributes, such as its level name, were required for all levels. The attributes included a validity rating and an estimate of the "truth" of the hypothesis represented as some integer value. The relationships between the hypotheses on different levels were represented by links, forming an AND/OR tree over the entire hierarchy, Alternative solutions were formed by expanding along the OR paths. Because of the uncertainty of the knowledge sources that generated the hypotheses, the blackboard had a potential for containing a large number of alternative hypotheses.

3.2.3 The Knowledge Source Structure

Each knowledge source had two major components: a condition part (often referred to as a precondition) and an action part. Both the condition and the action parts were written as arbitrary SAIL procedures. "The condition component prescribed the situations in which the knowledge sources may contribute to the problem-solving activity, and the action component specified what that contribution was and how to integrate it into the current situation."

When executed, the condition part searched the blackboard for hypotheses that were of interest to its corresponding action part; all the relevant hypotheses found during the search were passed on to the action part. Upon activation, the action part processed all the hypotheses passed to it. The tasks of the knowledge sources ranged from classification (classifying acoustic segments into phonetic classes), to recognition (recognizing words) to generation and evaluation of predictions.

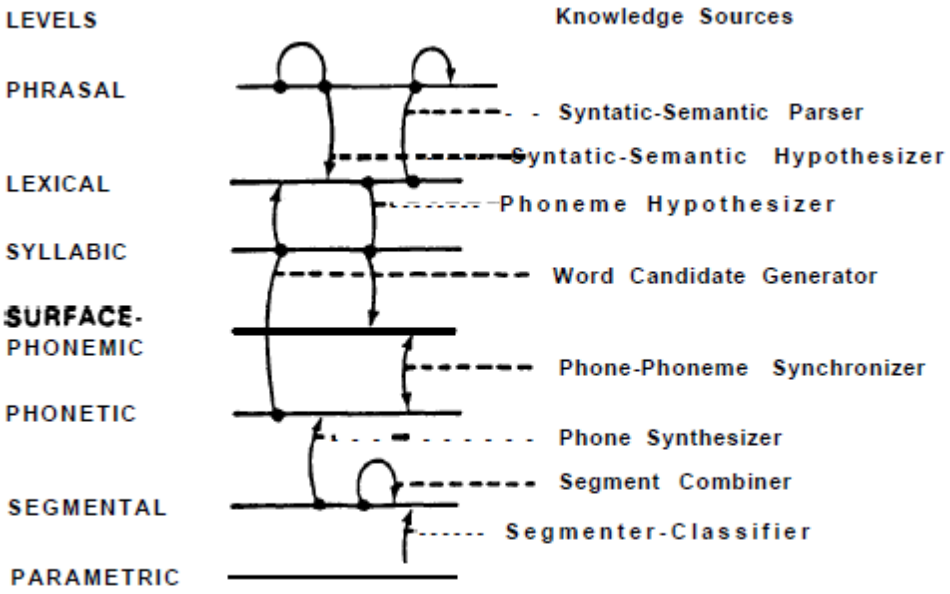


Figure 3.4: HEARSAY-II Blackboard and Knowledge Sources evaluation of predictions.

3.2.4 Control

The control component consisted of a blackboard monitor and a scheduler (see Figure 3.4). The monitor kept an account of each change made to the blackboard, its primitive change type, and any new hypotheses. Based on the change types and declarative information provided by the condition part of the knowledge sources, the monitor placed pointers to those condition parts which potentially could be executed on a scheduling queue. In addition to the condition parts ready for execution, the scheduling queue held a list of pointers to any action parts ready for execution. These actions parts were called the invoked knowledge sources. A knowledge source became invoked when its condition part was satisfied. The condition parts and the invoked knowledge sources on the scheduling queue were called activities. The scheduler calculated a priority for each activity at the start of each system cycle and executed the activity with the highest priority in that cycle.

In order to select the most productive activity (the most important and promising with the least amount of processing and memory requirements), the scheduler used experimentally derived heuristics to calculate the priority. These heuristics were represented as imbedded procedures within the scheduler. The information needed by the scheduler was provided in part by the condition part of each invoked knowledge source. The condition part provided a stimulus frame, a set of hypotheses that satisfied the condition; and a response frame, a stylized description of the blackboard changes the knowledge source action part might produce upon execution. For example, the stimulus frame might indicate a specific set of syllables, and the response frame would indicate an action that would produce a word. The scheduler used the stimulus-response frames and other information on the blackboard to select the next thing to do.

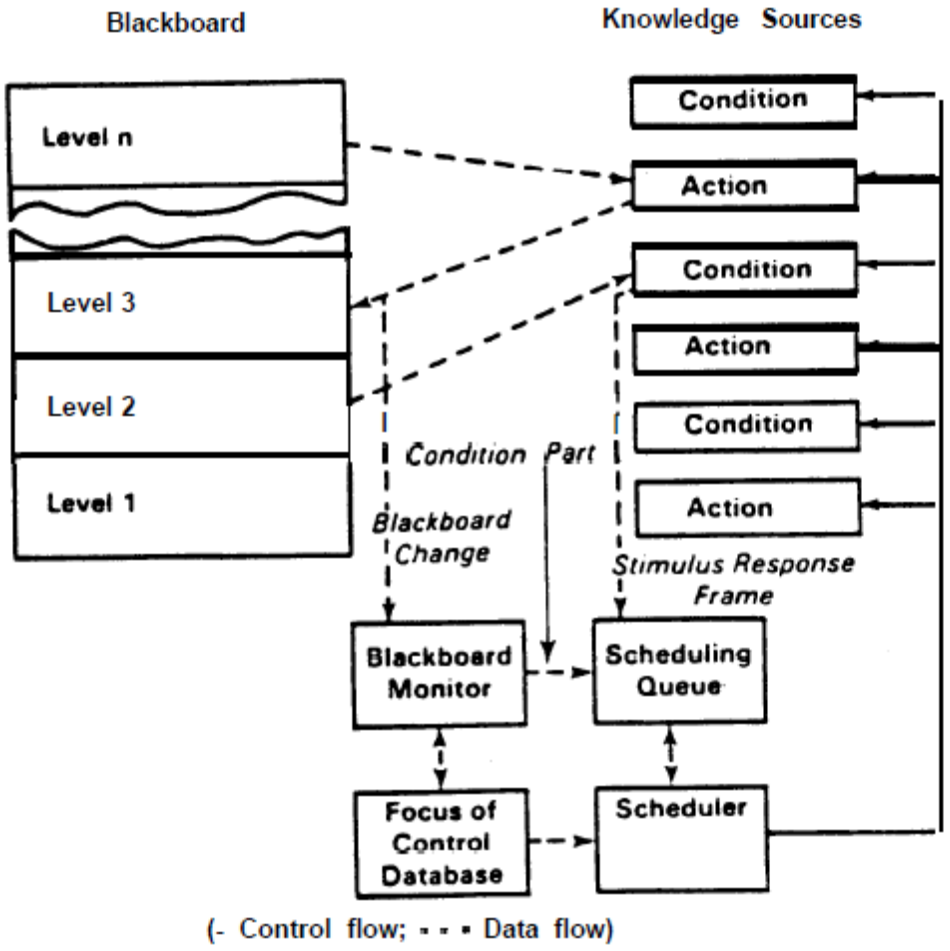


Figure 3.5: Schematic of HEARSAY-II Architecture

The control component iteratively executed the following basic steps:

1. The scheduler selected from the scheduling queue an activity to be executed.
2. If a condition part was selected and executed and if it was satisfied, a set of stimulus-response frames was put on the scheduling queue together with *a pointer to the invoked knowledge source.:
3. If an action part was selected and executed, the blackboard was modified. The blackboard monitors posted pointers to the condition parts that could follow up the change on the scheduling queue.

The problem of focus of attention was defined in the context of this architecture as a problem of developing a method which minimized the total number of knowledge source executions and which achieved a relatively low rate of error. The focus-of-attention problem was viewed as a knowledge-scheduling problem as well as a resource-allocation problem? In order to control the problem-solving

behavior of the system, the scheduler needed to know the goals of the task and the strategies for knowledge application to be able to evaluate the next best move.

Although various general solutions to this problem have been suggested [lS], it appears that ultimately one needs a knowledge-based scheduler for the effective utilization of the knowledge sources.

3.2.5 Knowledge-Application Strategy

Within the system framework described earlier, HEARSAY-II employed two problem-solving strategies. The first was a bottom-up strategy whereby interpretations were synthesized directly from the data, working up the abstraction hierarchy. For example, a word hypothesis was synthesized from a sequence of phones. The second was a top-down strategy in which alternative sentences were produced from a sentential concept, alternative sequences of words from each sentence, alternative sequences of phones from each word, and so on. The goal of this recursive generation process was to produce a sequence on the parametric level that was consistent with the input data (that is, to generate a hypothetical solution and to test it against the data). Both approaches have the potential for generating a vast number of alternative hypotheses and with it a combinatorially explosive number of knowledge source activaiions. Problem-solving activity was, therefore, constrained by selecting only a limited subset of invoked knowledge sources for execution. The scheduling module thus played a crucial role within the HEARSAY -II system.

Orthogonal to the top-down and bottom-up approaches, HEARSAY-II employed a general hypothesize-and-test strategy. A knowledge source would generate hypotheses, and their validity would be evaluated by some other knowledge source. The hypothesis could be generated by a top-down analytic or a bottom-up synthetic approach. Often, a knowledge source generated or tested hypotheses by matching its input data against a “matching prototype” in its knowledge base. For example, a sequence of hypothesized phones on the phone level were matched against a table containing prototypical patterns of phones for each word in the vocabulary. A word whose phones satisfied a matching criterion became a word hypothesis for the phones.

The validation process involved assigning credibility to the hypothesis based on the consistency of interpretation with the hypotheses on an adjacent level.

At each problem-solving step, any one of the bottom-up synthesis, top-down goal generation, neighborhood prediction, hypothesis generation, and hypothesis evaluation might have been initiated. The decision about whether a knowledge source could contribute to a solution was local to the knowledge source (precondition). The decision about which knowledge source should be executed in which one of many contexts was global to the solution state (the blackboard), and the decision was made by a global scheduler. The scheduler was opportunistic in choosing the next step, and the solution was created one step at a time.

Additional Notes

1. The condition parts of the knowledge sources were complex, CPU-intensive procedures that needed to search large areas of the blackboard. Each knowledge source needed to determine what changes had been made since the last time it viewed the blackboard. To keep from firing the condition parts continually, each condition part declared a priori the kinds of blackboard changes it was interested in. The condition part, when executed, looked only at the relevant

changes since the last cycle. All the changes that could be processed by the action part were passed-to it to avoid repetitive executions of the action part.

2. The HEARSAY-II system maintained alternative hypotheses. However, the maintenance and the processing of alternatives are always complex and expensive, especially when the system does not provide a general support for this. In HEARSAY-II, the problem was aggravated by an inadequate network structure that did not allow the shared network to be viewed from different perspectives. In the current jargon, it did not have good mechanisms for processing multiple worlds.
3. The evidence to support a hypothesis at a given level can be found on lower levels or on higher levels. For example, given a word hypothesis, its validity could be supported by a sequence of syllables or by grammatical constraints. The evidential support is represented by directional links from the evidence to the hypothesis it supports. The link that goes from a higher-level to a lower-level hypothesis represents a '*support from above" (that is, the justification for the hypothesis can be found at a higher level). A link that goes in the opposite direction represents support from below (that is, the reason for the hypothesis can be found at a lower level). Although the names of the support mechanisms were first coined in HASP [341, the bidirectional reasoning mechanisms were first used in the HEARSAY-II system.
4. In HEARSAY-II the confidence in a hypothesis generated by a knowledge source was represented by an integer between 1 and 100. The overall confidence in the hypothesis was accumulated by simple addition of the confidence attached to the evidence (that is, supporting hypotheses). When the confidence in a hypothesis was changed, the change was propagated up (if the support was from below) and down (if the support was from above) the entire structure.

4.0 Conclusion

This unit introduces the Blackboard model based expert systems. HEARSAY-II was used as a case study to illustrate how the blackboard model was implemented. The blackboard structure of HEARSAY-II was explained, the structure of the knowledge source was mentioned. Finally, the HEARSAY-II knowledge-application strategy was also discussed.

5.0 Summary

We hope you enjoyed this unit. This unit discussed Blackboard based expert systems. Now, let us attempt the questions below.

6.0 Tutor Marked Assignment

i. Read about the CRYSLIS system and make a comparative discussion between HEARSAY-II and CRYSALIS system. (Note: The task of the CRYSALIS system was to infer the three-dimensional structure of protein molecules.)

7.0 References/Further Readings

1) Nagori, V., & Trivedi, B. (2014). Types of Expert System: Comparative Study. Asian Journal of Computer and Information Systems (ISSN: 2321–5658), 2(02).
2) Nii, H. P. (1986). Blackboard Systems (No. KSL-86-18). STANFORD UNIV CA KNOWLEDGE SYSTEMS LAB.
3) Buchanan, B. G. (1981). Research on expert systems (No. STAN-CS-81-837). STANFORD UNIV CA DEPT OF COMPUTER SCIENCE.
4) Liebowitz, J. (Ed.). (1997). The handbook of applied expert systems. Crc Press.

Unit Three: Frame Based; KEE

CONTENT

1.0 Introduction

In rule base expert system and fuzzy expert system, IF- THEN rules are used to represent knowledge. In frame based expert system, frames are used to represent knowledge. Now we will try to explore the concept of frames.

2.0 Objectives

At the end of this unit, you should be able to:

- Define frame as it relates to data structures in expert systems
- Define slots as it relates to data structures in expert systems
- Outline the possible attributes of a slot
- Define facet as it relates to data structures in expert systems
- Outline the steps of developing frame based expert systems
- Outline the uses of frame in KEE
- Discuss how control of reasoning is achieved with KEE

3.0 Main Content

3.1 Frame

A frame is a data structure with typical knowledge about a particular object or concept. Frames are used to capture and represent knowledge in a frame based expert system. (Minsky, 1975) Each frame has its own name and set of attributes or slots associated with it.

The frame provides a natural way for the structured and concise representation of knowledge. We can combine all necessary knowledge about a particular object or concept in a single entity. In general frames are an application of object-oriented programming for expert systems. The advantage what frames offer over the rule base is that we just need to search through frames only to execute rules unlike of rule base expert system, where it goes through systematic search through all rules for execution.

3.1.1 Frame as a knowledge representation technique

The concept of a frame is defined by a collection of slots. Each slot describes a particular attribute or operation of the frame. Slots are used to store values. A slot may contain a default value or a pointer to another frame, a set of rules or procedure by which the slot value is obtained. (Negnevitsky, 2008)

In general, slot may include the following information:

1) Frame name

2) Relationship of the frame to the other frames.

3) Slot value

4) Default slot value

5) Range of slot value

6) Procedural information: A procedure is executed if the slot value is changed or needed. There are two types of procedures attached to slots.

a. When changed procedure is executed when new information is placed in the slot.

b. When needed procedure is executed when information is needed for the problem solving but the slot value is unspecified. Such procedural attachments are called demons.

Frame based expert systems also provide an extension to the slot value structure through the application of facets.

3.1.2 Facet

A facet is a means of providing extended knowledge about an attribute of a frame. Facets are used to establish the attribute value, end-user queries, and tell the inference engine how to process the attribute.

There are three kinds of facets which can be attached with frame based expert system. They are value facets, prompt facets and inference facets. Value facet specifies default and initial values of an attribute. Prompt facets enable the end-user to enter the attribute value on-line during a session with the expert

system. And finally inference facet allows us to stop the inference process when the values of a specified attribute changes.

3.1.3 Methods and demons

Frames provide us structured way of representing knowledge. But what should we do when we want to validate and manipulate the knowledge? The answer to that lies in methods and demons.

Method is a procedure associated with a frame attribute that is executed whenever requested. (Durkin, 1994) Method is represented by a series of commands similar to a macro in Microsoft excel.

In general, demons have an IF-THEN structure. It is executed whenever an attribute in the demon's IF statement changes its value.

3.1.4 Inference engine

In a frame based expert system, the inference engine searches for the goal or a specific attribute. In a frame based expert system, rule plays a secondary role. Knowledge is stored in frames and both methods and demons are used to add action to the frames. (Negnevitsky, 2008) The inference engine find those rules whose consequents contain the goal of interest and examine them one by one in order of rule base. If all the rules are valid, then inference engine will conclude that goal is reached else if any of the antecedents are invalid, then it is concluded that goal is not reached.

3.1.5 Frame based expert system development process

The steps in the frame based expert system development process are as follows: (Negnevitsky, 2008)

i. Specify the problem and define the scope of the system.
ii. Determine the classes and their attributes.
iii. Define instances.
iv. Design displays
v. Define when changed and when needed methods, and demons.
vi. Define rules.
vii. Evaluate and expand the system.

3.2 KEE

KEE is actually a large set of well-integrated AI paradigms. It includes the production rules formalism, a frame-based language with inheritance, logic-oriented assertions, object-oriented programming paradigms with message passing, and access to the host LISP system. The first strong point of KEE, as we will see, is the way it integrates these paradigms. KEE also provides means for organizing and aggregating knowledge into specific components and for explicitly structuring the reasoning process. The third strong point lies in the power and the user-friendliness of the interface.

3.2.1 KEE Knowledge Representation Formalisms

KEE is a frame-based environment. Frames in KEE are called units and present an extended scope.

- First, frames can be used for the taxonomic description of objects.

- Second, frames can have a procedural role and can enable behavioral models of objects and expertise to be built. To this end, active values and methods can be attached to slots. Active values can, for instance, selectively activate rule sets.
- Third, frames are also used to represent and encapsulate rules within structured hierarchies of classes with inheritance properties.
- Also, the refining nature of the inheritance lattice can be used in performing implicit inferences and in guiding the reasoning.

The frame-based language of KEE thus allows the representation of behavioral models of independent complex entities as required by the model-based reasoning approaches. It permits the decomposition of knowledge into task-oriented components. For instance, it allows the hierarchical division of the production rules into a lattice of frames with specific roles. Each knowledge component can be activated on request. This satisfies the main requirement of the structured reasoning approach: partitioning of the knowledge base into role specific components with selective access.

Before showing how the frame system of KEE offers satisfactory ways of structuring and managing the reasoning, we present the KEE frame formalism in more detail and list the technical features that differ from those of used in rule based systems like ART. (ART is currently one of the most innovative expert system tools with respect to the technology of rule-based systems.)

Descriptions of objects and rules are represented in class/subclass hierarchies of frames. Inheritance might be multiple and can be overridden in lower classes. Two main inheritance relations are available:

classes/subclasses and class/instance relationships. Each object is made up of slots; the system differentiates between own slots and member slots. Own slots are used to describe attributes or properties of the class considered as an object by its own, whereas member slots describe generic properties of the class members. Slots may have different facets that can have multiple values. Constraints on their class membership and on their cardinality may be stated (together with boolean combination of classes and restrictions on their range). These constraints may be used as automatic utilities for checking the integrity of knowledge. This differs from ART, where those kinds of constraints could be expressed only by using rules.

Active values can be associated with slots; they can be used for side-effect purposes (e.g., to activate level change in diagrams when corresponding values in the knowledge base are accessed). Functions can be activated (e.g., to compute the value of the slots). Rule classes can also be activated. The role of such facilities, which had to be expressed as rules in ART, will be discussed with the reasoning strategies of KEE.

The production rule formalism is well developed too. Variables and LISP function calls are allowed in both action and conditional parts.

3.2.2 Control of Reasoning with KEE

KEE provides several techniques for modeling reasoning. Basic search mechanisms are given. KEE also allows implicit knowledge to be wired into the frames and into the refinement nature of the inheritance lattice. This inheritance-based system gives a means of modeling both generalization- and specialization like operations of reasoning. Message passing can also be used in modeling the reasoning into organized strategies of subtasks.

State-Space Search

First, like ART, KEE provides the usual main search strategies of rule based expert systems: backward and forward chaining. It is also possible to mix chaining directions and get automatic backtracking ala PROLOG.

In this context, user-defined functions can be used to specify strategies such as depth first, breath first, and progressive deepening in the backward-chaining mode, whereas user-defined conflict resolution functions can be provided in the forward-chaining mode. These functions can be represented as slots in the rule classes. This allows greater flexibility because it permits the application of different strategies for different problem components, both in advance and dynamically.

KEE (release 3.0) is now provided with an assumption-based truth maintenance system called KEEWorld. According to IntelliCorp, this provides capabilities for supporting the fundamental problem-solving paradigms of state-space searching. KEE has been enhanced with the addition of three new types of rules. It now provides both what-if capabilities and assumption-based truth maintenance system facilities in the same manner as ART.

Frame-Based System and Reasoning

We have already discussed how the frame formalism of KEE can be used in the representation of knowledge. It provides a way of structuring composite entities and sets of rules into a hierarchy of different task-oriented knowledge components.

Strong points of KEE concern the ways in which the reasoning process can be wired, operated, or guided with the help of the frame component.

First, the inheritance lattice of frames allows several kinds of relations between objects to be stated (mainly class inclusion and membership relations).

The refining nature of the inheritance paradigms allows some information to be made implicit in the representation. For instance, a member of an inferior class inherits by default the properties stated for more general classes. The system is provided with the abilities for automatic retrieval of this implicit information.

This information is said to be wired in the representational machinery of frames (Fikes and Kehler, 1985). Such knowledge is made explicit in ART at compilation time. It is also possible to wire in a frame structure some procedural information that involves reasoning. For instance, active values can be attached to frame slots and can act as automatic constraints on the values of the slots each time they are modified. Thus, an active value operates a form of automatic reasoning.

KEE is provided with efficient automatic retrieval and automatic checking capabilities in this information lattice. To this end, Tell and Ask is a logic-based language that is used in asserting facts and retrieving facts from KEE knowledge bases. As such, it mimics standard data base retrieval functions and makes use of the wired reasoning capabilities of the inheritance lattices in order to find the implicit information and take into account the wired procedural knowledge. It is used both for direct interaction with the user and in the rule evaluation process.

With respect to the good-quality software objective of the structured approach, one should restrict oneself to using active values for entity local reasoning (type checking, access to external data bases, etc.) or in an access-oriented way for adapting display diagrams. Indeed, because the strict distinction between reasoning and knowledge is no longer conserved, the user should be careful not to make the system unreliable or unpredictable by introducing uncontrolled procedural-oriented reasoning.

The frame language also gives basic mechanisms with which basic steps of reasoning can be modeled and assembled. Actually, two types of procedural knowledge can be used as a means of modeling steps of reasoning. Procedural knowledge can be a rule set or a LISP procedure (an active value or a method).

Control of reasoning can be elaborated with message-passing access-oriented programming and inheritance as basic paradigms. First, inheritance paradigms give a direct means of representing the refinement and generalization-like operations that underlies classification problems. More general frames can activate more specialized frames that are associated with subtasks or more precise tasks. Second, a method can be activated by sending a message to an object. At least, active values can activate either a rule set or a procedure each time a slot value is accessed or modified.

All these paradigms allowing forms of control of reasoning are lacking in ART, which restricts its reasoning paradigm to general search over production rules.

3.2.3 Graphical Development Environment of KEE

The user interface of KEE is very versatile and elaborate. It includes powerful editors, knowledge base browsers, explanation facilities, world browser, etc.

It is widely recognized as superior to that of ART (Richer, 1986; Wall et al., 1986). Once again, a simple demonstration would be worth more than a thousand words. We are not going to try to give the reader an inadequate description of such a high-quality user interface.

KEE provides the user with the graphics tools KEEpictures and ActiveImages. They help in building graphic images. The images can be attached to frame slots in such a way that a modification of a slot value causes a change in the picture and vice versa.

3.2.4 Evolution of KEE

First, we must note the recent evolution of KEE to cope with the new advances of the technology. KEE 2.1 had no hypothetical or assumption-based truth maintenance system. KEE 3.0 is now available with these new abilities. In this respect, its open architecture is an important factor in customizing and extending KEE. Parts of KEE are written in KEE itself. On the one hand, this is useful.for ensuring KEE's flexibility and ability to evolve. On the other hand, it can lead to run-time inefficiency.

Second, KEE is written in COMMONLISP. Unlike ART, KEE has not yet been ported to C. Up to now, IntelliCorp has not announced such a transport. However, a PC-host delivery system is available for KEE which allows end users to access an AI application from a PC while the actual application is running on a classical host system.

Third, IntelliCorp is currently developing specialized software packages to run on top of KEE. One such system, SIMKIT, has already appeared. It is designed for simulation problems.

3.2.5 RUN-TIME EFFICIENCY ISSUE

The run-time efficiency of an expert system is, of course, a vital issue, especially when large real-world or real-time applications are involved. Unfortunately, objective and precise information about the run-time efficiency of systems developed with the help of large toolkits is lacking. Significant benchmarks are difficult to tune, too. This is due mainly to the diversity and large range of both possible problems and available paradigms. Moreover, run-time results on small examples cannot be extended to large real-world applications.

However, some run-time efficiency results are now available for both KEE and ART. We will refer mainly to the benchmarks operated by Riley (1986) at NASA Johnson Space Center. Riley's results are now widely used, but often out of the original context. As Riley himself stated, his benchmark is just one of many possible benchmarks and does not fully test the capabilities of the expert system tools. To draw the conclusion that ART is a better expert system tool than KEE, or that KEE is a better expert system tool than ART, was beyond the scope of his benchmark. As Riley suggested, ART and KEE have completely different design philosophies and their strengths lie in different types of problems.

In brief, the benchmark operated by NASA showed different run-time results using several expert system tools on a small example. The problem tested was the classical monkey and bananas problem. This is a typical toy planning problem where a final goal (the monkey eats the bananas) is to be reached by generating an action plan. A production system consisting of 30 rules and requiring 81 to 86 rule firings was used to obtain the solution.

The results of the benchmarks were particularly bad for KEE 2.1 when a rule-based approach was used together with the object description facilities of KEE. Nevertheless, a test that modeled the reasoning with the help of the KEE 2.1 message-passing mechanism was shown to be particularly efficient.

Although some conclusions can be drawn from this benchmark, they must be mitigated because of the different philosophies of the two products.

- First, it must be noted that the benchmark was operated with compiled rules for ART and interpreted rules for KEE. This difference is due to the different philosophies of the products. ART was developed to perform fast execution of rules. In contrast, KEE aims at giving the best development environment.
  In this respect, the KEE development philosophy is based on the assumption that interpreted rules are superior to compiled rules during many stages of the development cycle, as the interpreted mode provides the high interactive level that programmers need during that cycle. However, it should be noted that a rule compiler for backward-chaining rules has been announced for KEE.
- The monkeys and bananas problem is a typical small problem that can be (and has been) addressed by the techniques of the shallow knowledge approach; it is not representative of large problems addressed in knowledge engineering.
  It must be stressed that an intrinsically fast mechanism for the execution of rules is of prime importance in the shallow knowledge approach. Such an approach is usually taken when the problem is reducible only to an often-disorganized set of rules on which a general search mechanism must operate. In a structured reasoning method, speed is obtained by explicitly

structuring the reasoning process in a composition of tasks operating on specific blocks of knowledge.

Although intrinsic speed (i.e., the number of rules that are fired per second) is still an important issue, it is no longer the first concern. When large applications are involved, favoring intrinsic speed with a general search mechanism can be less advantageous than favoring an organized solution relying on lower intrinsic speed in the treatment of rules. Moreover, the other software qualities, such as easy maintenance and reliability, are of course more favored in a structured reasoning approach.

- Some other technical results of the NASA experiment are that the dynamical creation of objects is time-expensive in KEE 2.1 and that backward-chaining rules are sometimes more efficient in KEE 2.1 for well-known forward-chaining problems.

  When the system is running on conventional hardware (i.e., not a LISP machine), another important factor in run-time efficiency is the level at which it is executed. For example, whether the tool is available only on top of a LISP language or directly coded in C might affect run-time results. The fact that ART 3.0 is now written in C might give it a strong advantage with respect to the shallow knowledge approach and with respect to KEE, which is still only available in COMMONLISP.

3.2.6 TWO DIFFERENT BEST COMPETENCES

ART is an extended rule-based development system. The main motivation that seems to have directed its design has stressed run-time performance in the treatment of rules. The Inference Corporation has succeeded with this objective; currently, ART and OPS-83 are the fastest rule-based expert system tools available.

The basic knowledge representation formalism of ART is the production rules paradigm. Its expressive power is well developed and rules are presented in a rich set of different and original types. In particular, such rules should be very useful with problems that involve time-dependent or what-if reasoning. In this respect, the Viewpoints mechanism also seems useful for debugging rules and simulating their execution. A purely declarative structured knowledge com ponent is also linked to the rules as a means of describing objects and relations between objects referred to by rules.

Thus, ART seems appropriate when a shallow approach to a problem is to be undertaken. It should be a convenient tool for problems involving time dependent or what-if reasoning. Problems that deliver a complete formalization into rules and that are elegantly solved via a state-space search are also good candidates for ART.

KEE is an extended frame-based development system. Besides its excellent graphical development environment, we find that its formalisms for representing and aggregating both declarative and procedural knowledge are powerful. Techniques for explicitly managing and structuring the reasoning process are also provided.

Its frame-based paradigms give many facilities with respect to the model-based reasoning approach. Frames can be used for structuring and encoding declarative and procedural knowledge into knowledge blocks and specific rule sets that can be selectively activated. Message passing and wired inheritance give powerful means of guiding the reasoning. All these well-integrated features allow both the knowledge and its manipulation (i.e., the reasoning) to be organized and structured. This is obviously of prime importance when maintainability, reliability, and run-time efficiency are needed.

Because of these features, we think that KEE is a suitable tool when a model-based or a structured reasoning approach can be undertaken.

A current deficiency of KEE compared to ART might be the intrinsic runtime inefficiency of its general rule evaluation mechanism. However, significant progress seems to have been achieved in KEE 3.0. Moreover, KEE allows the reasoning to operate selectively on specific and structured blocks of knowledge, which may make it more efficient for large applications. Nevertheless, the fact that KEE is still written in COMMONLISP appears to be a real shortcoming compared to the C version of ART.

4.0 Conclusion

This unit presents an overview of frame based expert systems and the steps of developing frame based expert systems. A case study of KEE was presented and how the frame method differs from rule-based method were highlighted. We have also shown how KEE best suits the model-based and structured reasoning methods.

5.0 Summary

We hope you enjoyed this unit. This unit discussed knowledge representation in frame based expert systems. Now, let us attempt the questions below.

6.0 Tutor Marked Assignment

i. Discuss how knowledge representation in Frame based expert systems differs from that used in rule-based systems.

7.0 References/Further Readings

1) Nagori, V., & Trivedi, B. (2014). Types of Expert System: Comparative Study. Asian Journal of Computer and Information Systems (ISSN: 2321–5658), 2(02).
2) Gregoire, E. (1988). Evaluation of the expert system tools KEE and ART: a case study. Applied Artificial Intelligence an International Journal, 2(1), 1-23.

Unit Four: Extensive independent study of recent development and the submission of a group proposal for the application of Expert System in different areas.

CONTENT

1.0 Introduction

In this unit, students will work in small teams to tackle a comprehensive AI design proposal for the application of Expert System in an area, building upon the overall course material offered for CIT 903. Throughout the course, students will be expected to complete milestones related to the design process including: problem definition, generation and evaluations of concepts, analysis and testing, and preparation of design documentation. Each team will be supervised by a faculty advisor with whom they will meet regularly. Submitted work will be reviewed both by the faculty advisor and the course coordinator.

2.0 Objectives

By the end of this course, students should understand and be able to apply sound AI design principles and methodology to the solution of an open-ended design problem. Students should also have good oral and written communication skills, and should be able to function effectively in a design team. In particular by the end of the course, students should be able to:

- Properly identify and define an AI problem requiring a significant component of design, analysis and synthesis and to carry it out in a professional manner with minimum supervision.
- Conduct background research on relevant existing and emerging technology.
- Generate multiple possible solutions using disciplinary knowledge
- Effectively evaluate alternatives and select the best one.
- Apply AI knowledge and tools from different disciplines to perform appropriate analysis.
- Schedule and manage a large design project including developing a realistic budget and a suitable Gantt chart for carrying out the tasks associated with the project and to adhere to all required deadlines.
- Produce professional quality design documentation, including detailed drawings, software designs etc.
- Effectively communicate ideas in written and oral forms.
- Use appropriate computer tools to support all phases of the design process

3.0 Tutor Marked Assignment

i. The submission of a group proposal for the application of Expert System in different areas.

4.0 Further Readings

a. Russell, S., & Norvig, P. (2005). AI a modern approach. Learning, 2(3), 4.
b. Rich E., & Knight K. (1991) Artificial Intelligence, McGraw-Hill, New York, NY

c. Shi, Z. (2011). Advanced artificial intelligence (Vol. 1). World Scientific.
2) Rothman, D. (2018). Artificial Intelligence By Example: Develop machine intelligence from scratch using real artificial intelligence use cases. Packt Publishing Ltd.
3) Coppin, B. (2004). Artificial intelligence illuminated. Jones & Bartlett Learning.
4) Nilsson, N. J., & Nilsson, N. J. (1998). Artificial intelligence: a new synthesis. Morgan Kaufmann.
5) Rothman, D., Lamons, M., Kumar, R., Nagaraja, A., Ziai, A., & Dixit, A. (2018). Python: Beginner's Guide to Artificial Intelligence: Build applications to intelligently interact with the world around you using Python. Packt Publishing Ltd.
6) Lucas, P., & Van Der Gaag, L. (1991). Principles of expert systems. Wokingham: Addison-Wesley.
7) Liebowitz, J. (Ed.). (1997). The handbook of applied expert systems. Crc Press.
8) Poole, D. L., & Mackworth, A. K. (2017). Python code for Artificial Intelligence: Foundations of Computational Agents.
9) Poole, D. L., & Mackworth, A. K. (2010). Artificial Intelligence: foundations of computational agents. Cambridge University Press.