NATIONAL OPEN UNIVERSITY OF NIGERIA

FACULTY OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

COURSE CODE: CIT478

COURSE TITLE: ARTIFICIAL INTELLIGENCE

**COURSE
GUIDE**

CIT478
ARTIFICIAL INTELLIGENCE

Course Team   Dr. J.N. Ndunagu (Developer/Writer) - NOUN
                    Dr. J.N. Ndunagu (Coordinator) - NOUN

NATIONAL OPEN UNIVERSITY OF NIGERIA

CONTENTS   **PAGE**

Introduction

Welcome to CIT478 Artificial Intelligence which is a two credit unit course offered in the fourth year to students of the undergraduate degree programme in Communication Technology and Computer Science. There are eleven study Units in this course. There are no prerequisites for studying this course. It has been developed with appropriate local and foreign examples suitable for audience.

This course guide is for distance learners enrolled in the B.Sc. Communication Technology and Computer Science programmes of the National Open University of Nigeria. This guide is one of the several resource tools available to you to help you successfully complete this course and ultimately your programme.

In this guide you will find very useful information about this course, aims and objectives, what the course is about, what course materials you will be used, available services to support your learning, information on assignments and examination. It also offers you guidelines on how to plan your time for study the amount of time you are likely to spend on each study unit as well as your tutor-marked assignments.

I strongly recommend that you go through this course guide and complete the feedback form at the end before you begin studying the course. The feedback form must be submitted to your tutorial facilitator along with your first assignment.

I wish you all the best in your learning experience and successful completion of this course.

What You Will Learn in This Course

The overall aim of this course, CIT478 is to introduce you to artificial Intelligence and the different faculties involved in it. It also examines different ways of approaching AI. It starts with the basics and then moves on to the more advanced concepts. The Search in artificial Intelligence - State Space Search, uninformed Search, informed Search Strategies and tree Search are also treated. You will also learn about Knowledge Representation and programming languages for AI. Finally,

you will be introduced to Artificial Intelligence and its applications – Expert System and Robotics.

Course Aim

This course aims at introducing you to Artificial Intelligent (AI), different types of intelligent agents (IA) and types of AI search. You are not expected to have experience in Artificial Intelligent before using this course material. It is hoped that the knowledge would help you solve some real world problems.

Course Objectives

In order to achieve this aim, the course has a set of objectives. Each unit has specific objectives which are included at the beginning of the unit. You are expected to read these objectives before you study the unit. You may wish to refer to them during your study to check on your progress. You should always look at the unit objectives after completion of each unit. By doing so, you would have followed the instructions in the unit. Below are the comprehensive objectives of the course as a whole. By meeting these objectives, you should have achieved the aim of the course. Therefore, after going through this course you should be able to:

- State the definition of Artificial Intelligence
- List the different faculties involved with intelligent behavior
- Explain the different ways of approaching AI
- Look at some example systems that use AI
- Describe the history of AI
- Explain what an agent is and how it interacts with the environment.
- Identify the percepts available to the agent and the actions that the agent can execute, if given a problem situation
- Measure the performance used to evaluate an agent
- State based agents
- Identify the characteristics of the environment
- Describe the state space representation.
- Describe Some algorithms

- Formulate, when given a problem description, the terms of a state space search problem
- Analyze the properties of Some algorithms
- Analyze a given problem and identify the most suitable search strategy for the problem.
- Solve Some Simple problems
- Explain Uninformed Search
- List two types of Uninformed Search
- Describe Depth First and Breadth First Search
- Solve simple problems on Uninformed Search
- Explain informed Search
- Mention other names of informed Search
- Describe Best-first Search
- Describe Greedy Search
- Solve simple problems on informed Search
- Describe a Game tree
- Describe Some Two-Player Games Search Algorithms
- Explain Intelligent Backtracking
- Solve Some Simple problems on tree search.
- Explain the meaning of Knowledge Representation
- **Describe the history of History of knowledge representation and reasoning**
- **List some Characteristics of KR**
- **List 4 main features of KR language**
- Describe the History of IPL
- Discuss the similarities between Lisp and Prolog Programming
- list the areas where Lisp can be used
- Describe the history of natural language processing
- List major tasks in NLP
- Mention different types of evaluation of NPL
- Explain an Expert System
- Distinction between expert systems and traditional problem solving programs
- Explain the term "Knowledge Base"
- Explain the word Robotics
- List 4 types of Robotics you know
- Describe the history of Robotics

Working through This Course

To complete this course, you are required to read each study unit, read the textbooks and read other materials which may be provided by the National Open University of Nigeria.

Each unit contains tutor marked assignments and at certain points in the course you would be required to submit assignment for assessment purposes. At the end of the course there is a final examination. The course should take you about a total of eleven (11) weeks to complete. Below is the list of all the components of the course, what you have to do and how you should allocate your time to each unit in order to complete the course on time and successfully.

This course entails that you spend a lot of time to read and practice. For easy understanding of this course, I will advise that you avail yourself the opportunity of attending the tutorials sessions where you would have the opportunity to compare your knowledge with that of other people, and also have your questions answered.

The Course Material

The main components of this course are:

1.   The Course Guide
2.   Study Units
3.   Further Reading/References
4.   Assignments
5.   Presentation Schedule

Study Units

There are 11 study units and 4 modules in this course. They are:

Module 1   Introduction to AI

Unit 1   What is Artificial Intelligent (AI)?
Unit 2   Introduction to Intelligent Agent (IA)

Module 2   Search in Artificial Intelligence

Unit 1   Introduction to State Space Search
Unit 2   Uninformed Search

Textbooks and References

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

Adrian Walker; Michael McCord; John F. Sowa and Walter G. Wilson
     (1990). *Knowledge Systems and Prolog* (Second Edition).
     Addison-Wesley.

Argumentation in Artificial Intelligence by Iyad Rahwan, Guillermo R.
     Simari

Arthur B. Markman (1998). *Knowledge Representation.* Lawrence
     Erlbaum Associates.

Asimov, Isaac (1996) [1995]. "The Robot Chronicles". Gold. London:
     Voyager. pp. 224–225. ISBN 0-00-648202-3.

Bates, M. (1995). Models of Natural Language Understanding.
     Proceedings of the National Academy of Sciences of the United
     States of America, Vol. 92, No. 22 (Oct. 24, 1995), pp. 9977–
     9982.

Bowling, M. and Veloso, M. (2002). Multiagent Learning Using a Variable Learning Rate *Artificial Intelligence*, 136(2): 215-250.

Chein, M. & Mugnier, M.-L. (2009). *Graph-based Knowledge Representation: Computational Foundations of Conceptual Graphs*, Springer, 2009,ISBN 978-1-84800-285-2.

Christopher D. Manning, Hinrich Schütze (1999). Foundations of Statistical Natural Language Processing, MIT Press, ISBN 978-0-262-13360-9, p. Xxxi.

Crane, Carl D.; Joseph Duffy (1998-03). Kinematic Analysis of Robot Manipulators. Cambridge University Press. ISBN 0521570638. http://www.cambridge.org/us/catalogue/catalogue.asp?isbn=0521 570638.

Crevier, Daniel (1993). *AI: The Tumultuous Search for Artificial Intelligence*. New York, NY: Basic Books, ISBN 0-465-02997-3.

Davis, R. Shrobe, H.E. Representing Structure and Behavior of Digital Hardware, IEEE Computer, Special Issue on Knowledge Representation, 16(10):75-82.

Dechter, Rina; Judea Pearl (1985). "Generalized best-first search strategies and the optimality of A*". *Journal of the ACM* 32 (3): 505–536. doi:10.1145/3828.3830.

Dowe, D.L. and Hajek, A. R. (1997). "A Computational Extension to the Turing Test". *Proceedings of the 4th Conference of the Australasian Cognitive Science jSociety*. http://www.csse.monash.edu.au/publications/1997/tr-cs97-322-abs.html.

Hermann Helbig: *Knowledge Representation and the Semantics of Natural Language*. Springer, Berlin, Heidelberg, New York 2006

Jean-Luc Hainaut, Jean-Marc Hick, Vincent Englebert, Jean Henrard, Didier Roland: Understanding Implementations of IS-A Relations. ER 1996: 42-57

Jeen Broekstraa, Michel Klein, Stefan Deckerc, Dieter Fenselb, Frank van Harmelenb and Ian Horrocks Enabling knowledge representation on the Web by extending RDF Schema, , April 16 2002.

John F. Sowa (2000). *Knowledge Representation*: Logical, Philosophical, and Computational Foundations. New York: Brooks/Cole.

John McCarthy (1979). *History of Lisp* "LISP prehistory - Summer 1956 through Summer 1958."

Jose H. (2000). "Beyond the Turing Test". *Journal of Logic, Language and Information* 9 (4): 447–466. doi:10.1023/A:1008367325700. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.44.8943.

Koenig, Sven; Maxim Likhachev, Yaxin Liu, David Furcy (2004). "Incremental heuristic search in AI". *AI Magazine* 25 (2): 99–112. http://portal.acm.org/citation.cfm?id=1017140.

Lowerre, Bruce (1976). "The Harpy Speech Recognition System", Ph.D. thesis, Carnegie Mellon University.

Marakas, George. Decision Support Systems in the 21st Century. Prentice Hall, 1999, p.29.

McCarthy, John (November 12, 2007). "What Is Artificial Intelligence?". http://www-formal.stanford.edu/jmc/whatisai/whatisai.html

Michael Wooldridge, An Introduction to Multiagent Systems, John Wiley & Sons, Ltd.

Nilsson, N. J. (1980). *Principles of Artificial Intelligence*. Palo Alto, California: Tioga Publishing Company. ISBN 0-935382-01-1.

Nilsson, Nils (1998). *Artificial Intelligence: A New Synthesis*, Morgan Kaufmann Publishers, ISBN 978-1-55860-467-4.

Nishibori; *et al.* (2003). Robot Hand with Fingers Using Vibration-Type Ultrasonic Motors (Driving Characteristics). Journal of Robotics and Mechatronics. http://www.fujipress.jp/finder/xslt.php?mode= present&inputfile=ROBOT001500060002.xml.

OWL DL Semantics. http://www.obitko.com/tutorials/ontologies-semantic-web/owl-dl-semantics.html.

Park; *et al.* (2005). Synthetic Personality in Robots and Its Effect on Human-Robot Relationship.

Pearl, Judea (1984). *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley Longman Publishing Co., Inc.. ISBN 0-201-05594-5.

Philippe Martin "Knowledge representation in RDF/XML, KIF, Frame-CG and Formalized-English", , Distributed System Technology Centre, QLD, Australia, July 15-19, 2002

Poole, David; Mackworth, Alan; Goebel, Randy (1998). *Computational Intelligence: A Logical Approach*. New York: Oxford University Press, ISBN 0195102703, http://www.cs.ubc.ca/spider/poole/ci.html

Pople H, Heuristic Methods for Imposing Structure on Ill-Structured Problems, in AI in Medicine, Szolovits (ed.). AAAS Symposium 51, Boulder: Westview Press.

Randall Davis, Howard Shrobe, and Peter Szolovits; What Is a Knowledge Representation? AI Magazine, 14(1):17-33,1993

Ronald Fagin, Joseph Y. Halpern, Yoram Moses, Moshe Y. Vardi *Reasoning About Knowledge*. MIT Press, 1995, ISBN 0-262-06162-7.

Ronald J. Brachman, Hector J. Levesque (eds) *Readings in Knowledge Representation*, Morgan Kaufmann, 1985, ISBN 0-934613-01-X

Ronald J. Brachman, Hector J. Levesque *Knowledge Representation and Reasoning*, Morgan Kaufmann, 2004 ISBN 978-1-55860-932-7

Ronald J. Brachman; What IS-A is and Isn't. An Analysis of Taxonomic Links in Semantic Networks; IEEE Computer, 16 (10); October 1983.

Rosheim, Mark E. (1994). Robot Evolution: The Development of Anthrobotics. Wiley-IEEE. pp. 9–10. ISBN 0471026220.

Russell, S. J.; Norvig, P. (2003). *Artificial Intelligence: A Modern Approach*. Upper Saddle River, N.J.: Prentice Hall. pp. 97–104. ISBN 0-13-790395-2.

Russell, Stuart J.; Norvig, Peter (2003). *Artificial Intelligence: A Modern Approach* (2nd ed.), Upper Saddle River, New Jersey: Prentice Hall, ISBN 0-13-790395-2, http://aima.cs.berkeley.edu/

Russell, Stuart J.; Norvig, Peter (2003). *Artificial Intelligence: A Modern Approach* (2nd ed.), Upper Saddle River, New Jersey: Prentice Hall, pp. 111–114, ISBN 0-13-790395-2

Russell, Stuart J.; Norvig, Peter (2003). *Artificial Intelligence: A Modern Approach* (2nd ed.). Upper Saddle River, New Jersey: Prentice Hall, ISBN 0-13-790395-2, http://aima.cs.berkeley.edu/

Russell, Stuart J.; Norvig, Peter (2010). Artificial Intelligence: A Modern Approach (3rd ed.), Upper Saddle River, New Jersey: Prentice Hall, ISBN 0-13-604259-7, p. 437-439.

Sebesta, Robert W. (1996). *Concepts of Programming Languages,* (Third Edition). Addison-Wesley Publishing Company, Menlo Park, California.

Serenko, Alexander; Detlor, Brian (2004). "Intelligent agents as innovations". *AI and Society* 18 (4): 364–381. doi:10.1007/s00146-004-0310-5. http://foba.lakeheadu.ca/serenko/papers/Serenko_Detlor_AI_and_Society.pdf

Serenko, Alexander; Ruhi, Umar; Cocosila, Mihail (2007). "Unplanned effects of intelligent agents on Internet use: Social Informatics approach". *AI and Society* 21 (1–2): 141–166. doi:10.1007/s00146-006-0051-8. http://foba.lakeheadu.ca/serenko/papers/AI_Society_Serenko_Social_Impacts_of_Agents.pdf

Shapiro, Stuart C. (1992). "Artificial Intelligence". In Shapiro, Stuart C.. *Encyclopedia of Artificial Intelligence* (2nd ed.). New York: John Wiley. pp. 54–57. ISBN 0471503061. http://www.cse.buffalo.edu/~shapiro/Papers/ai.pdf.

Skillings, J. (2006). "Getting Machines to Think Like Us". *cnet*. http://news.cnet.com/Getting-machines-to-think-like-us/2008-11394_3-6090207.html.

Stuart Russell; Peter Norvig (2010). *Artificial Intelligence: A Modern Approach* (3 ed.). Prentice Hall. ISBN 978-0-13-6042594

Turing, Alan (1950), "Computing Machinery and Intelligence", *Mind* LIX (236): 433–460, doi:10.1093/mind/LIX.236.433, ISSN 0026-4423, http://loebner.net/Prizef/TuringArticle.html.

Yucong Duan, Christophe Cruz (2011). Formalizing Semantic of Natural Language through Conceptualization from Existence. International Journal of Innovation, Management and Technology (2011) 2 (1), pp. 37-42.

Zhou, Rong. Hansen, Eric (2005). "Beam-Stack Search: Integrating Backtracking with Beam Search". http://www.aaai.org/Library/ICAPS/2005/icaps05-010.php

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 11 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 11 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 CIT478 will be of last for a period of 2 hours and have a value of 60% of the total course grade. The examination will consist of questions which reflect the 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 1-11	11 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 the work	Weeks Activity	Assessment (End of Unit)
	Course Guide	Week 1	
Module 1	**Introduction to AI**		
1	What Is AI?	Week 1	Assessment 1
2	Introduction to Agent	Week 2	Assessment 2
Module 2	**Search in Artificial Intelligence**		
1	Introduction to State Space Search	Week 3	Assessment 3
2	Uninformed Search	Week 4	Assessment 4
3	- Informed Search Strategies	Week 5	Assessment 5
4	Tree Search	Week 6	Assessment 6
Module 3	**Knowledge Representation and Programming Languages for AI**		
1	Knowledge Representation	Week 7	Assessment 7
2	Programming Languages for Artificial Intelligence	Week 8	Assessment 8
3	– Natural Language Processing	Week 9	Assessment 9
Module 4	**Artificial Intelligence and the Future**		
1	Expert System	Week 10	Assessment 10
2	Robotics	Week 11	Assessment 11

How to Get the Best from This Course

In distance learning, the study units replace the university lecturer. This is one of the great advantages of distance learning; you can read and work through specially designed study materials at your own pace, and at a time and place that suit you best. Think of it as reading the lecture instead of listening to a lecturer. In the same way that a lecturer might set you some reading to do, the study units tell you when to read your set books or other material. Just as a lecturer might give you an in-class exercise, your study units provide exercises for you to do at appropriate points.

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 is a set of learning objectives. These objectives enable you know what you should be able to do by the time you have completed the unit. You should use these objectives to guide your study. When you have finished the units you must go back and check whether you have achieved the objectives. If you make a habit of doing this you will significantly improve your chances of passing the course.

Remember that your tutor's job is to assist you. When you need help, don't hesitate to call and ask your tutor to provide it.

- Read this *Course Guide* thoroughly.
- 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 assignments relate to the units. Whatever method you chose to use, you should decide on it and write in your own dates for working on each unit.

- 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 lag behind in their course work.
- Turn to *Unit 1* and read the introduction and the objectives for the unit.
- Assemble the study materials. Information about what you need for a unit is given in the 'Overview' at the beginning of each unit. You will almost always need both the study unit you are working on and one of your set of books on your desk at the same time.

- 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 instructed to read sections from your set books or other articles. Use the unit to guide your reading.
- Review the objectives for each study unit to confirm that you have achieved them. If you feel unsure about any of the objectives, review the study material or consult your tutor.
- When you are confident that you have achieved a unit's objectives, you can then start on the next unit. Proceed unit by unit through the course and try to pace your study so that you keep yourself on schedule.
- 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 on the tutor-marked assignment form. Consult your tutor as soon as possible if you have any questions or problems.
- 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*).

Facilitators/Tutors and Tutorials

There are 11 hours of tutorials provided in support of this course. You will be notified of the dates, times and location of these tutorials, 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 or submit your tutor-marked assignments to your tutor well before the due date (at least two working days are required). They will be marked by your tutor and returned to you as soon as possible.

- Do not hesitate to contact your tutor by telephone, or e-mail 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 a question or problem 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 to ask questions which are answered instantly. You can raise any problem encountered in the course of your study. To gain the maximum benefit from course tutorials, prepare a question list before attending them. You will learn a lot from participating in discussions actively. GOODLUCK!

Course Code   CIT478
Course Title   Artificial Intelligence

Course Team   Dr. J.N. Ndunagu (Developer/Writer) - NOUN
               Dr. J.N. Ndunagu (Coordinator) - NOUN

NATIONAL OPEN UNIVERSITY OF NIGERIA

CONTENTS   PAGE

MODULE 1   INTRODUCTION TO AI

Unit 1   What Is Artificial Intelligent (AI)?
Unit 2   Introduction to Intelligent Agent (IA)

UNIT 1   WHAT IS ARTIFICIAL INTELLIGENT (AI)?

CONTENTS

1.0   INTRODUCTION

This unit introduces you to Artificial Intelligence and the different faculties involve in it. It also examines different ways of approaching AI.

2.0   OBJECTIVES

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

•   state the definition of Artificial Intelligence
•   list the different faculties involved with intelligent behavior
•   explain the different ways of approaching AI
•   look at some example systems that use AI
•   describe the history of AI.

3.0   MAIN CONTENT

3.1   Definition of AI

What is AI?
Artificial Intelligence is a branch of *Science* which deals with helping machines find solutions to complex problems in a more human-like fashion. This generally involves borrowing characteristics from human intelligence, and applying them as algorithms in a computer friendly way. A more or less flexible or efficient approach can be taken depending on the requirements established, which influences how artificial the intelligent behaviour appears.

AI is generally associated with *Computer Science*, but it has many important links with other fields such as *Mathematics*, *Psychology*, *Cognition*, *Biology* and *Philosophy*, among many others. Our ability to combine knowledge from all these fields will ultimately benefit our progress in the quest of creating an intelligent artificial being
It is also concerned with the design of intelligence in an artificial device. The term was coined by McCarthy in 1956. There are two ideas in the definition.

1.   Intelligence
2.   Artificial device

What is intelligence?

-    Is it that which characterize humans? Or is there an absolute standard of judgment?
-    Accordingly there are two possibilities:
-    A system with intelligence is expected to behave as intelligently as a human
-    A system with intelligence is expected to behave in the best possible manner
-    Secondly what type of behavior are we talking about?
-    Are we looking at the thought process or reasoning ability of the system?
-    Or are we only interested in the final manifestations of the system in terms of its actions?

Given this scenario different interpretations have been used by different researchers as defining the scope and view of Artificial Intelligence.

1.   One view is that artificial intelligence is about designing systems that are as intelligent as humans. This view involves trying to

understand human thought and an effort to build machines that emulate the human thought process. This view is the cognitive science approach to AI.

2. The second approach is best embodied by the concept of the Turing Test. Turing held that in future computers can be programmed to acquire abilities rivaling human intelligence. As part of his argument Turing put forward the idea of an 'imitation game', in which a human being and a computer would be interrogated under conditions where the interrogator would not know which was which, the communication being entirely by textual messages. Turing argued that if the interrogator could not distinguish them by questioning, then it would be unreasonable not to call the computer intelligent. Turing's 'imitation game' is now usually called 'the Turing test' for intelligence.
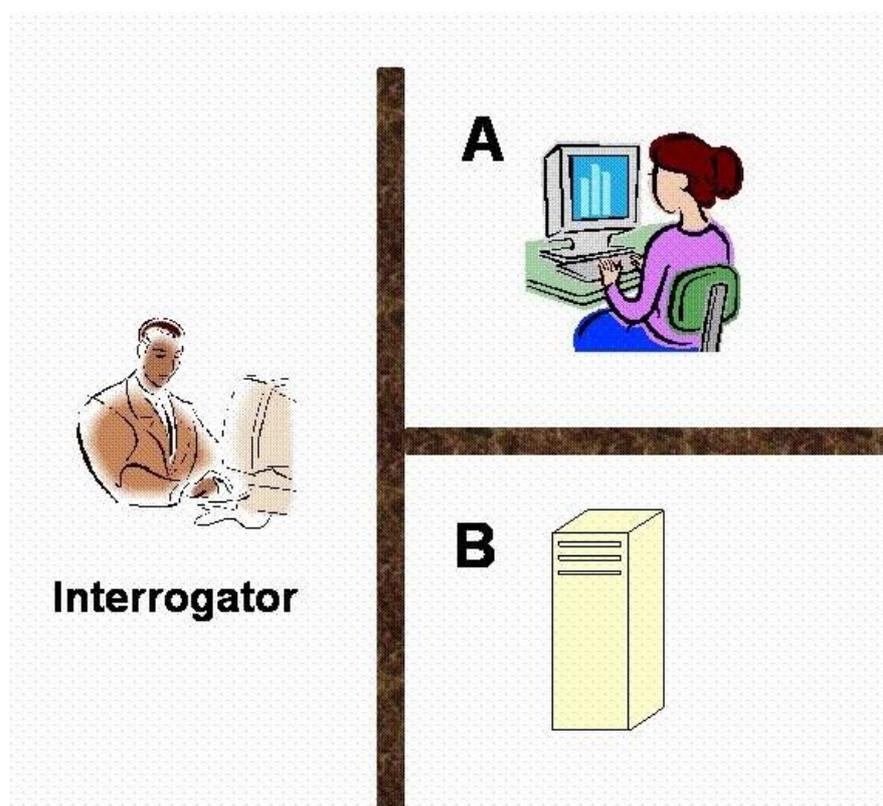

Figure 1:   Turing Test

Turing Test

Consider the following setting. There are two rooms, A and B. One of the rooms contains a computer. The other contains a human. The interrogator is outside and does not know which one is a computer. He can ask questions through a teletype and receives answers from both A

and B. The interrogator needs to identify whether A or B are humans. To pass the Turing test, the machine has to fool the interrogator into believing that it is human. For more details on the Turing test visit the site http://cogsci.ucsd.edu/~asaygin/tt/ttest.html

3.   Logic and laws of thought deals with studies of ideal or rational thought process and inference. The emphasis in this case is on the inferencing mechanism, and its properties. That is how the system arrives at a conclusion, or the reasoning behind its selection of actions is very important in this point of view. The soundness and completeness of the inference mechanisms are important here.

4.   The fourth view of AI is that it is the study of rational agents. This view deals with building machines that act rationally. The focus is on how the system acts and performs, and not so much on the reasoning process. A rational agent is one that acts rationally, that is, is in the best possible manner.

3.1.2 Typical AI problems

While studying the typical range of tasks that we might expect an "intelligent entity" to perform, we need to consider both "common-place" tasks as well as expert tasks.
Examples of common-place tasks include

-       *Recognizing* people, objects.
-       Communicating (through *natural language*).
-       *Navigating* around obstacles on the streets

These tasks are done matter of firstly and routinely by people and some other animals.

Expert tasks include:

•   Medical diagnosis.
•   Mathematical problem solving
•   Playing games like chess

These tasks cannot be done by all people, and can only be performed by skilled specialists.

Now, which of these tasks are easy and which ones are hard? Clearly tasks of the first type are easy for humans to perform, and almost all are able to master them. The second range of tasks requires skill

4

development and/or intelligence and only some specialists can perform them well. However, when we look at what computer systems have been able to achieve to date, we see that their achievements include performing sophisticated tasks like medical diagnosis, performing symbolic integration, proving theorems and playing chess.

On the other hand it has proved to be very hard to make computer systems perform many routine tasks that all humans and a lot of animals can do. Examples of such tasks include navigating our way without running into things, catching prey and avoiding predators. Humans and animals are also capable of interpreting complex sensory information. We are able to recognize objects and people from the visual image that we receive. We are also able to perform complex social functions.

Intelligent behaviour. This discussion brings us back to the question of what constitutes intelligent behaviour. Some of these tasks and applications are:

- Perception involving image recognition and computer vision
- Reasoning
- Learning
- Understanding language involving natural language processing, speech processing
- Solving problems
- Robotics

3.1.3 Practical Impact of AI

AI components are embedded in numerous devices e.g. in copy machines for automatic correction of operation for copy quality improvement. AI systems are in everyday use for identifying credit card fraud, for advising doctors, for recognizing speech and in helping complex planning tasks. Then there are intelligent tutoring systems that provide students with personalized attention

Thus AI has increased understanding of the nature of intelligence and found many applications. It has helped in the understanding of human reasoning, and of the nature of intelligence. It will also help you understand the complexity of modeling human reasoning.

You can now look at a few famous AI systems.

1.   ALVINN

Autonomous Land Vehicle in a Neural Network

In 1989, Dean Pomerleau at CMU created ALVINN. This is a system which learns to control vehicles by watching a person drive. It contains a neural network whose input is a 30x32 unit two dimensional camera image. The output layer is a representation of the direction the vehicle should travel.

The system drove a car from the East Coast of USA to the west coast, a total of about 2850 miles. Out of this about 50 miles were driven by a human being and the rest solely by the system.

2.   Deep Blue

In 1997, the Deep Blue chess program created by IBM, beat the current world chess champion, Gary Kasparov.

3.   Machine translation

A system capable of translations between people speaking different languages will be a remarkable achievement of enormous economic and cultural benefit. Machine translation is one of the important fields of endeavour in AI. While some translating systems have been developed, there is a lot of scope for improvement in translation quality.

4.   Autonomous agents

In space exploration, robotic space probes autonomously monitor their surroundings, make decisions and act to achieve their goals.
NASA's Mars rovers successfully completed their primary three-month missions in April, 2004. The Spirit rover had been exploring a range of Martian hills that took two months to reach. It is finding curiously eroded rocks that may be new pieces to the puzzle of the region's past. Spirit's twin, Opportunity, had been examining exposed rock layers inside a crater.

5.   Internet Agents

The explosive growth of the internet has also led to growing interest in internet agents to monitor users' tasks, seek needed information, and to learn which information is most useful

3.1.4 Approaches to AI

Strong AI aims to build machines that can truly reason and solve problems. These machines should be self aware and their overall intellectual ability needs to be indistinguishable from that of a human being. Excessive optimism in the 1950s and 1960s concerning strong AI has given way to an appreciation of the extreme difficulty of the problem. Strong AI maintains that suitably programmed machines are capable of cognitive mental states.

Weak AI deals with the creation of some form of computer-based artificial intelligence that cannot truly reason and solve problems, but can act as if it were intelligent. Weak AI holds that suitably programmed machines can simulate human cognition.

Applied AI aims to produce commercially viable "smart" systems such as, security system that is able to recognise the faces of people who are permitted to enter a particular building. Applied AI has already enjoyed considerable success.

Cognitive AI: computers are used to test theories about how the human mind works--for example, theories about how we recognise faces and other objects, or about how we solve abstract problems.

3.1.5 Limits of AI Today

Today's successful AI systems operate in well-defined domains and employ narrow, specialized knowledge. Common sense knowledge is needed to function in complex, open-ended worlds. Such a system also needs to understand unconstrained natural language. However these capabilities are not yet fully present in today's intelligent systems.

✓   What can AI systems do?

Today's AI systems have been able to achieve limited success in some of these tasks.

- In Computer vision, the systems are capable of face recognition
- In Robotics, we have been able to make vehicles that are mostly autonomous
- In Natural language processing, we have systems that are capable of simple machine translation
- Today's Expert systems can carry out medical diagnosis in a narrow domain

- Speech understanding systems are capable of recognizing several thousand words continuous speech
- Planning and scheduling systems had been employed in scheduling experiments with the Hubble Telescope
- The Learning systems are capable of doing text categorization into about a 1000 topics
- In Games, AI systems can play at the Grand Master level in chess (world champion), checkers, etc.

✓ What can AI systems NOT do yet?

- Understand natural language robustly (e.g., read and understand articles in a newspaper)
- Surf the web
- Interpret an arbitrary visual scene
- Learn a natural language
- Construct plans in dynamic real-time domains
- Exhibit true autonomy and intelligence

3.2   AI History

Intellectual roots of AI date back to the early studies of the nature of knowledge and reasoning. The dream of making a computer imitate humans also has a very early history.

The concept of intelligent machines is found in Greek mythology. There is a story in the 8th century A.D about Pygmalion Olio, the legendary king of Cyprus. He fell in love with an ivory statue he made to represent his ideal woman. The king prayed to the goddess Aphrodite, and the goddess miraculously brought the statue to life. Other myths involve human-like artifacts. As a present from Zeus to Europa, Hephaestus created Talos, a huge robot. Talos was made of bronze and his duty was to patrol the beaches of Crete.

Aristotle (384-322 BC) developed an informal system of syllogistic logic, which is the basis of the first formal deductive reasoning system.

Early in the 17th century, Descartes proposed that bodies of animals are nothing more than complex machines.

Pascal in 1642 made the first mechanical digital calculating machine.

In the 19th century, George Boole developed a binary algebra representing (some) "laws of thought."

8

Charles Babbage & Ada Byron worked on programmable mechanical calculating machines.

In the late 19th century and early 20th century, mathematical philosophers like Gottlob Frege, Bertram Russell, Alfred North Whitehead, and Kurt Gödel built on Boole's initial logic concepts to develop mathematical representations of logic problems.

The advent of electronic computers provided a revolutionary advance in the ability to study intelligence.

In 1943 McCulloch & Pitts developed a Boolean circuit model of brain. They wrote the paper "A Logical Calculus of Ideas Immanent in Nervous Activity", which explained how it is possible for neural networks to compute.

Marvin Minsky and Dean Edmonds built the SNARC in 1951, which is the first randomly wired neural network learning machine (SNARC stands for Stochastic Neural-Analog Reinforcement Computer).It was a neural network computer that used 3000 vacuum tubes and a network with 40 neurons.

In 1950 Turing wrote an article on "Computing Machinery and Intelligence" which articulated a complete vision of AI. For more on Alan Turing see the site http://www.turing.org.uk/turing/ . Turing's paper talked of many things, of solving problems by searching through the space of possible solutions, guided by heuristics. He illustrated his ideas on machine intelligence by reference to chess. He even propounded the possibility of letting the machine alter its own instructions so that machines can learn from experience.

In 1956 a famous conference took place in Dartmouth. The conference brought together the founding fathers of artificial intelligence for the first time. In this meeting the term "Artificial Intelligence" was adopted.

Between 1952 and 1956, Samuel had developed several programs for playing checkers. In 1956, Newell & Simon's Logic Theorist was published. It is considered by many to be the first AI program. In 1959, Gelernter developed a Geometry Engine. In 1961 James Slagle (PhD dissertation, MIT) wrote a symbolic integration program SAINT. It was written in LISP and solved calculus problems at the college freshman level. In 1963, Thomas Evan's program Analogy was developed which could solve IQ test type analogy problems.

In 1963, Edward A. Feigenbaum & Julian Feldman published Computers and Thought, the first collection of articles about artificial intelligence.

In 1965, J. Allen Robinson invented a mechanical proof procedure, the **Resolution Method**, which allowed programs to work efficiently with formal logic as a representation language. In 1967, the Dendral program (Feigenbaum, Lederberg, Buchanan, Sutherland at Stanford) was demonstrated which could **interpret mass spectra on organic chemical compounds**. This was the first successful knowledge-based program for scientific reasoning. In 1969 the SRI robot, Shakey, demonstrated combining locomotion, perception and problem solving.
The years from 1969 to 1979 marked the early development of **knowledge-based systems**

In 1974, MYCIN demonstrated the power of rule-based systems for knowledge representation and inference in medical diagnosis and therapy. Knowledge representation schemes were developed. These included frames developed by Minski. Logic based languages like Prolog and Planner were developed.

We will now mention a few of the AI systems that were developed over the years.

The Meta-Dendral learning program produced new results in chemistry (rules of mass spectrometry)

In the 1980s, Lisp Machines developed and marketed.

Around 1985, neural networks return to popularity.

In 1988, there was a resurgence of probabilistic and decision-theoretic methods.

The early AI systems used general systems, little knowledge. AI researchers realized that specialized knowledge is required for rich tasks to focus reasoning.

The 1990's saw major advances in all areas of AI including the following:

- Machine learning, data mining
- Intelligent tutoring,
- Case-based reasoning,
- Multi-agent planning, scheduling,

- Uncertain reasoning,
- Natural language understanding and translation,
- Vision, virtual reality, games, and other topics.

Rod Brooks' COG Project at MIT, with numerous collaborators, made significant progress in building a humanoid robot.

The first official Robo-Cup soccer match featuring table-top matches with 40 teams of interacting robots was held in 1997. For details, see the site http://murray.newcastle.edu.au/users/students/2002/c3012299/bg.html

In the late 90s, Web crawlers and other AI-based information extraction programs become essential in widespread use of the world-wide-web.

Interactive robot pets ("smart toys") become commercially available, realizing the vision of the 18th century novelty toy makers.

In 2000, the Nomad robot explores remote regions of Antarctica looking for meteorite samples.

AI in the news
http://www.aaai.org/AITopics/html/current.html

4.0   CONCLUSION

Artificial intelligence (AI) is the intelligence of machines and the branch of computer science that aims to create it. AI textbooks define the field as "the study and design of intelligent agents" where an intelligent agent is a system that perceives its environment and takes actions that maximize its chances of success. John McCarthy, who coined the term in 1956, defines it as "the science and engineering of making intelligent machines."

The field was founded on the claim that a central property of humans, intelligence—the sapience of *Homo sapiens*—can be so precisely described that it can be simulated by a machine. This raises philosophical issues about the nature of the mind and the ethics of creating artificial beings, issues which have been addressed by myth, fiction and philosophy since antiquity. Artificial intelligence has been the subject of optimism, but has also suffered setbacks and, today, has become an essential part of the technology industry, providing the heavy lifting for many of the most difficult problems in computer science.

5.0   SUMMARY

In this unit, you have learnt that:

- Artificial Intelligence is a branch of *Science* which deals with helping machines find solutions to complex problems in a more human-like fashion
- Typical AI problems
- AI History
- Limits of AI Today

6.0   TUTOR-MARKED ASSIGNMENT

1. Define intelligence.
2. What are the different approaches in defining artificial intelligence?
3. List five tasks that you will like a computer to be able to do within the next five years.
4. List five tasks that computers are unlikely to be able to do in the next five years.

7.0   REFERENCES/FURTHER READING

Dowe D.L. & Hajek, A. R. (1997). *"A Computational Extension to the Turning Test". Proceedings of the 4th Conference of the Australasian Cognitive Science Society*. http://www.csse.monash. edu.au/publications/1997/tr-cs97-322-abs.html.

Jose, H. (2000). **"Beyond the Turing Test".** *Journal of Logic, Language and Information* 9 (4): 447–466. doi:10.1023/A:1008367325700. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.44.894 3.

McCarthy, J. (November 12, 2007). "What Is Artificial Intelligence? . http://www-formal.stanford.edu/jmc/whatisai/whatisai.html

Shapiro, S. C. (1992). "Artificial Intelligence". In Shapiro, Stuart, C.. *Encyclopedia of Artificial Intelligence* (2nd ed.). New York: John Wiley. pp. 54–57. ISBN 0471503061. http://www.cse.buffalo. edu/~shapiro/Papers/ai.pdf.

Skillings, J. (2006). "Getting Machines to Think Like Us". *cnet*. http://news.cnet.com/Getting-machines-to-think-like-us/2008-11394_3-6090207.html.

Turing, Alan (1950), "Computing Machinery and Intelligence", *Mind* LIX (236): 433–460, doi:10.1093/mind/LIX.236.433, ISSN 0026-4423, http://loebner.net/Prizef/TuringArticle.html.

UNIT 2   INTRODUCTION TO INTELLIGENT AGENTS

CONTENTS

1.0   INTRODUCTION

This unit introduces you to Intelligence Agents (IA), how it interacts with the environment and Agent architectures. IA is an autonomous entity which observes and acts upon an environment . It may use knowledge to achieve their goals. They may be very simple or very complex.

2.0   OBJECTIVES

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

•   explain what an agent is and how it interacts with the environment.

- identify the percepts available to the agent and the actions that the agent can execute, if given a problem situation
- measure the performance used to evaluate an agent
- list based agents
- identify the characteristics of the environment.

3.0   MAIN CONTENT

3.1   Introduction to Agent

An agent perceives its environment through sensors. The complete set of inputs at a given time is called a percept. The current percept or a sequence of percepts can influence the actions of an agent. The agent can change the environment through actuators or effectors. An operation involving an Effector is called an action. Actions can be grouped into action sequences. The agent can have goals which it tries to achieve.

Thus, an agent can be looked upon as a system that implements a mapping from percept sequences to actions.

A performance measure has to be used in order to evaluate an agent.
An autonomous agent decides autonomously which action to take in the current situation to maximize progress towards its goals.

3.1.1 Agent Performance

An agent function implements a mapping from perception history to action. The behaviour and performance of intelligent agents have to be evaluated in terms of the agent function.

The ideal mapping specifies which actions an agent ought to take at any point in time.

The performance measure is a subjective measure to characterize how successful an agent is. The success can be measured in various ways. It can be measured in terms of speed or efficiency of the agent. It can be measured by the accuracy or the quality of the solutions achieved by the agent. It can also be measured by power usage, money, etc.

3.1.2 Examples of Agents

1.   **Humans can be looked upon as agents.** They have eyes, ears, skin, taste buds, etc. for sensors; and hands, fingers, legs, mouth for effectors.

2. **Robots are agents.** Robots may have camera, sonar, infrared, bumper, etc. for sensors. They can have grippers, wheels, lights, speakers, etc. for actuators. Some examples of robots are Xavier from CMU, COG from MIT, etc.

Xavier Robot (CMU)

Figure 2:   Xavier Robot (CMU)

Then we have the AIBO entertainment robot from SONY.

Aibo from SONY

Figure 3:   Aibo from SONY

3. We also have software agents or softbots that have some functions as sensors and some functions as actuators. Askjeeves.com is an example of a softbot.
4. Expert systems like the Cardiologist are an agent.
5. Autonomous spacecrafts.
6. Intelligent buildings.

3.1.3 Agent Faculties

The fundamental faculties of intelligence are

- Acting
- Sensing
- Understanding, reasoning, learning

Blind action is not a characterization of intelligence. In order to act intelligently, one must sense. Understanding is essential to interpret the sensory percepts and decide on an action. Many robotic agents stress sensing and acting, and do not have understanding.

3.1.4 Intelligent Agents

An Intelligent Agent must sense, must act, must be autonomous (to some extent). It also must be rational.

AI is about building rational agents. An agent is something that perceives and acts.

A rational agent always does the right thing.

1. What are the functionalities (goals)?
2. What are the components?
3. How do we build them?

3.1.5 Rationality

Perfect Rationality assumes that the rational agent knows all and will take the action that maximizes her utility. Human beings do not satisfy this definition of rationality.

Rational Action is the action that maximizes the expected value of the performance measure given the percept sequence to date.

However, a rational agent is not omniscient. It does not know the actual outcome of its actions, and it may not know certain aspects of its environment. Therefore rationality must take into account the limitations of the agent. The agent has too select the best action to the best of its knowledge depending on its percept sequence, its background knowledge and its feasible actions. An agent also has to deal with the expected outcome of the actions where the action effects are not deterministic.

3.1.6 Bounded Rationality

"Because of the limitations of the human mind, humans must use approximate methods to handle many tasks." Herbert Simon, 1972 Evolution did not give rise to optimal agents, but to agents which are in some senses locally optimal at best. In 1957, Simon proposed the notion of Bounded Rationality: that property of an agent that behaves in a manner that is nearly optimal with respect to its goals as its resources will allow.

Under these promises an intelligent agent will be expected to act optimally to the best of its abilities and its resource constraints.

3.2 Agent Environment

Environments in which agents operate can be defined in different ways. It is helpful to view the following definitions as referring to the way the environment appears from the point of view of the agent itself.

3.2.1 Observability

In terms of observability, an environment can be characterized as fully observable or partially observable.

In a fully observable environment, the entire environment relevant to the action being considered is observable. In such environments, the agent does not need to keep track of the changes in the environment. A chess playing system is an example of a system that operates in a fully observable environment.

In a partially observable environment, the relevant features of the environment are only partially observable. A bridge playing program is an example of a system operating in a partially observable environment.

3.2.2 Determinism

In deterministic environments, the next state of the environment is completely described by the current state and the agent's action. Image analysis

If an element of interference or uncertainty occurs then the environment is stochastic. Note that a deterministic yet partially observable environment will *appear* to be stochastic to the agent. Ludo

If the environment state is wholly determined by the preceding state and the actions of *multiple* agents, then the environment is said to be strategic. Example: Chess

3.2.3 Episodicity

An episodic environment means that subsequent episodes do not depend on what actions occurred in previous episodes.

In a sequential environment, the agent engages in a series of connected episodes.

3.2.4 Dynamism

Static Environment: does not change from one state to the next while the agent is considering its course of action. The only changes to the environment are those caused by the agent itself.

- A static environment does not change while the agent is thinking.
- The passage of time as an agent deliberates is irrelevant.
- The agent doesn't need to observe the world during deliberation.

A Dynamic Environment changes over time independent of the actions of the agent -- and thus if an agent does not respond in a timely manner, this counts as a choice to do nothing

3.2.5 Continuity

If the number of distinct percepts and actions is limited, the environment is discrete, otherwise it is continuous.

3.2.6 Presence of Other agents

Single agent/ Multi-agent

A multi-agent environment has other agents. If the environment contains other intelligent agents, the agent needs to be concerned about strategic, game-theoretic aspects of the environment (for either cooperative *or* competitive agents)

Most engineering environments do not have multi-agent properties, whereas most social and economic systems get their complexity from the interactions of (more or less) rational agents.

3.3   Agent architectures

3.3.1 Table Based Agent

In table based agent the action is looked up from a table based on information about the agent's percepts. A table is simple way to specify a mapping from percepts to actions. The mapping is implicitly defined by a program. The mapping may be implemented by a rule based system, by a neural network or by a procedure.

There are several disadvantages to a table based system. The tables may become very large. Learning a table may take a very long time, especially if the table is large. Such systems usually have little autonomy, as all actions are pre-determined.

3.3.2 Percept based agent or reflex agent

In percept based agents,

1.   information comes from sensors - percepts
2.   changes the agents current state of the world
3.   triggers actions through the effectors

Such agents are called reactive agents or stimulus-response agents. Reactive agents have no notion of history. The current state is as the sensors see it right now. The action is based on the current percepts only.

The following are some of the characteristics of percept-based agents.

•   Efficient
•   No internal representation for reasoning, inference.
•   No strategic planning, learning.
•   Percept-based agents are not good for multiple, opposing, goals.

3.3.3 Subsumption Architecture

We will now briefly describe the subsumption architecture (Rodney Brooks, 1986). This architecture is based on reactive systems. Brooks notes that in lower animals there is no deliberation and the actions are based on sensory inputs. But even lower animals are capable of many complex tasks. His argument is to follow the evolutionary path and build simple agents for complex worlds.

The main features of Brooks' architecture are.

- There is no explicit knowledge representation
- Behaviour is distributed, not centralized
- Response to stimuli is reflexive
- The design is bottom up, and complex behaviours are fashioned from the combination of simpler underlying ones.
- Individual agents are simple

The Subsumption Architecture built in layers. There are different layers of behaviour. The higher layers can override lower layers. Each activity is modeled by a finite state machine.

The subsumption architecture can be illustrated by Brooks' Mobile Robot example.
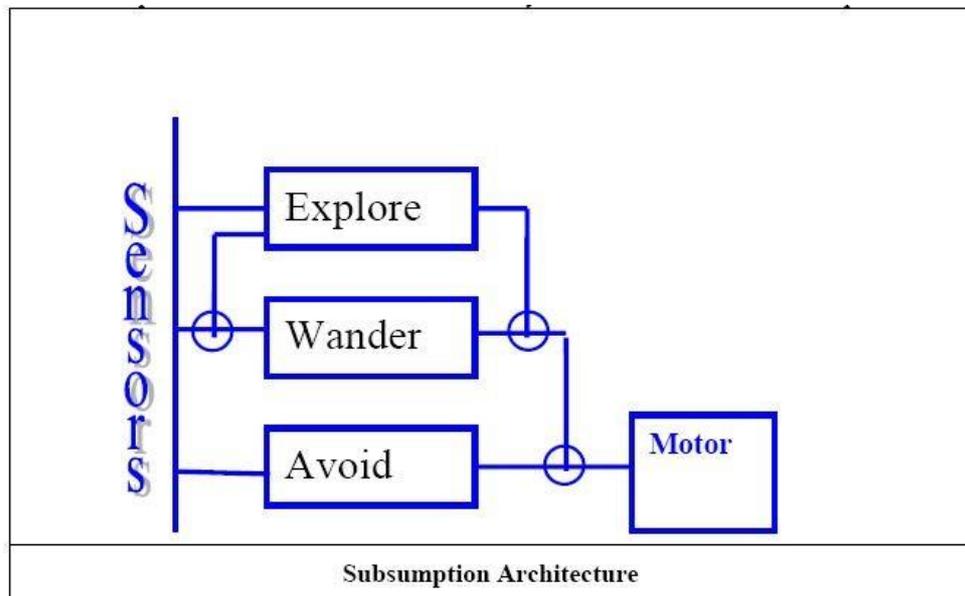
Figure 4:   Subsumption Architecture

The system is built in three layers.

1. **Layer 0:** Avoid Obstacles
2. **Layer1:** Wander behaviour
3. **Layer 2:** Exploration behavior

Layer 0 (Avoid Obstacles) has the following capabilities:

- **Sonar:** generate sonar scan
- **Collide:** send HALT message to forward
- **Feel force:** signal sent to run-away, turn

Layer1 (Wander behaviour)

- Generates a random heading
- Avoid reads repulsive force, generates new heading, feeds to turn and forward

Layer 2 (Exploration behaviour)

- Whenlook notices idle time and looks for an interesting place.
- Pathplan sends new direction to avoid.
- Integrate monitors path and sends them to the path plan.

3.3.4 State-Based Agent or Model-Based Reflex Agent

State based agents differ from percept based agents in that such agents maintain some sort of state based on the percept sequence received so far. The state is updated regularly based on what the agent senses, and the agent's actions. Keeping track of the state requires that the agent has knowledge about how the world evolves, and how the agent's actions affect the world.

Thus a state based agent works as follows:

- information comes from sensors – percepts
- based on this, the agent changes the current state of the world
- based on state of the world and knowledge (memory), it triggers actions through the effectors

3.3.5 Goal-based Agent

The goal based agent has some goal which forms a basis of its actions. Such agents work as follows:

- information comes from sensors - percepts
- changes the agents current state of the world
- based on state of the world and knowledge (memory) and goals/intentions, it chooses actions and does them through the effectors.

Goal formulation based on the current situation is a way of solving many problems and search is a universal problem solving mechanism in AI. The sequence of steps required to solve a problem is not known a priori and must be determined by a systematic exploration of the alternatives.

3.3.6 Utility-based Agent

Utility based agents provide a more general agent framework. In case that the agent has multiple goals, this framework can accommodate different preferences for the different goals.
Such systems are characterized by a utility function that maps a state or a sequence of states to a real valued utility. The agent acts so as to maximize expected utility.

3.3.7 Learning Agent

Learning allows an agent to operate in initially unknown environments. The learning element modifies the performance element. Learning is required for true autonomy

4.0   CONCLUSION

In conclusion, an intelligent agent (IA) is an autonomous entity which observes and acts upon an environment . Intelligent agents may also learn or use knowledge to achieve their goals. They may be very simple or very complex: a reflex machine such as a thermostat is an intelligent agent, as is a human being, as is a community of human beings working together towards a goal.

5.0   SUMMARY

In this unit, you have learnt that:

- AI is a truly fascinating field. It deals with exciting but hard problems. A goal of AI is to build intelligent agents that act so as to optimize performance.
- An agent perceives and acts in an environment that has architecture, and is implemented by an agent program.
- An ideal agent always chooses the action which maximizes its expected performance, given its percept sequence so far.
- An autonomous agent uses its own experience rather than built-in knowledge of the environment by the designer.
- An agent program maps from percept to action and updates its internal state.
- Reflex agents respond immediately to percepts.
- Goal-based agents act in order to achieve their goal(s).
- Utility-based agents maximize their own utility function.
- Representing knowledge is important for successful agent design.

- The most challenging environments are partially observable, stochastic, sequential, dynamic, and continuous, and contain multiple intelligent agents.

6.0   TUTOR-MARKED ASSIGNMENT

1. Define an agent.
2. What is a rational agent?
3. What is bounded rationality?
4. What is an autonomous agent?
5. Describe the salient features of an agent.

7.0   REFERENCES/FURTHER READING

Bowling, M. & Veloso, M. (2002). Multiagent Learning Using a Variable Learning Rate. *Artificial Intelligence*. 136(2): 215-250.

Serenko, A.; Detlor, B. (2004). "Intelligent Agents as Innovations". *AI and Society* 18 (4): 364–381. doi:10.1007/s00146-004-0310-5. http://foba.lakeheadu.ca/serenko/papers/Serenko_Detlor_AI_and _Society.pdf

Serenko, A.; Ruhi, U.; Cocosila, M. (2007). "Unplanned Effects of Intelligent Agents on Internet use: Social Informatics approach". *AI and Society* 21 (1–2): 141–166. doi:10.1007/s00146-006-0051-8. http://foba.lakeheadu.ca/serenko/papers/AI_Society_ Serenko_Social_Impacts_of_Agents.pdf.

**MODULE 2   SEARCH IN ARTIFICIAL
                 INTELLIGENCE**

UNIT 1   INTRODUCTION TO STATE SPACE SEARCH

CONTENTS

1.0   INTRODUCTION

In computer science, a search algorithm, broadly speaking, is an algorithm for finding an item with specified properties among a collection of items. The items may be stored individually as records in a database; or may be elements of a search space defined by a mathematical formula or procedure, such as the roots of an equation with integer variables; or a combination of the two, such as the Hamiltonian circuits of a graph.

Specifically, Searching falls under Artificial Intelligence (AI). A major goal of AI is to give computers the ability to think, or in other words, mimic human behavior. The problem is, unfortunately, computers don't

function in the same way our minds do. They require a series of *well-reasoned out* steps before finding a solution. Your goal, then, is to take a complicated task and convert it into simpler steps that your computer can handle. That conversion from something complex to something simple is what this unit is primarily about. Learning how to use two search algorithms is just a welcome side-effect. This unit will explain the background for AI search and some of the AI search techniques.

2.0   OBJECTIVES

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

- describe the state space representation
- describe some algorithms
- formulate, when given a problem description, the terms of a state space search problem
- analyse the properties of some algorithms
- analyse a given problem and identify the most suitable search strategy for the problem
- solve some simple problems.

3.0   MAIN CONTENT

3.1   State Space Search

Let us begin by introducing certain terms.

An <u>initial state</u> is the description of the starting configuration of the agent.

An <u>action</u> or an <u>operator</u> takes the agent from one state to another state which is called a successor state. A state can have a number of successor states.

A <u>plan</u> is a sequence of actions. The cost of a plan is referred to as the <u>path cost</u>. The path cost is a positive number, and a common path cost may be the sum of the costs of the steps in the path. The goal state is the partial description of the solution

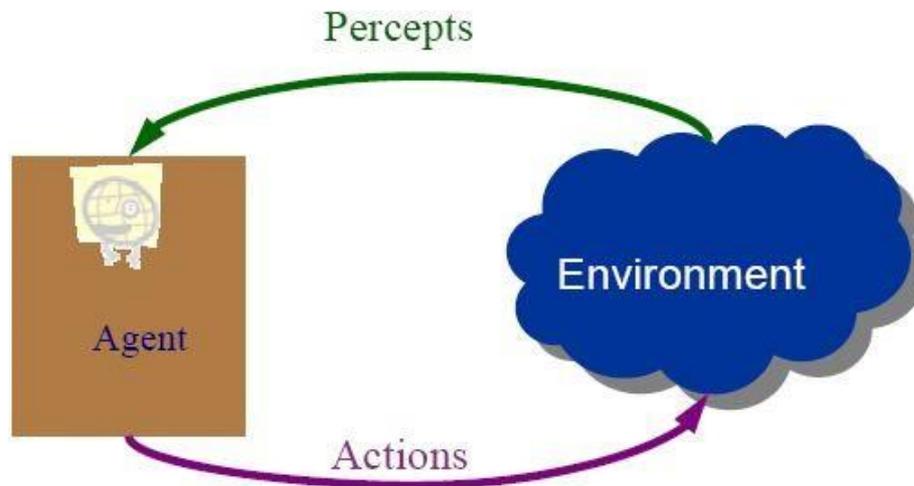### 3.1.1 Goal Directed Agent

Figure 1:   Goal Directed Agent

We have earlier discussed about an intelligent agent. In this unit we will study a type of intelligent agent which we will call a goal directed agent. A goal directed agent needs to achieve certain goals. Such an agent selects its actions based on the goal it has. Many problems can be represented as a set of states and a set of rules of how one state is transformed to another. Each state is an abstract representation of the agent's environment. It is an abstraction that denotes a configuration of the agent.

Let us look at a few examples of goal directed agents.

1.   15-puzzle: The goal of an agent working on a 15-puzzle problem may be to reach a configuration which satisfies the condition that the top row has the tiles 1, 2 and 3. The details of this problem will be described later.

2.   The goal of an agent may be to navigate a maze and reach the HOME position.

The agent must choose a sequence of actions to achieve the desired goal.

3.1.2 State Space Search Notations

Now let us look at the concept of a search problem.

Problem formulation means choosing a relevant **set of states** to consider, and a feasible **set of operators** for moving from one state to another.

Search is the process of considering various possible sequences of operators applied to the initial state, and finding out a sequence which culminates in a goal state.

3.2   Problem Space

What is problem space?

A problem space is a set of states and a set of operators. The operators map from one state to another state. There will be one or more states that can be called initial states, one or more states which we need to reach what are known as goal states and there will be states in between initial states and goal states known as intermediate states. So what is the solution? The solution to the given problem is nothing but a sequence of operators that map an initial state to a goal state. This sequence forms a solution path. What is the best solution? Obviously the shortest path from the initial state to the goal state is the best one. Shortest path has only a few operations compared to all other possible solution paths. Solution path forms a tree structure where each node is a state. So searching is nothing but exploring the tree from the root node.

3.2.1 Search Problem

We are now ready to formally describe a search problem.

A search problem consists of the following:

- S: the full set of states

- s_0 : the initial state

- A:S→S is a set of operators

- G is the set of final states. Note that G ⊆S

These are schematically depicted in Figure 2.
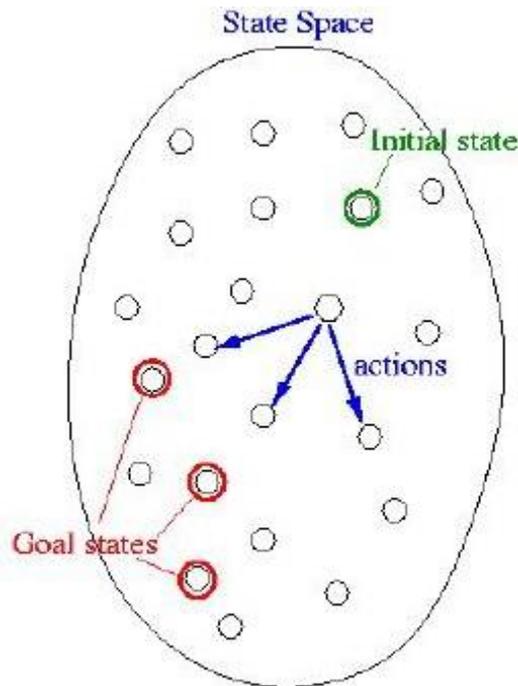
Figure 2

The search problem is to find a sequence of actions which transforms the agent from the initial state to a goal state g∈G. A search problem is represented by a 4-tuple $\{S, s_0, A, G\}$.

S: set of states
$s_0 \in$ S: initial state
A: S? S operators/ actions that transform one state to another state
G: goal, a set of states. $G \subseteq S$

This sequence of actions is called a solution plan. It is a path from the initial state to a goal state. A *plan* P is a sequence of actions.

$P = \{a_0, a_1, a_N\}$ which leads to traversing a number of states $\{s_0, s_1, Sn+_1 \in G\}$. A sequence of states is called a path. The cost of a path is a positive number. In many cases the path cost is computed by taking the sum of the costs of each action.

Representation of search problems
A search problem is represented using a directed graph.

- The states are represented as nodes.
- The allowed actions are represented as arcs.

28

Searching process

The generic searching process can be very simply described in terms of the following steps:

Do until a solution is found or the state space is exhausted.

1. Check the current state

2. Execute allowable actions to find the successor states.

3. Pick one of the new states.

4. Check if the new state is a solution state

If it is not, the new state becomes the current state and the process is repeated

3.3   Examples

3.3.1 Illustration of a search process

We will now illustrate the searching process with the help of an example. Consider the problem depicted in Figure 3.
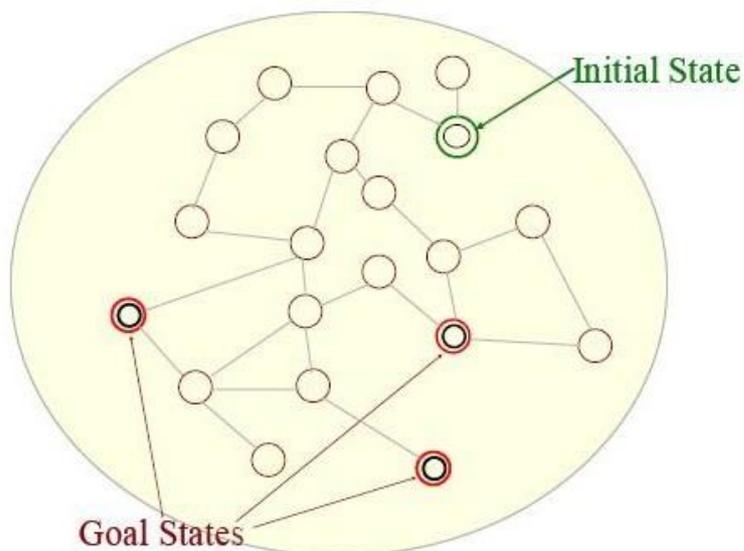
Figure 3

s_0 is the initial state.
The successor states are the adjacent states in the graph.

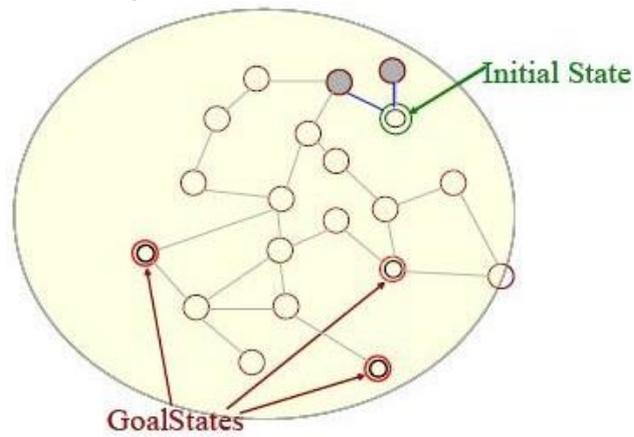There are three goal states.

Figure 4

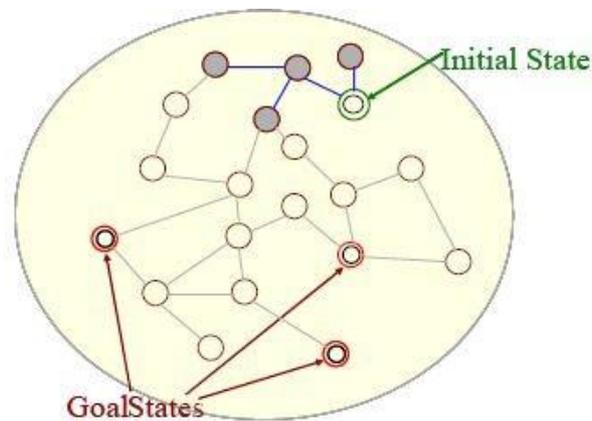The two successor states of the initial state are generated.

Figure 5

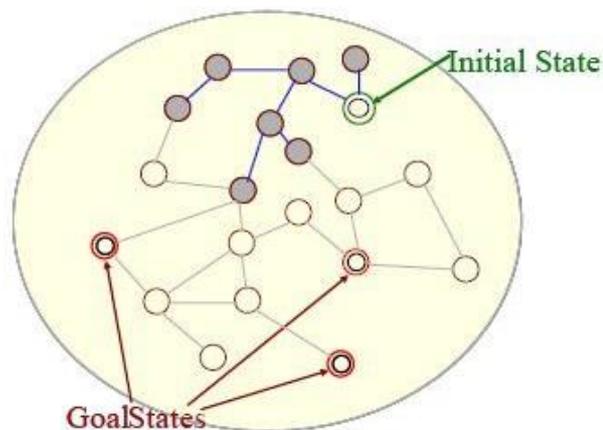The successors of these states are picked and their successors are generated.

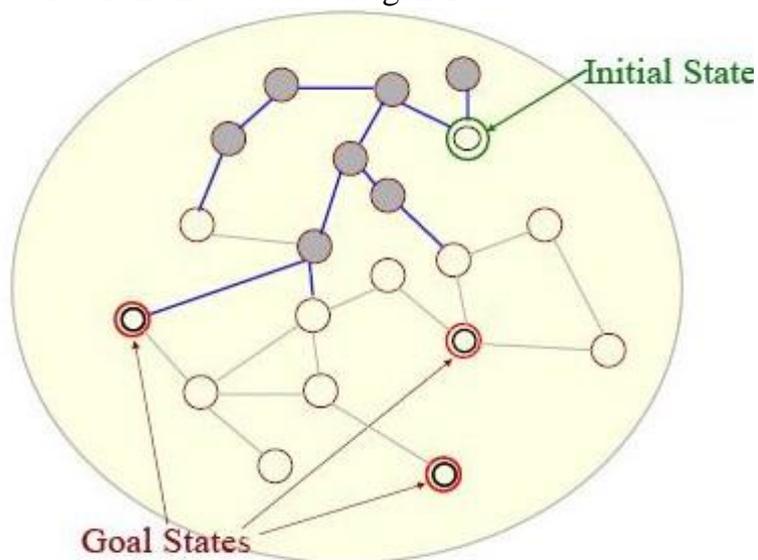Figure 6

Successors of all these states are generated.

Figure 7

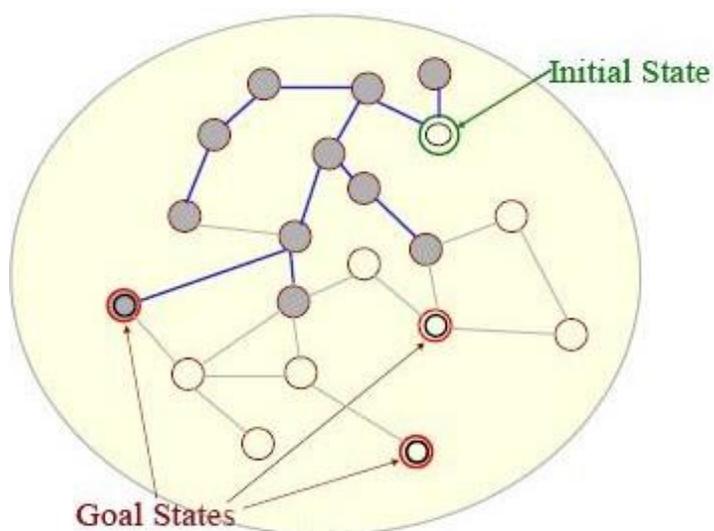The successors are generated.

Figure 8

A goal state has been found.

The above example illustrates how we can start from a given state and follow the successors, and be able to find solution paths that lead to a goal state. The grey nodes define the search tree. Usually the search tree is extended one node at a time. The order in which the search tree is extended depends on the search strategy.

We will now illustrate state space search with one more example – the pegs and disks problem. We will illustrate a solution sequence which when applied to the initial state takes us to a goal state.

3.3.2 Example problem: Pegs and Disks problem

Consider the following problem. We have 3 pegs and 3 disks.

Operators: one may move the topmost disk on any needle to the topmost position to any other needle.

In the goal state all the pegs are in the needle B as shown in the figure below.
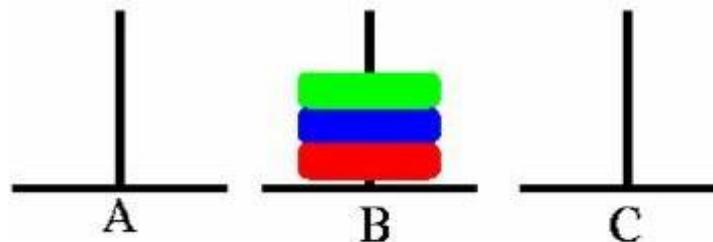

Figure 9

The initial state is illustrated below.
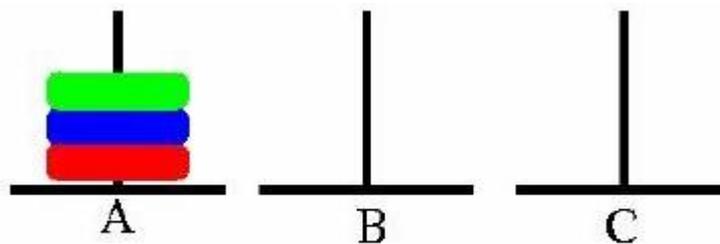

Figure 10

Now we will describe a sequence of actions that can be applied on the initial state.

Step 1: Move A → C
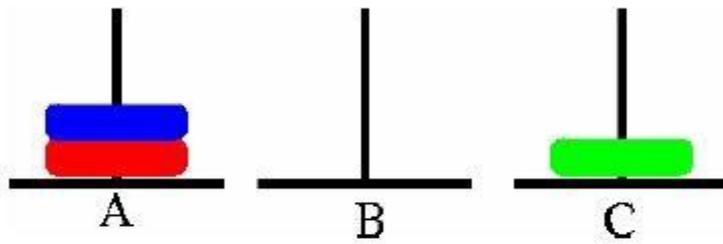
Figure 11

Step 2: Move A → B
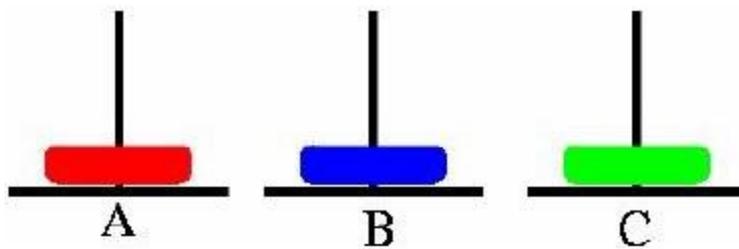
Figure 12

Step 3: Move A → C
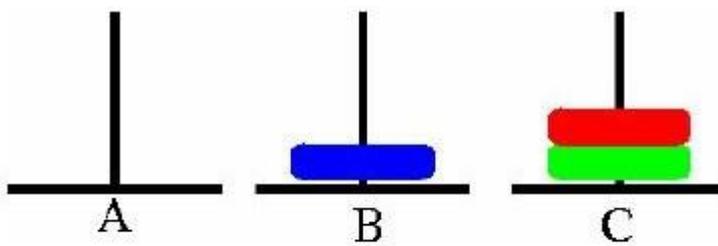
Figure 13

Step 4: Move B→ A
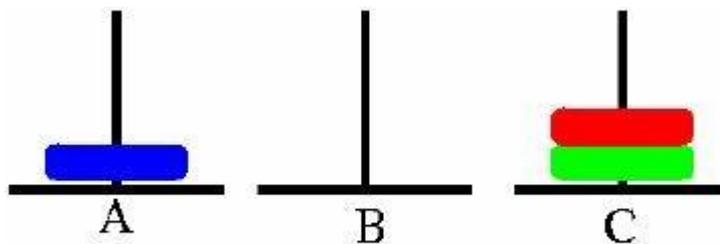
Figure 14

Step 5: Move C → B
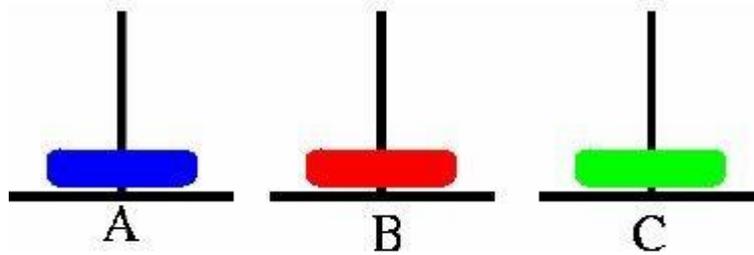
Figure 15

Step 6: Move A → B
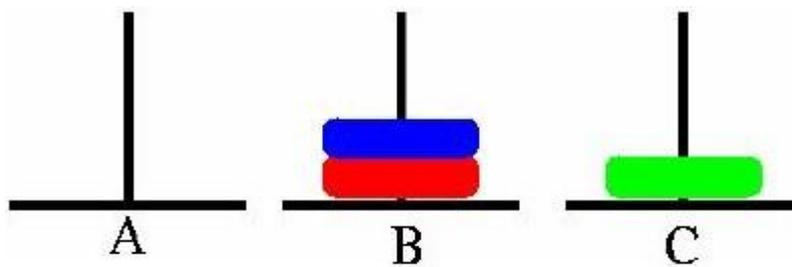
Figure 16

Step 7: Move C→ B
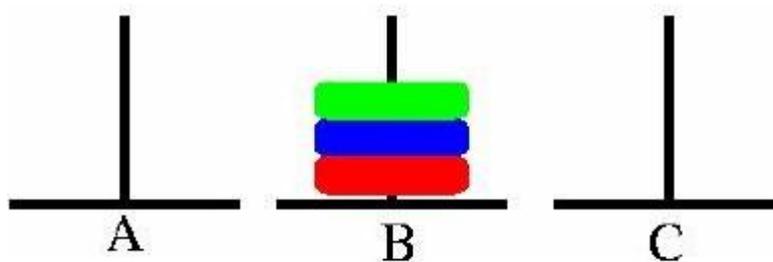
Figure 17

We will now look at another search problem – the 8-queens problem, which can be generalized to the N-queens problem.

3.3.3 Queens Problem

The problem is to place 8 queens on a chessboard so that no two queens are in the same row, column or diagonal.

The picture below on the left shows a solution of the 8-queens problem. The picture on the right is not a correct solution, because some of the queens are attacking each other.
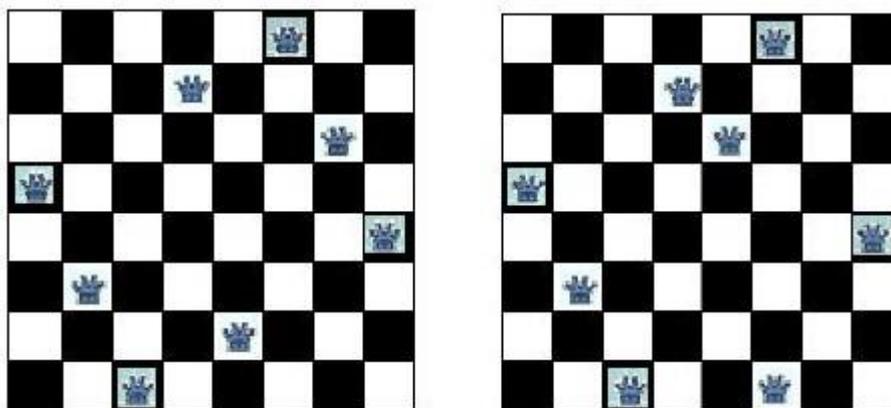
Figure 18:   Queens Problem

How do we formulate this in terms of a state space search problem? The problem formulation involves deciding the representation of the states, selecting the initial state representation, the description of the operators, and the successor states. We will now show that we can formulate the search problem in several different ways for this problem.

N queens problem formulation 1

- **States:** Any arrangement of 0 to 8 queens on the board
- **Initial state:** 0 queens on the board
- **Successor function:** Add a queen in any square
- **Goal test:** 8 queens on the board, none are attacked

The initial state has 64 successors. Each of the states at the next level has 63 successors, and so on. We can restrict the search tree somewhat by considering only those successors where no queen is attacking each other. To do that, we have to check the new queen against all existing queens on the board. The solutions are found at a depth of 8.
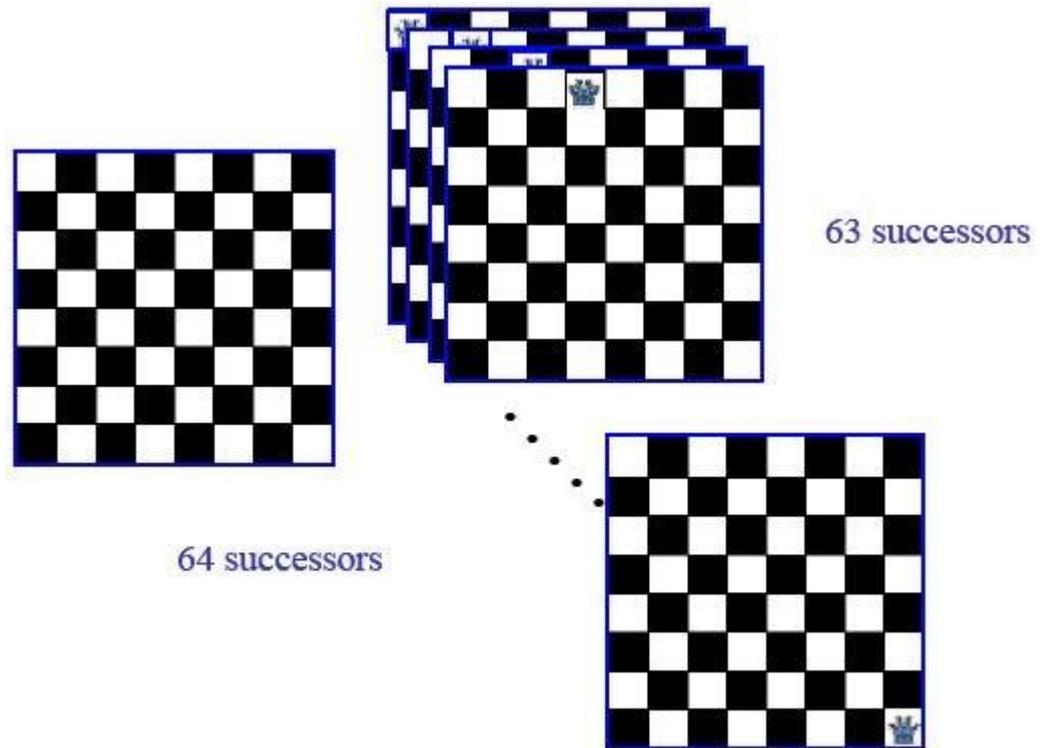
Figure 19

N queens problem formulation 2

- **States:** Any arrangement of 8 queens on the board
- **Initial state:** All queens are at column 1
- **Successor function:** Change the position of any one queen
- **Goal test:** 8 queens on the board, none are attacked
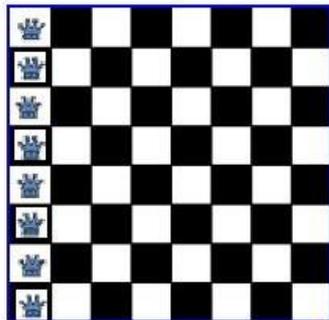
Figure 20

If we consider moving the queen at column 1, it may move to any of the seven remaining columns.

N queens problem formulation 3

- **States:** Any arrangement of k queens in the first k rows such that none are attacked
- **Initial state:** 0 queens on the board
- **Successor function:** Add a queen to the (k+1) th row so that none are attacked.
- **Goal test :** 8 queens on the board, none are attacked
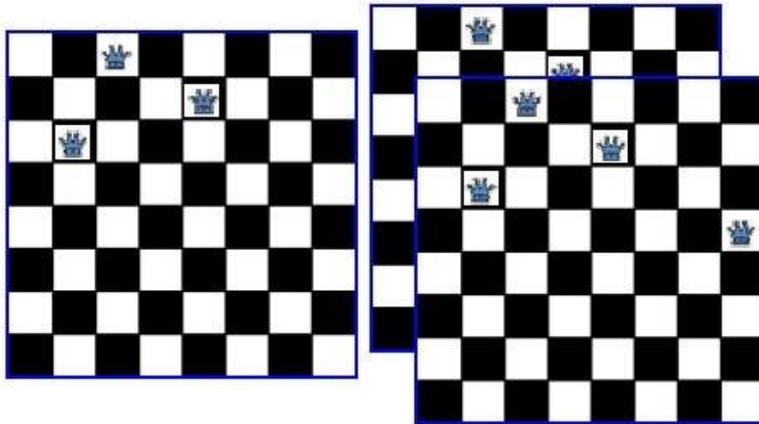
Figure 21

We will now take up yet another search problem, the 8 puzzle.

3.3.4 Problem Definition - Example, 8 puzzle

Figure 22

In the 8-puzzle problem we have a 3×3 square board and 8 numbered tiles. The board has one blank position. Bocks can be slid to adjacent blank positions. We can alternatively and equivalently look upon this as the movement of the blank position up, down, left or right. The objective of this puzzle is to move the tiles starting from an initial position and arrive at a given goal configuration.

The 15-puzzle problems is similar to the 8-puzzle. It has a 4×4 square board and 15 numbered tiles

The state space representation for this problem is summarized below:
States: A state is a description of each of the eight tiles in each location that it can occupy.

Operators/Action: The blank moves left, right, up or down

Goal Test: The current state matches a certain state (e.g. one of the ones shown on previous slide)

Path Cost: Each move of the blank costs 1

A small portion of the state space of 8-puzzle is shown below. Note that we do not need to generate all the states before the search begins. The states can be generated when required.
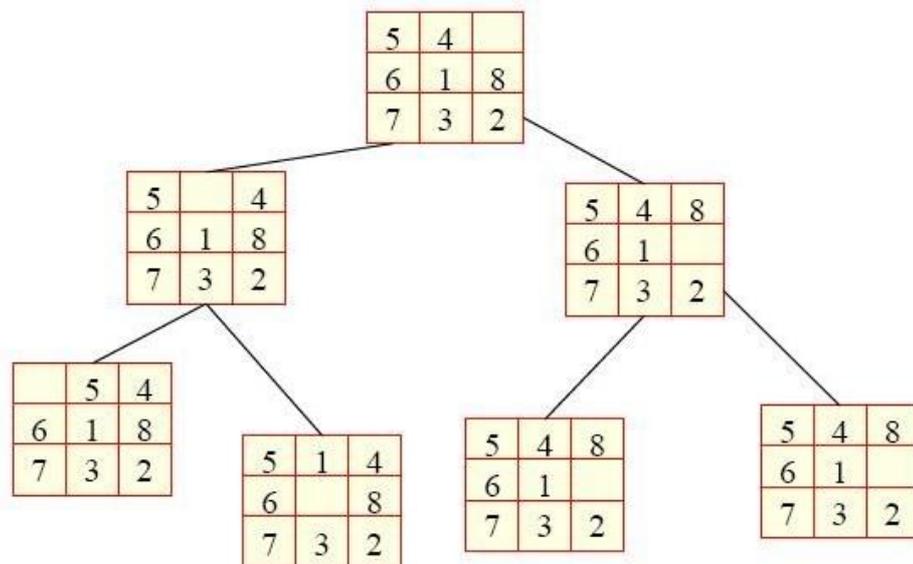
Figure 23

8-puzzle partial state space

3.4   Types of AI Search Techniques

Solution can be found with less information or with more information. It all depends on the problem we need to solve. Usually when we have more information it will be easy to solve the problem. The following are the types of AI search namely: Uninformed Search, List search, Tree search , Graph search, SQL search, Tradeoff Based search, Informed search, Adversarial search. This module will only deal with uninformed search, informed search and Tree search.

4.0   CONCLUSION

State space search is a process used in the field of computer science, including artificial intelligence (AI), in which successive configurations or *states* of an instance are considered, with the goal of finding a *goal state* with a desired property.

Problems are often modelled as a state space, a set of *states* that a problem can be in. The set of states forms a graph where two states are connected if there is an *operation* that can be performed to transform the first state into the second.

State space search often differs from traditional computer science search methods because the state space is *implicit*: the typical state space graph is much too large to generate and store in memory. Instead, nodes are generated as they are explored, and typically discarded thereafter. A solution to a combinatorial search instance may consist of the goal state itself, or of a path from some *initial state* to the goal state.

5.0   SUMMARY

In this unit, you have learnt that:

- State space search is a process used in the field of computer science, including artificial intelligence (AI), in which successive configurations or *states* of an instance are considered, with the goal of finding a *goal state* with a desired property
- The search problem_is to find a sequence of actions which transforms the agent from the initial state to a goal state $g \in G$. A search problem is represented by a 4-tuple $\{S, s_0, A, G\}$.

- Solution can be found with less information or with more information. It all depends on the problem we need to solve

6.0   TUTOR-MARKED ASSIGNMENT

1.   Find a path from a boxed node to the goal node (p).

State Space Graph
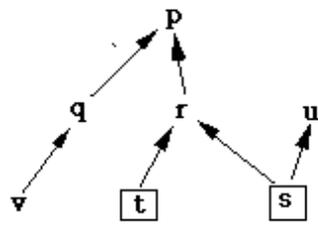
Assertions

q => p
r => p
v => q
s => r
t => r
s => u
s
t

The path [s r p] corresponds to: s and s => r yields r
                                    r and r => p yields p

Data Driven Proof of p: Find a path from a boxed node
(start node) to the goal node (p).

2.   Using the following AND/OR graph, where is fred?

location(X,Z)

gooddog(X)   master(X,Y)   location(Y,Z)

collie(X)   trained(X)   master(fred,sam)   location(sam,museum)

collie(fred)   trained(fred)   day(sat)   ¯warm(sat))
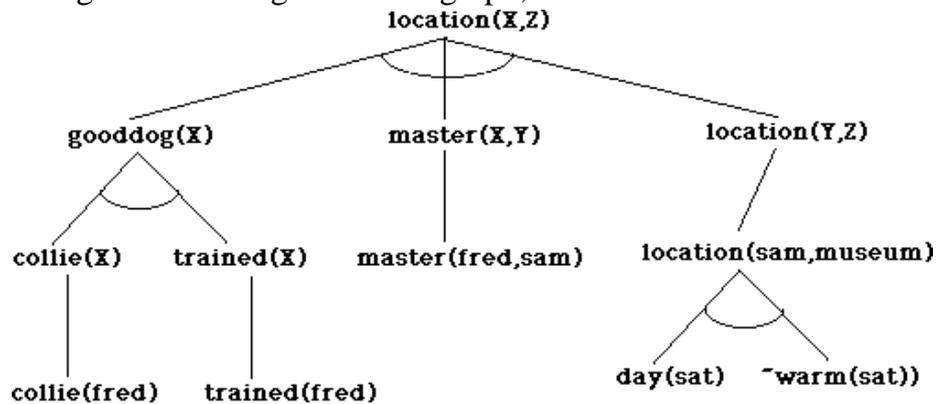
Goal-Driven Search: Where is fred?
Substitutions: {fred/X, sam/Y, museum/Z}

7.0   REFERENCES/FURTHER READING

Dechter, R. & Judea, P. (1985). "Generalized Best-First Search
        Strategies and the Optimality of A*". *Journal of the ACM* 32 (3):
        505–536. doi:10.1145/3828.3830.

Koenig, S.; Maxim, L.; Yaxin, L.; David, F. (2004). "Incremental
        Heuristic Search in AI". *AI Magazine* 25 (2): 99–112.
        http://portal.acm.org/citation.cfm?id=1017140.

Nilsson, N. J. (1980). *Principles of Artificial Intelligence*. Palo Alto,
        California: Tioga Publishing Company. ISBN 0-935382-01-1.

Pearl, J. (1984). *Heuristics: Intelligent Search Strategies for Computer
        Problem Solving*. Addison-Wesley Longman Publishing Co., Inc.
        ISBN 0-201-05594-5.

Russell, S. J. & Norvig, P. (2003). *Artificial Intelligence. A Modern Approach*. Upper Saddle River, N.J.: Prentice Hall. pp. 97–104. ISBN 0-13-790395-2.

**UNIT 2   UNINFORMED SEARCH OR BRUTE FORCE
            SEARCH**

CONTENTS

1.0   INTRODUCTION

In computer science, uninform-cost search (UCS) is a tree search
algorithm used for traversing or searching a weighted tree, tree structure,
or graph. The search begins at the root node. The search continues by
visiting the next node which has the least total cost from the root. Nodes
are visited in this manner until a goal state is reached.

Typically, the search algorithm involves expanding nodes by adding all
unexpanded neighbouring nodes that are connected by directed paths to
a priority queue. In the queue, each node is associated with its total path
cost from the root, where the least-cost paths are given highest priority.
The node at the head of the queue is subsequently expanded, adding the
next set of connected nodes with the total path cost from the root to the
respective node. The uniform-cost search is complete and optimal if the
cost of each step exceeds some positive bound ε.

2.0   OBJECTIVES

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

* explain uninformed search
* list two types of uninformed search
* describe depth first and breadth first search
* solve simple problems on uninformed search.

3.0   MAIN CONTENT

3.1   Uninformed Search

Sometimes we may not get much relevant information to solve a problem. Suppose we lost our car key and we are not able to recall where we left, we have to search for the key with some information such as in which places we used to place it. It may be our pant pocket or may be the table drawer. If it is not there then we have to search the whole house to get it. The best solution would be to search in the places from the table to the wardrobe. Here we need to search blindly with fewer clues. This type of search is called uninformed search or blind search. There are two popular AI search techniques in this category: breadth first search and depth first search.

3.2   Depth First and Breadth First Search

If you want to go from Point A to Point B, you are employing some kind of search. For a direction finder, going from Point A to Point B literally means finding a path between where you are now and your intended destination. For a chess game, Point A to Point B might be two points between its current position and its position 5 moves from now. For a genome sequence, Points A and B could be a link between two DNA sequences.

As you can tell, going from Point A to Point B is different for every situation. If there is a vast amount of interconnected data, and you are trying to find a relation between few such pieces of data, you would use search. In this unit, you will learn about two forms of searching, depth first and breadth first.

Our Search Representation

Lets you learn how we humans could solve a search problem. First, we need a representation of how our search problem will exist. The following is an example of our search tree. It is a series of interconnected nodes that we will be searching through:
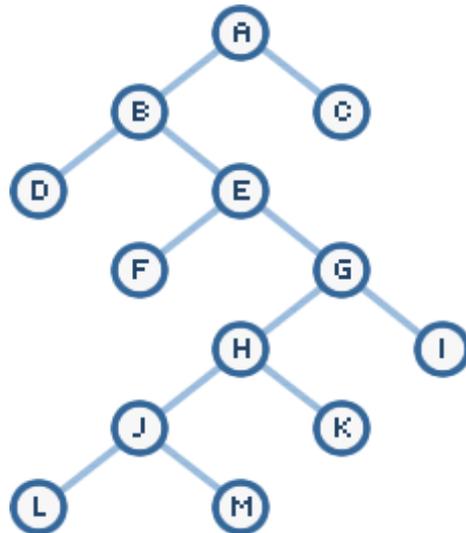
In our above graph, the path connections are not two-way. All paths go only from top to bottom. In other words, A has a path to B and C, but B and C do not have a path to A. It is basically like a one-way street.

Each lettered circle in our graph is a node. A node can be connected to other via our edge/path, and those nodes that are connected to be called neighbors. B and C are neighbors of A. E and D are neighbors of B, and B is not a neighbor of D or E because B cannot be reached using either D or E.

Our search graph also contains depth:
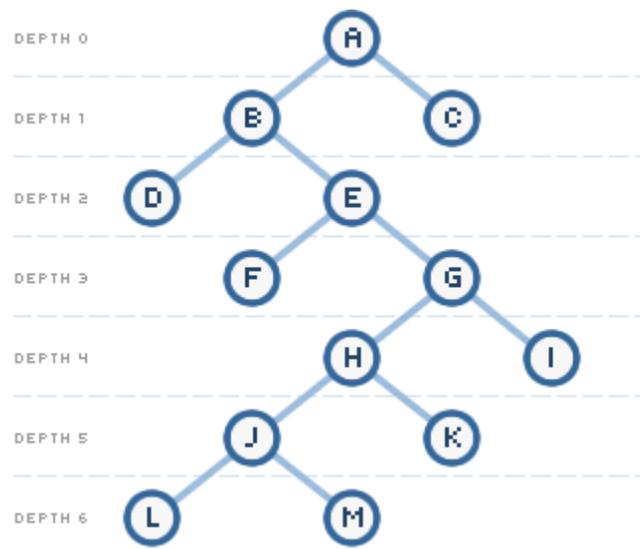
We now have a way of describing location in our graph. We know how the various nodes (the lettered circles) are related to each other

(neighbors), and we have a way of characterizing the depth each belongs in. Knowing this information isn't directly relevant in creating our search algorithm, but they do help us to better understand the problem.

3.2.1 Depth First Search

Depth first search works by taking a node, checking its neighbors, expanding the first node it finds among the neighbors, checking if that expanded node is our destination, and if not, continue exploring more nodes.

The above explanation is probably confusing if this is your first exposure to depth first search. I hope the following demonstration will help you more. Using our same search tree, let's find a path between nodes A and F:
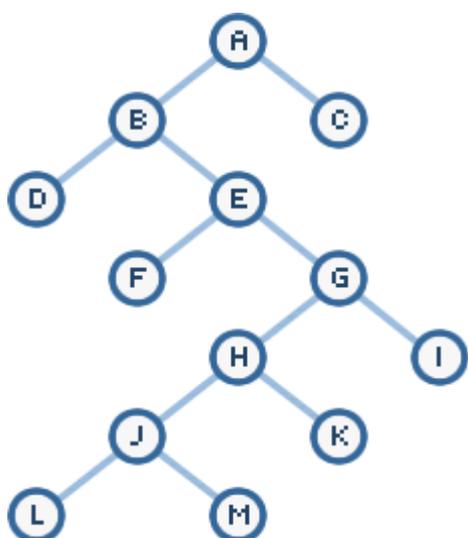
Step 0

let's start with our root/goal node:

I will be using two lists to keep track of what we are doing - an Open list and a Closed List. An Open list keeps track of what you need to do, and the Closed List keeps track of what you have already done. Right now, we only have our starting point, node A. We haven't done anything to it yet, so let's add it to our Open list.

- Open List: A
- Closed List: <empty>

Step   1

Now, let's explore the neighbors of our A node. To put another way, let's take the first item from our Open list and explore its neighbors:

Node A's neighbors are the B and C nodes. Because we are now done with our A node, we can remove it from our Open list and add it to our Closed List. You aren't done with this step though. You now have two new nodes B and C that need exploring. Add those two nodes to our Open list.

Our current Open and Closed Lists contain the following data:

- Open List: B, C
- Closed List: A

Step 2

Our Open list contains two items. For depth first search and breadth first search, you always explore the first item from our Open list. The first item in our Open list is the B node. B is not our destination, so let's explore its neighbors:
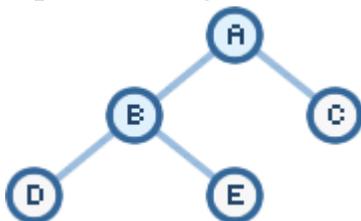
Because I have now expanded B, I am going to remove it from the Open list and add it to the Closed List. Our new nodes are D and E, and we add these nodes to the *beginning* of our Open list:

- Open List: D, E, C
- Closed List: A, B

Step 3

You should start to see a pattern forming. Because D is at the beginning of our Open List, we expand it. D isn't our destination, and it does not contain any neighbors. All you do in this step is remove D from our Open List and add it to our Closed List:

- Open List: E, C
- Closed List: A, B, D

Step 4

We now expand the E node from our Open list. E is not our destination, so we explore its neighbors and find out that it contains the neighbors F and G. Remember, F is our target, but we don't stop here though. Despite F being on our path, we only end when we are about to *expand* our target Node - F in this case:
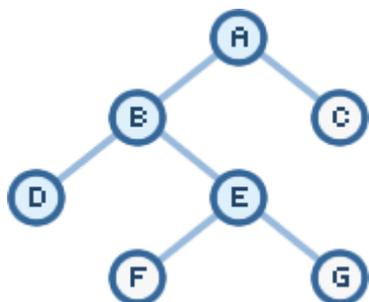
Our Open list will have the E node removed and the F and G nodes added. The removed E node will be added to our Closed List:

- Open List: F, G, C
- Closed List: A, B, D, E

Step 5

We now expand the F node. Since it is our intended destination, we stop:
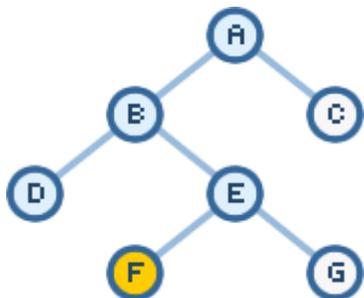
We remove F from our Open list and add it to our Closed List. Since we are at our destination, there is no need to expand F in order to find its neighbors. Our final Open and Closed Lists contain the following data:

- Open List: G, C
- Closed List: A, B, D, E, F

The final path taken by our depth first search method is what the final value of our Closed List is: A, B, D, E, F. Towards the end of this tutorial, I will analyze these results in greater detail so that you have a better understanding of this search method.

3.2.2 Breadth First Search

The reason I cover both depth and breadth first search methods in the same unit is because they are both similar. In depth first search, newly explored nodes were added to the beginning of your Open list. In breadth first search, newly explored nodes are added to the end of your Open list.

Let's see how that change will affect our results. For reference, here is our original search tree:
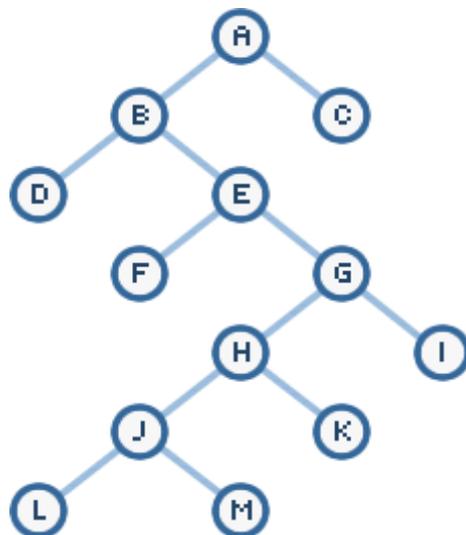
Let's try to find a path between nodes A and E.

Step 0

let's start with our root/goal node:

Like before, I will continue to employ the Open and Closed Lists to keep track of what needs to be done:

- Open List: A
- Closed List: <empty>

Step 1

Now, let's explore the neighbours of our A node. So far, we are following in depth first's footsteps:

We remove A from our Open list and add A to our Closed List. A's neighbours, the B and C nodes, are added to our Open list. They are added to the end of our Open list, but since our Open list was empty (after removing A), it's hard to show that in this step.

Our current Open and Closed Lists contain the following data:

- Open List: B, C
- Closed List: A

Step 2

Here is where things start to diverge from our depth first search method. We take a look the B node because it appears first in our Open List. Because B isn't our intended destination, we explore its neighbours:
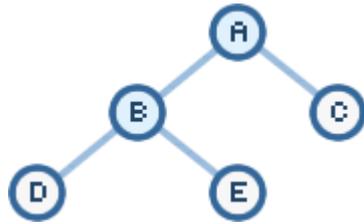
B is now moved to our Closed List, but the neighbours of B, nodes D and E are added to the *end* of our Open list:

- Open List: C, D, E
- Closed List: A, B

Step 3

We now expand our C node:
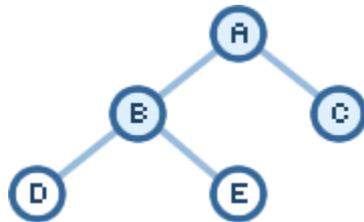
Since C has no neighbours, all we do is remove C from our Closed List and move on:

- Open List: D, E
- Closed List: A, B, C

Step 4

Similar to Step 3, we expand node D. Since it isn't our destination, and it too does not have any neighbours, we simply remove D from our to Open list, add D to our Closed List, and continue on:

- Open List: E
- Closed List: A, B, C, D

Step 5

Because our Open list only has one item, we have no choice but to take a look at node E. Since node E is our destination, we can stop here:
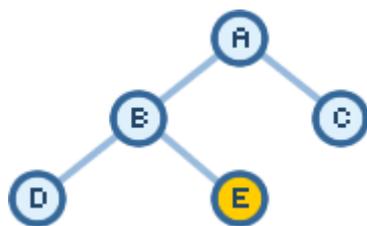
Our final versions of the Open and Closed Lists contain the following data:

- Open List: <empty>
- Closed List: A, B, C, D, E

Traveling from A to E takes you through B, C, and D using breadth first search

4.0   CONCLUSION

1.   Uninformed search strategies -Also known as "blind search," uninformed search strategies use no information about the likely "direction" of the goal node(s).
2.   Uninformed search major methods are Breadth-first and depth-first

5.0   SUMMARY

In this unit, you learnt that:

- Uninformed strategies use only the information available in the problem definition.
- Some such strategies considered :

-    Breadth-first search
-    Tree Search
-    Depth-first search

6.0   TUTOR -MARKED ASSIGNMENT

Water Jug Problem

i.   Given a 5-gallon jug and a 2-gallon jug, with the 5-gallon jug initially full of water and the 2-gallon jug empty, the goal is to fill the 2-gallon jug with exactly one gallon of water.
ii.   8-Puzzle
       Given an initial configuration of eight numbered tiles on a 3 x 3

board, move the tiles in such a way so as to produce a desired goal configuration of the tiles.

iii.   Missionaries and Cannibals
There are three missionaries, three cannibals, and 1 boat that can carry up to two people on one side of a river. Goal: Move all the missionaries and cannibals across the river.

iv.   Remove five Sticks
Given the following configuration of sticks; remove exactly five sticks in such a way that the remaining configuration forms exactly three squares.

7.0   REFERENCES/FURTHER READING

Christopher, D. M. & Hinrich, S., *Foundations of Statistical Natural Language Processing.* The MIT Press.

Wooldridge, M. *An Introduction to Multiagent Systems*. John Wiley & Sons, Ltd.

Russell, S. J. & Norvig, P. (2003). *Artificial Intelligence: A Modern Approach* (2nd ed.). Upper Saddle River, New Jersey: Prentice Hall, ISBN 0-13-790395-2, http://aima.cs.berkeley.edu/

Russell, S. & Norvig, P. (2010). *Artificial Intelligence: A Modern Approach* (3 ed.). Prentice Hall. ISBN 978-0-13-6042594.

UNIT 3   INFORMED   SEARCH   OR   HEURISTIC SEARCH

CONTENTS

1.0   INTRODUCTION

We have seen that uninformed search methods that systematically explore the state space and find the goal. They are inefficient in most cases. Informed search methods use problem specific knowledge, and may be more efficient. Informed Search will be able to unravel the factoring an effective way if we now have relevant information, clues or hints. The clues that assist solve the factor constitute heuristic information. Informed search could also be known as heuristic search.

According to George Polya *heuristic* is the study of the methods and rules of discovery and invention. In state space search, *heuristic* define the rules for choosing branches in a state space that are most likely to lead to an acceptable solution. There are two cases in AI searches when heuristics are needed:

- The problem has no exact solution. For example, in medical diagnosis doctors use heuristic to choose the most likely diagnoses given a set of symptoms.
- The problem has an exact solution but is too complex to allow for a *brute force* solution.

Key Point: Heuristics are fallible. Because they rely on limited information, they may lead to a suboptimal solution or to a dead end.

2.0   OBJECTIVES

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

- explain informed search
- mention other names of informed search
- describe best-first search
- describe greedy search
- solve simple problems on informed search.

3.0   MAIN CONTENT

3.1   What is Heuristic?

Heuristic search methods explore the search space "intelligently". That is, evaluating possibilities without having to investigate every single possibility.

Heuristic search is an AI search technique that employs heuristic for its moves. *Heuristic* is a rule of thumb that probably leads to a solution. Heuristic play a major role in search strategies because of exponential nature of the most problems. Heuristics help to reduce the number of alternatives from an exponential number to a polynomial number. In Artificial Intelligence, heuristic search has a general meaning, and a more specialized technical meaning. In a general sense, the term heuristic is used for any advice that is often effective, but is not guaranteed to work in every case.

Heuristic means "rule of thumb". To quote Judea Pearl, "Heuristics are criteria, methods or principles for deciding which among several alternative courses of action promises to be the most effective in order to achieve some goal". In heuristic search or informed search, heuristics are used to identify the most promising search path.

3.1.1 Examples of Heuristic Function

A heuristic function at a node n is an estimate of the optimum cost from the current node to a goal. It is denoted by *h (n)*.
H (n) = estimated cost of the cheapest path from node n to a goal node

Example 1: We want a path from Kolkata to Guwahati Heuristic for Guwahati may be straight-line distance between Kolkata and Guwahati
h(Kolkata) = euclideanDistance(Kolkata, Guwahati)

Example 2: 8-puzzle: Misplaced Tiles Heuristics is the number of tiles out of place.

2	8	3
1	6	4
	7	5

Initial State

1	2	3
8		4
7	6	5
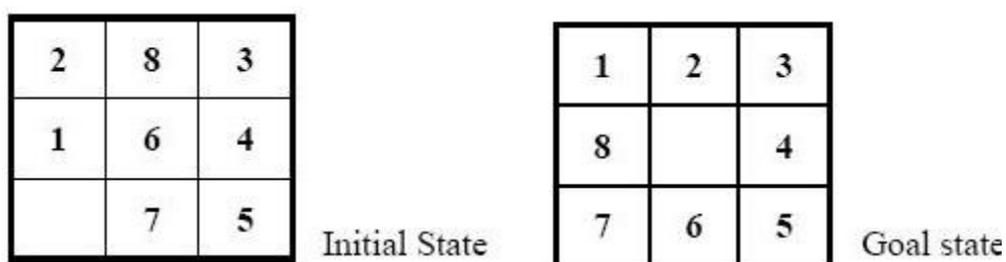
Goal state

Figure 1: 8 puzzle

The first picture shows the current state *n*, and the second picture the goal state.

h(n) = 5 because the tiles 2, 8, 1, 6 and 7 are out of place.

Manhattan Distance Heuristic: Another heuristic for 8-puzzle is the Manhattan distance heuristic. This heuristic sums the distance that the tiles are out of place. The distance of a tile is measured by the sum of the differences in the x-positions and the y-positions.

For the above example, using the Manhattan distance heuristic,
h(n) = 1 + 1 + 0 + 0 + 0 + 1 + 1 + 2 = 6

We will now study a heuristic search algorithm best-first search.

3.2   Best-First Search

Best-first search is a search algorithm which explores a graph by expanding the most promising node chosen according to a specified rule.

Judea Pearl described best-first search as estimating the promise of node *n* by a "heuristic evaluation function *f(n)* which, in general, may depend on the description of *n*, the description of the goal, the information gathered by the search up to that point, and most important, on any extra knowledge about the problem domain.

Uniform Cost Search is a special case of the best first search algorithm. The algorithm maintains a priority queue of nodes to be explored. A cost function f(n) is applied to each node. The nodes are put in OPEN in the order of their f values. Nodes with smaller f(n) values are expanded earlier. The generic best first search algorithm is outlined below.

Best First Search
Let *fringe* be a priority queue containing the initial state Loop if *fringe* is empty return failure Node ?□remove-first (fringe) if Node is a goal then return the path from initial state to Node else generate all successors of Node, and put the newly generated nodes into fringe according to their f values End Loop

We will now consider different ways of defining the function f. This leads to different search algorithms.

3.2.1 Greedy Search

In greedy search, the idea is to expand the node with the smallest estimated cost to reach the goal.

We use a heuristic function
$f(n) = h(n)$
$h(n)$ estimates the distance remaining to a goal.

A greedy algorithm is any algorithm that follows the problem solving heuristic of making the locally optimal choice at each stage with the hope of finding the global optimum. In general, greedy algorithms are used for optimization problems.

Greedy algorithms often perform very well. They tend to find good solutions quickly, although not always optimal ones.

The resulting algorithm is not optimal. The algorithm is also incomplete, and it may fail to find a solution even if one exists. This can be seen by

running greedy search on the following example. A good heuristic for the route-finding problem would be straight-line distance to the goal.

S is the starting state, G is the goal state.
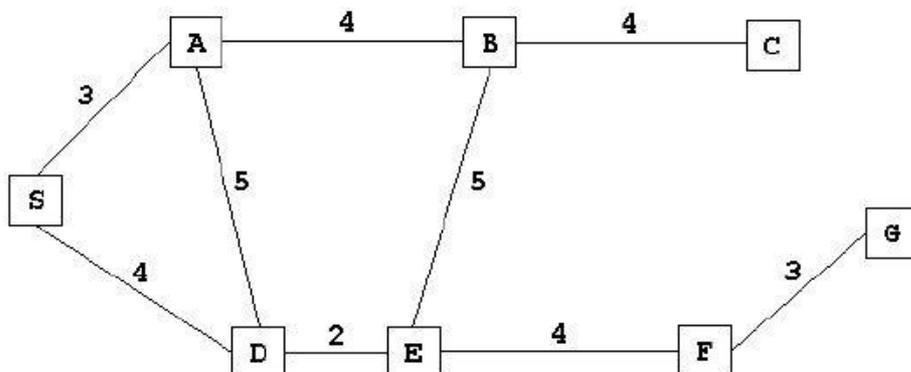
Figure 2 is an example of a route finding problem.
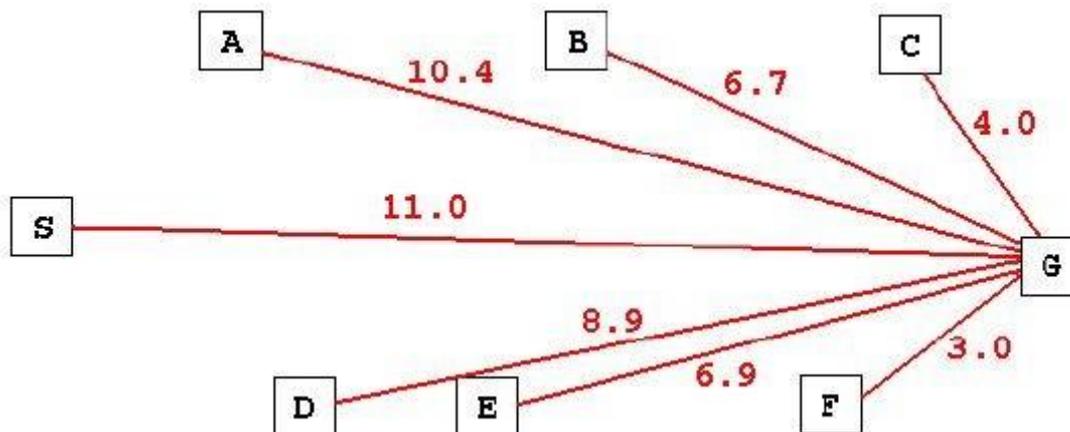
Figure 3 -The straight line distance heuristic estimates for the nodes.

Let us run the greedy search algorithm for the graph given in Figure 2. The straight line distance heuristic estimates for the nodes are shown in Figure 3.

Step 1: S is expanded. Its children are A and D.
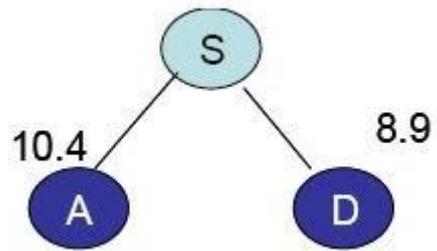
Step 2: D has smaller cost and is expanded next.
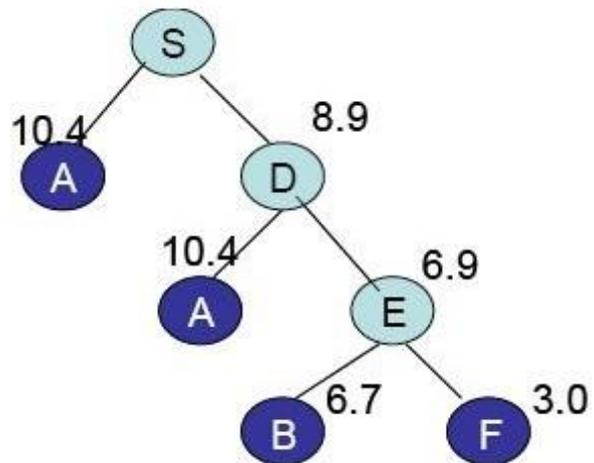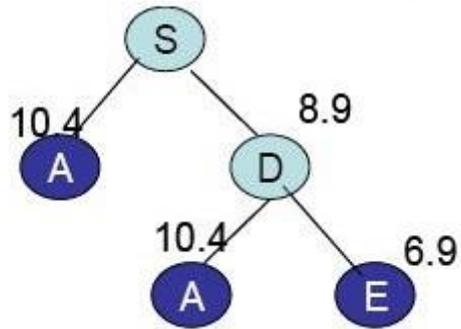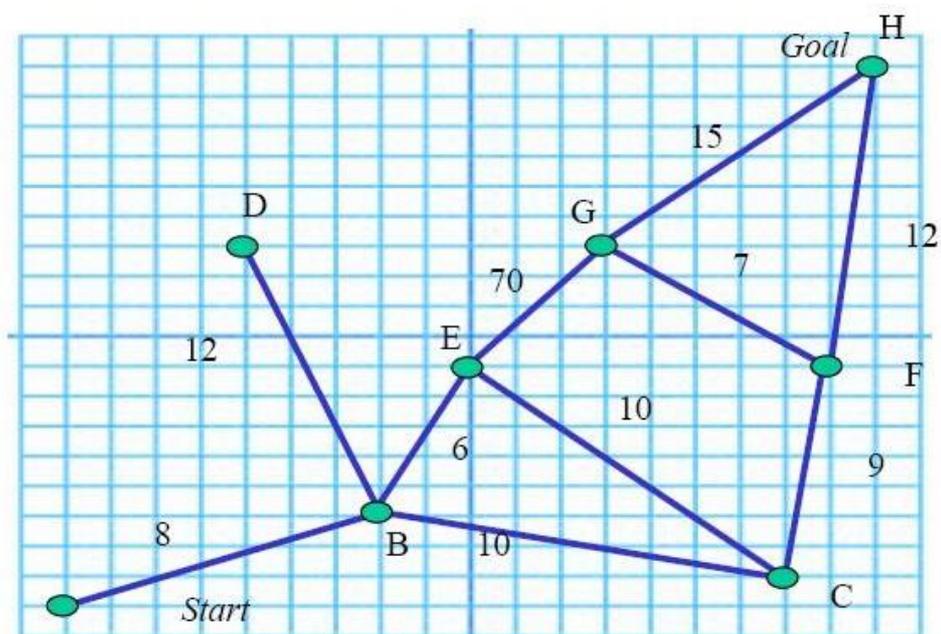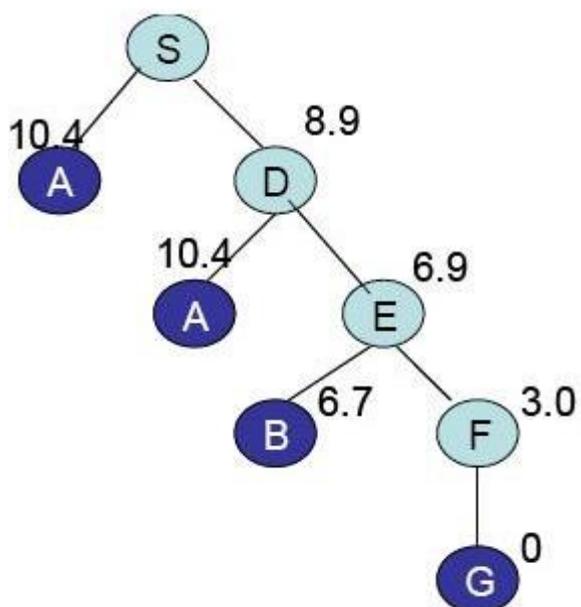
Figure 4

Greedy Best-First Search illustrated

We will run greedy best first search on the problem in Figure 2. We use the straight line heuristic. *h(n)* is taken to be the straight line distance from *n* to the goal position.

The nodes will be expanded in the following order:

    A
    B
    E
    G
    H

The path obtained is A-B-E-G-H and its cost is 99
Clearly this is not an optimum path. The path A-B-C-F-H has a cost of 39.

3.2.2 A* Search

We will next consider the famous A* algorithm. This algorithm was given by Hart, Nilsson & Rafael in 1968.

A* is a best first search algorithm with
$f(n) = g(n) + h(n)$
where
$g(n)$ = sum of edge costs from start to n
$h(n)$ = estimate of lowest cost path from n to goal

$f(n)$ = actual distance so far + estimated distance remaining

$h(n)$ is said to be admissible if it underestimates the cost of any solution that can be reached from n. If $C^*(n)$ is the cost of the cheapest solution path from n to a goal node, and if h is admissible,
$h(n) <= C^*(n)$.
We can prove that if $h(n)$ is admissible, then the search will find an optimal solution.

The algorithm A* is outlined below:

Algorithm A*
OPEN = nodes on frontier.   CLOSED = expanded nodes.
OPEN = {<s, nil>}
while OPEN is not empty
remove from OPEN the node <n,p> with minimum $f(n)$
place <n,p> on CLOSED
if n is a goal node,
return success (path p)
for each edge connecting n & m with cost c
if <m, q> is on CLOSED and {p|e} is cheaper than q
then remove n from CLOSED,
put <m,{p|e}> on OPEN
else if <m,q> is on OPEN and {p|e} is cheaper than q
then replace q with {p|e}
else if m is not on OPEN
then put <m,{p|e}> on OPEN

return failure

3.2.1 A* illustrated
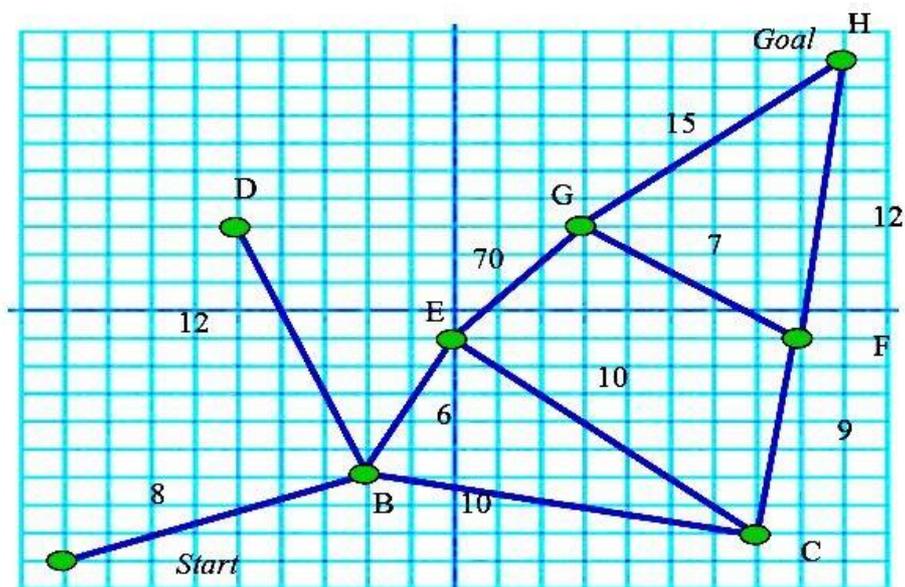
The heuristic function used is straight line distance. The order of nodes expanded, and the status of Fringe is shown in the following table.

Steps	Fringe	Node expanded	Comments
1	A		
2	B(26.6)	A	
3	E(27.5), C(35.1), D(35.2)	B	
4	C(35.1), D(35.2), C(41.2) G(92.5)	E	C is not inserted as there is another C with lower cost.
5	D(35.2), F(37), G(92.5)	C	
6	F(37), G(92.5)	D	
7	H(39), G(42.5)	F	G is replaced with a lower cost node
8	G(42.5)	H	Goal test successful.

The path returned is A-B-C-F-H.
The path cost is 39. This is an optimal path.

3.2.2 A* search: properties

The algorithm A* is admissible. This means that provided a solution exists, the first solution found by A* is an optimal solution. A* is admissible under the following conditions:

- In the state space graph
  - Every node has a finite number of successors
  - Every arc in the graph has a cost greater than some $\varepsilon > 0$
- Heuristic function: for every node n, $h(n) \le h^*(n)$

A* is also complete under the above conditions.

A* is optimally efficient for a given heuristic – of the optimal search algorithms that expand search paths from the root node, it can be shown that no other optimal algorithm will expand fewer nodes and find a solution

However, the number of nodes searched still exponential in the worst case.

Unfortunately, estimates are usually not good enough for A* to avoid having to expand an exponential number of nodes to find the optimal solution. In addition, A* must keep all nodes it is considering in memory.

A* is still much more efficient than uninformed methods.

It is always better to use a heuristic function with higher values as long as it does not overestimate.

A heuristic is consistent if:

$$h(n) <= cost(n, n') + h(n')$$

For example, the heuristic shown below is inconsistent, because $h(n) = 4$, but $cost(n, n') + h(n') = 1 + 2 = 3$, which is less than 4. This makes the value of f decrease from node n to node n':
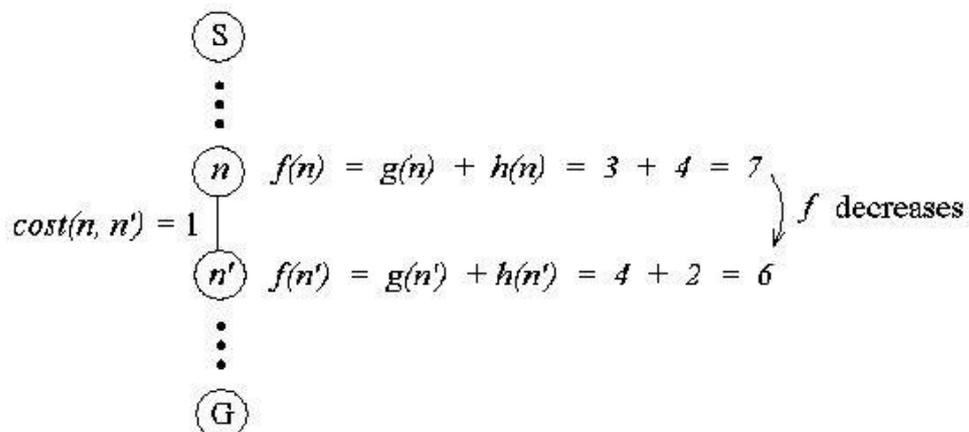
If a heuristic h is consistent, the f values along any path will be nondecreasing:

$f(n')$ = estimated distance from start to goal through n'

= actual distance from start to n + step cost from n to n' + estimated distance from n' to goal

= $g(n) + cost(n, n') + h(n')$

$\geq g(n) + h(n)$ because $cost(n, n') + h(n') \geq h(n)$ by consistency

= $f(n)$

Therefore $f(n') \geq f(n)$, so f never decreases along a path.

If a heuristic h is inconsistent, we can tweak the f values so that they behave as if h were consistent, using the **pathmax equation**:
$f(n') = \max(f(n), g(n') + h(n'))$

This ensures that the f values never decrease along a path from the start to a goal.
Given nondecreasing values of f, we can think of A* as searching outward from the start node through successive **contours** of nodes, where all of the nodes in a contour have the same f value:
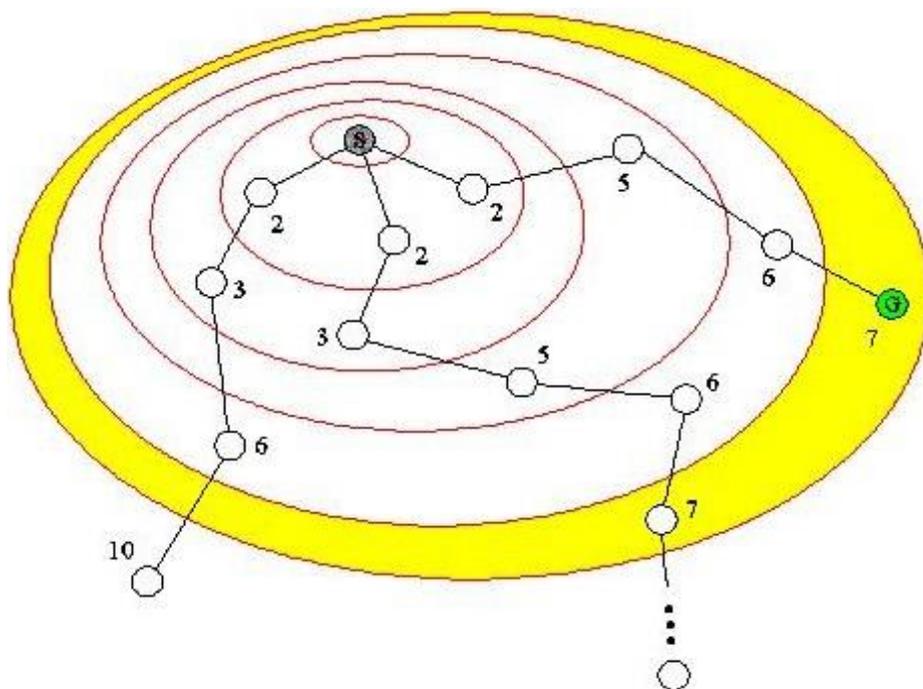
For any contour, A* examines all of the nodes in the contour before looking at any contours further out. If a solution exists, the goal node in the closest contour to the start node will be found first.

We will now prove the admissibility of A*.

3.2.3 Proof of Admissibility of A*

We will show that A* is admissible if it uses a monotone heuristic.

A monotone heuristic is such that along any path the f-cost never decreases.
But if this property does not hold for a given heuristic function, we can make the f value monotone by making use of the following trick (m is a child of n)
$f(m) = max (f(n), g(m) + h(m))$

- o   Let G be an optimal goal state
- o   C* is the optimal path cost.
- o   G2 is a suboptimal goal state: g(G2) > C*

Suppose A* has selected G2 from OPEN for expansion.
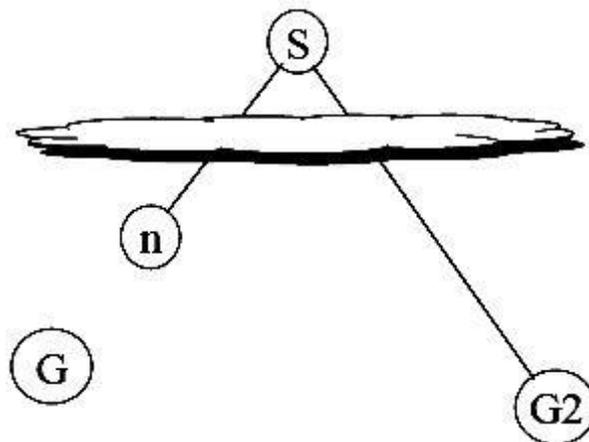
Consider a node n on OPEN on an optimal path to G. Thus $C^* \geq f(n)$
Since *n* is not chosen for expansion over G2, $f(n) \geq f(G2)$
G2 is a goal state. $f(G2) = g(G2)$
Hence $C^* \geq g(G2)$.
This is a contradiction. Thus A* could not have selected G2 for expansion before
reaching the goal by an optimal path.

3.2.4 Proof of Completeness of A*

Let G be an optimal goal state.
A* cannot reach a goal state only if there are infinitely many nodes where f(n) ≤ C*.
This can only happen if either happens:
   o   There is a node with infinite branching factor. The first condition takes care of
       this.
   o   There is a path with finite cost but infinitely many nodes. But we assumed that
       Every arc in the graph has a cost greater than some ε> 0. Thus if there are
       infinitely many nodes on a path g(n) > f*, the cost of that path will be infinite.

Lemma: A* expands nodes in increasing order of their f values.

A* is thus **complete** and **optimal**, assuming an admissible and consistent heuristic
function (or using the pathmax equation to simulate consistency).
A* is also **optimally efficient**, meaning that it expands only the minimal number of
nodes needed to ensure optimality and completeness.

3.2.4 Performance Analysis of A*

Model the search space by a uniform b-ary tree with a unique start state s, and a goal
state, g at a distance N from s.
The number of nodes expanded by A* is exponential in N unless the heuristic estimate is
logarithmically accurate
$|h(n) - h^*(n)| \leq O(\log h^*(n))$

In practice most heuristics have proportional error.

It becomes often difficult to use A* as the OPEN queue grows very large.
A solution is to use algorithms that work with less memory.

3.2.5 Properties of Heuristics

Dominance:
h2 is said to dominate h1 iff $h2(n) \geq h1(n)$ for any node n.
A* will expand fewer nodes on average using h2 than h1.

Proof:

Every node for which $f(n) < C^*$ will be expanded. Thus n is expanded whenever

$$h(n) < f^* - g(n)$$

Since $h2(n) \geq h1(n)$ any node expanded using h2 will be expanded using h1.

3.2.6 Using multiple heuristics

Suppose you have identified a number of non-overestimating heuristics for a problem:
$h1(n), h2(n), \ldots, hk(n)$

Then
$max (h1(n), h2(n), \ldots, hk(n))$
is a more powerful non-overestimating heuristic. This follows from the property of dominance

3.3   Beam Search

In computer science, beam search is a heuristic search algorithm that explores a graph by expanding the most promising node in a limited set. Beam search is an optimization of best-first search that reduces its memory requirements. Best-first search is a graph search which orders all partial solutions (states) according to some heuristic which attempts to predict how close a partial solution is to a complete solution (goal state). In beam search, only a predetermined number of best partial solutions are kept as candidates.

Beam search uses breadth-first search to build its search tree. At each level of the tree, it generates all successors of the states at the current level, sorting them in increasing order of heuristic cost.http://en.wikipedia.org/wiki/Beam_search - cite_note-1 However, it only stores a predetermined number of states at each level (called the beam width). The greater the beam width, the fewer states are pruned. With an infinite beam width, no states are pruned and beam search is identical to breadth-first search. The beam width bounds the memory required to perform the search. Since a goal state could potentially be pruned, beam search sacrifices completeness (the guarantee that an

algorithm will terminate with a solution, if one exists) and optimality (the guarantee that it will find the best solution).

The beam width can either be fixed or variable. One approach that uses a variable beam width starts with the width at a minimum. If no solution is found, the beam is widened and the procedure is repeated.

3.3.1 Name and Uses

The term "beam search" was coined by Raj Reddy, Carnegie Mellon University, 1976.

A beam search is most often used to maintain tractability in large systems with insufficient amount of memory to store the entire search tree. For example, it is used in many machine translation systems. To select the best translation, each part is processed, and many different ways of translating the words appear. The top best translations according to their sentence structures are kept and the rest are discarded. The translator then evaluates the translations according to a given criteria, choosing the translation which best keeps the goals. The first use of a beam search was in the Harpy Speech Recognition System, CMU 1976.

3.3.2 Extensions

Beam search has been made complete by combining it with depth-first search, resulting in Beam Stack Search and Depth-First Beam Search, and limited discrepancy search, resulting in Beam Search Using Limited Discrepancy Backtrackinghttp://en.wikipedia.org/wiki/Beam_search - cite_note-furcy-3 (BULB). The resulting search algorithms are anytime algorithms that find good but likely sub-optimal solutions quickly, like beam search, then backtrack and continue to find improved solutions until convergence to an optimal solution.

3.4   Hill climbing

In computer science, hill climbing is a mathematical optimization technique which belongs to the family of local search. It is an iterative algorithm that starts with an arbitrary solution to a problem, then attempts to find a better solution by incrementally changing a single element of the solution. If the change produces a better solution, an incremental change is made to the new solution, repeating until no further improvements can be found.

For example, hill climbing can be applied to the travelling salesman problem. It is easy to find an initial solution that visits all the cities but will be very poor compared to the optimal solution. The algorithm starts with such a solution and makes small improvements to it, such as switching the order in which two cities are visited. Eventually, a much shorter route is likely to be obtained.

Hill climbing is good for finding a local optimum (a solution that cannot be improved by considering a neighbouring configuration) but it is not guaranteed to find the best possible solution (the global optimum) out of all possible solutions (the search space). The characteristic that only local optima are guaranteed can be cured by using restarts (repeated local search), or more complex schemes based on iterations, like iterated local search, on memory, like reactive search optimization and tabu search, on memory-less stochastic modifications, like simulated annealing.

The relative simplicity of the algorithm makes it a popular first choice amongst optimizing algorithms. It is used widely in artificial intelligence, for reaching a goal state from a starting node. Choice of next node and starting node can be varied to give a list of related algorithms. Although more advanced algorithms such as simulated annealing or tabu search may give better results, in some situations hill climbing works just as well. Hill climbing can often produce a better result than other algorithms when the amount of time available to perform a search is limited, such as with real-time systems. It is an anytime algorithm: it can return a valid solution even if it's interrupted at any time before it ends.

3.4.1 Mathematical description

Hill climbing attempts to maximize (or minimize) a target function $f(x)$, where $\mathbf{x}$ is a vector of continuous and/or discrete values. At each iteration, hill climbing will adjust a single element in $\mathbf{x}$ and determine whether the change improves the value of $f(x)$. (Note that this differs from gradient descent methods, which adjust all of the values in $\mathbf{x}$ at each iteration according to the gradient of the hill.) With hill climbing, any change that improves $f(x)$ is accepted, and the process continues until no change can be found to improve the value of $f(x)$. $\mathbf{x}$ is then said to be "locally optimal".

In discrete vector spaces, each possible value for $\mathbf{x}$ may be visualized as a vertex in a graph. Hill climbing will follow the graph from vertex to

vertex, always locally increasing (or decreasing) the value of $f(\mathbf{x})$, until a local maximum (or local minimum) x_m is reached.

3.4.2 Variants

In simple hill climbing, the first closer node is chosen, whereas in steepest ascent hill climbing all successors are compared and the closest to the solution is chosen. Both forms fail if there is no closer node, which may happen if there are local maxima in the search space which are not solutions. Steepest ascent hill climbing is similar to best-first search, which tries all possible extensions of the current path instead of only one.

Stochastic hill climbing does not examine all neighbours before deciding how to move. Rather, it selects a neighbour at random, and decides (based on the amount of improvement in that neighbour) whether to move to that neighbour or to examine another.

Random-restart hill climbing is a meta-algorithm built on top of the hill climbing algorithm. It is also known as Shotgun hill climbing. It iteratively does hill-climbing, each time with a random initial condition x_0. The best x_m is kept: if a new run of hill climbing produces a better x_m than the stored state, it replaces the stored state.

Random-restart hill climbing is a surprisingly effective algorithm in many cases. It turns out that it is often better to spend CPU time exploring the space, than carefully optimizing from an initial condition.

4.0   CONCLUSION

Informed search strategies -Also known as "heuristic search," informed search strategies use information about the domain to (try to) (usually) head in the general direction of the goal node(s)

-Informed search methods: Hill climbing, best-first, greedy search, beam search, A, A*

5.0   SUMMARY

In this unit, you learnt that:

- Heuristic search is an AI search technique that employs heuristic for its moves.
- Best-first search is a search algorithm which explores a graph by expanding the most promising node chosen according to a specified rule.
- A greedy algorithm is any algorithm that follows the problem solving heuristic of making the locally optimal choice at each stage with the hope of finding the global optimum
- Beam search is a heuristic search algorithm that explores a graph by expanding the most promising node in a limited set
- Hill climbing is a mathematical optimization technique which belongs to the family of local search.

6.0   TUTOR-MARKED ASSIGNMENT

1.   What is A* Search?
2.   Consider the following table

	A	B	C	D	E	F	G	H	I	J	K	L	M
A		36	61										
B				31									
C				32		31						80	
D					52								
E							43						
F										122	112		
G													
H													
I								20		45			
J									40		36		
K													32
L													102
M													0

Using the A* algorithm work out a route from town A to town M. Use the following cost functions.

1.   G(n) = The cost of each move as the distance between each town (shown on map).
2.   H(n) = The Straight Line Distance between any town and town M. These distances are given in the table below.
                Provide the search tree for your solution and indicate the order in which you expanded the nodes. Finally, state the route you would take and the cost of that route.

7.0   REFERENCES/FURTHER READING

Lowerre, B. (1976). "The Harpy Speech Recognition System", Ph.D. thesis, Carnegie Mellon University.

Russell, S. J. & Norvig, P. (2003). *Artificial Intelligence: A Modern Approach*. (2nd ed.). Upper Saddle River, New Jersey: Prentice Hall, pp. 111–114, ISBN 0-13-790395-2.

Zhou, R. & Hansen, E. (2005). "Beam-Stack Search: Integrating Backtracking with Beam Search". http://www.aaai.org/Library/ICAPS/2005/icaps05-010.php

UNIT 4   TREE SEARCH

CONTENTS

1.0   INTRODUCTION

Tree search algorithms are specialised versions of graph search algorithms, which take the properties of trees into account. An example of tree search is the game tree of multiple-player games, such as chess or backgammon, whose nodes consist of all possible game situations that could result from the current situation. The goal in these problems is to find the move that provides the best chance of a win, taking into account all possible moves of the opponent(s). Similar problems occur when humans or machines have to make successive decisions whose outcomes are not entirely under one's control, such as in robot guidance or in marketing, financial or military strategy planning. This kind of problems has been extensively studied in the context of artificial intelligence. Examples of algorithms for this class are the minimax algorithm, alpha-beta pruning, and the A* algorithm.

2.0   OBJECTIVES

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

* describe a game tree
* describe some two-player games search algorithms
* explain intelligent backtracking
* solve some simple problems on tree search.

3.0   MAIN CONTENT

3.1   Game Tree

A game tree is a directed graph whose nodes are positions in a game and whose edges are moves. The complete game tree for a game is the game tree starting at the initial position and containing all possible moves from each position; the complete tree is the same tree as that obtained from the extensive-form game representation.
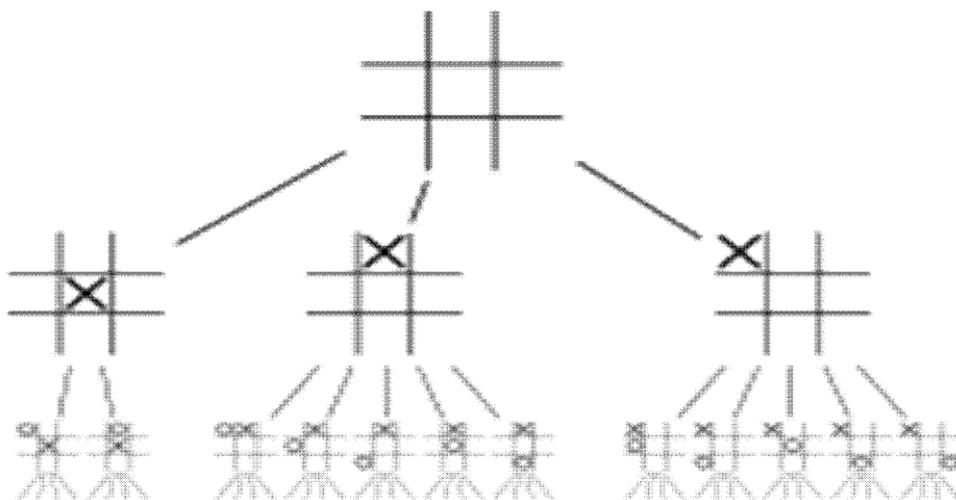
Figure 1:   Game tree for tic-tac-toe

The first two plies of the game tree for tic-tac-toe.

The diagram shows the first two levels, or *plies*, in the game tree for tic-tac-toe. We consider all the rotations and reflections of positions as being equivalent, so the first player has three choices of move: in the center, at the edge, or in the corner. The second player has two choices for the reply if the first player played in the center, otherwise five choices. And so on.

The number of leaf nodes in the complete game tree is the number of possible different ways the game can be played. For example, the game tree for tic-tac-toe has 26,830 leaf nodes.

Game trees are important in artificial intelligence because one way to pick the best move in a game is to search the game tree using the minimax algorithm or its variants. The game tree for tic-tac-toe is easily searchable, but the complete game trees for larger games like chess are much too large to search. Instead, a chess-playing program searches a partial game tree: typically as many plies from the current position as it

can search in the time available. Except for the case of "pathological" game trees (which seem to be quite rare in practice), increasing the search depth (i.e., the number of plies searched) generally improves the chance of picking the best move.

Two-person games can also be represented as and-or trees. For the first player to win a game there must be a winning move for all moves of the second player. This is represented in the and-or tree by using disjunction to represent the first player's alternative moves and using conjunction to represent all of the second player's moves.

3.2   Two-Player Games Search Algorithms

The second major application of heuristic search algorithms in Artificial Intelligence is two-player games. One1 of the original challenges of AI, which in fact predates the term, Artificial Intelligence, was to build a program that could play chess at the level of the best human players, a goal recently achieved.

Following are the algorithms meant to solve this problem.

- Minimax Search
- Alpha-Beta Pruning
- Quiecence
- Transposition Tables
- Limited Discrepancy Search
- Intelligent Backtracking

3.2.1 Minimax Search Algorithm

The standard algorithm for two-player perfect-information games such as chess, checkers or Othello is minimax search with heuristic static evaluation. The minimax search algorithm searches forward to a fixed depth in the game tree, limited by the amount of time available per move. At this search horizon, a heuristic function is applied to the frontier nodes. In this case, a heuristic evaluation is a function that takes a board position and returns a number that indicates how favourable that position is for one player relative to the other. For example, a very simple heuristic evaluator for chess would count the total number of pieces on the board for one player, appropriately weighted by their relative strength, and subtract the weighted sum of the opponent's places. Thus, large positive values would correspond to strange positions for one player called *MAX*, whereas large negative values would represent advantageous situation for the opponent called *MIN*.

Given the heuristic evaluations of the frontier nodes, minimax search algorithm recursively computes the values for the interior nodes in the tree according to the *maximum rule*. The value of a node where it is MAX's turn to move is the maximum of the values of its children, while the value of the node where MIN is to move is the minimum of the values of its children. Thus at alternative levels of the tree, the maximum values of the children are backed up. This continues until the values of the immediate children of the current position are computed at which point one move to the child with the maximum or minimum value is made depending on whose turn it is to move.

3.2.2   Alpha-Beta Pruning

One of the most elegant of all AI search algorithms is alpha-beta pruning. The idea, similar to branch-and-bound, is that the minimax value of the root of a game tree can be determined without examining all the nodes at the search frontier.

Only the labeled nodes are generated by the algorithm, with the heavy black lines indicating pruning. At the square node MAX is to move, while at the circular nodes it MIN's turn. The search proceeds depth-first to minimize the memory required, and only evaluates a node when necessary. First node and f are statically evaluated at 4 and 5, respectively, and their minimum value, 4 is backed up to their parent node d. Node h is then evaluated at 3, and hence the value of its parent node g must be less than or equal to 3, since it is the minimum of 3 and the unknown value of its right child. Thus, we level node g as <=3. The value of node c must be 4 then, because it is the maximum of 4 and a value that is less than or equal to 3. Since we have determined the minimax value of node c, we do not need to evaluate or even generate the brother of node h.

Similarly, after evaluating nodes k and l at 6 and 7 respectively, the backed up value of their parent node j is 6, the minimum of these values. This tells us that the minimax value of node I must be greater than or equal to 6 since it is the maximum of 6 and the unknown value of its right child. Since the value of node b is the minimum of 4 and a value that is greater than or equal to 6, it must be 4 and hence we achieve another cut off.

The right half of the tree shows an example of deep pruning. After evaluating the left half of the tree, we know that the value of the root node a is greater than or equal to four, the minimax value of node b. Once node q is equated at 1, the value of its parent node nine must be less than or equal to 1. Since the value of the root is greater than or

CIT478   *ARTIFICIAL INTELLIGENCE*

equal to two. Moreover, since the value of node m is the minimum of the value of node n and its brother, and node n has a value less than or equal to two, the value of node m must also be less than or equal to two. This causes the brother of node n to be pruned, since the value of the root node a is greater than or equal to four. Thus, we computed the minimax value of the root of the tree to be four, by generating only seven of sixteen leaf nodes in this area.

Since alpha-beta pruning performs a minimax search while pruning much of the tree, its effect is to allow a deeper search with the same amount of computation. This raises the question of how much does alpha-beta improve performance. The best way to characterize the efficiency of a pruning algorithm is in terms of its effective branching factor. The effective branching factor is the dth root of the frontier nodes that must be evaluated in a search to depth d, in the limit of large d.

The efficiency of alpha-beta pruning depends upon the order in which nodes are encountered at the search frontier. For any set of frontier node values, there exists same ordering of the values such that alpha-beta will not perform any cut offs at all. In that case, the effective branching factor is reduced from b to $b^{1/2}$., the square root of the brute-force branching factor. Another way of viewing the perfect ordering case is that for the same amount of computation, one can search twice as deep as with alpha-beta pruning as without since the search tree grows exponentially with depth, doubling the search horizon is a dramatic improvement.

In between worst-possible ordering and perfect ordering is random ordering, which is the average case. Under random ordering of the frontier nodes, alpha-beta pruning reduces the effective branching factor approximately b3/4. This means that one can search 4/3 as deep with alpha-beta, yielding as 33% improvement in search depth.

In practice, however, the effective branching factor of alpha-beta is closer to the best case of b1/2 due to node ordering. The idea of node ordering is that instead of generating the tree left to right, we can reorder the tree based on static evaluations of the interior nodes. In other words, the children of MAX nodes are expanded in decreasing order of their static values, while the children of MIN nodes are expanded in increasing order of their static values.

3.2.3 Quiescence Search

The idea of quiescence is that the static evaluator should not be applied to positions whose values are unstable, such as those occurring in the

middle of the piece trade. In those positions, a small secondary search is conducted until the static evaluation becomes more stable. In games such as chess or checkers, this can be achieved by always exploring any capture moves one level deeper. This extra search is called quiescence search. Applying quiescence search to capture moves quickly will resolve the uncertainties in the position.

3.2.4   What is Transposition Table?

A transposition table is a table of previously encountered game states, together with their backed-up minimax values. Whenever a new state is generated, if it is stored in the transposition table, its stored value is used instead of searching the tree below the node. Transposition table can be used very effectively so that reachable search depth in chess, for example, can be doubled.

3.2.5 Limited Discrepancy Search (LDS)

Limited Discrepancy Search (LDS) is a completely general tree-search algorithm, but is most useful in the context of constraint satisfaction problems in which the entire tree is too large to search exhaustively. In that case, we would like to search that subset of the tree that is most likely to yield a solution in the time available. Assume that we can heuristically order a binary tree so that at any node, the left branch is more likely to lead to a solution than the right branch. LDS then proceeds in a series of depth-first iterations. The first iteration explores just the left-most path in the tree. The second iteration explores those root-to-leaf paths with exactly one right branch, or discrepancy in them. In general, each iteration explores those paths with exactly k discrepancies, with k ranging from zero to the depth of the tree. The last iteration explores just the right most branch. Under certain assumptions, one can show that LDS is likely to find a solution sooner than a strict left-to-right depth-first search.

3.2.6 What is Intelligent Backtracking?

Performance of brute force backtracking can be improved by using a number of techniques such as variable ordering, value ordering, back jumping, and forward checking.

The order in which variables are instantiated can have a large effect on the size of the search tree. The idea of variable ordering is to order the variables form most constrained to least constrained. For example, if a variable has only a single value remaining that is consistent with the previously instantiated variable, it should be assigned that value

77

immediately. In general, the variables should be instantiated in increasing order of the size of their remaining domains. This can either be done statically at the beginning of the search or dynamically, reordering the remaining variables each time a variable is assigned a new value.

The order in which the value of a given variable is chosen determines the order in which the tree is searched. Since it does not affect the size of the tree, it makes no difference if all solutions are to be found. If only a single solution is required, however, value ordering can decrease the time required to find a solution. In general, one should order the values from least constraining to most constraining in order to minimize the time required to find a first solution.

An important idea, originally called back jumping, is that when an impasse is reached, instead of simply undoing the last decision made, the decision that actually caused the failure should be modified. For example, consider the three-variable problem where the variables are instantiated in the order x,y,z. Assume that values have been chosen for both x and y, but that all possible values for z conflict with the value chosen for x. In chronological backtracking, the value chosen for y would be changed, and then all the possible values for z would be tested again, to no avail. A better strategy in this case is to go back to the source of the failure, and change the value of x before trying different values for y.

When a variable is assigned a value, the idea of forward checking is to check each remaining uninstantiated variable to make sure that there is at least one assignment for each of them that is consistent with the previous assignments. If not, the original variable is assigned its next value.

4.0   CONCLUSION

In computer science, a search tree is a binary tree data structure in whose nodes data values are stored from some ordered set, in such a way that in-order traversal of the tree visits the nodes in ascending order of the stored values. This means that for any internal node containing a value v, the values x stored in its left sub tree satisfy $x \leq v$, and the values y stored in its right sub tree satisfy $v \leq y$. Each sub tree of a search tree is by itself again a search tree.

5.0   SUMMARY

In this unit, you learnt that:

- A game tree is a directed graph whose nodes are positions in a game and whose edges are moves
- The second major application of heuristic search algorithms in Artificial Intelligence is two-player games
- The standard algorithm for two-player perfect-information games such as chess, checkers or Othello is minimax search with heuristic static evaluation
- One of the most elegant of all AI search algorithms is alpha-beta pruning.
- The idea of quiescence is that the static evaluator should not be applied to positions whose values are unstable, such as those occurring in the middle of the piece trade.

6.0   TUTOR-MARKED ASSIGNMENT

Answer the following questions on informed search and heuristics:

1. Which of the following are admissible, given admissible heuristics $h1$, $h2$? Which of the following are consistent, given consistent heuristics $h1$, $h2$?
2. $h(n) = min\{h1(n), h2(n)\}$
3. $h(n) = wh1(n) + (1 - w)h2(n)$, where $0\ w\ 1$
4. $h(n) = max\{h1(n), h2(n)\}$
5. The heuristic path algorithm is a best-first search in which the objective function is $f(n) = (2 - w)g(n) + wh(n)$. For what values of w is this algorithm guaranteed to be optimal when h is admissible? What kind of search does this perform when $w = 0$? When $w = 1$? When $w = 2$?

7.0   REFERENCES/FURTHER READING

Christopher, D. Manning & Schutze, H. *Foundations of Statistical Natural Language Processing*. The MIT Press.

Hu, Te Chiang. Shing, M. (2002). *Combinatorial Algorithms*. Courier Dover Publications. ISBN 0486419622. http://books.google.com/?id=BF5_bCN72EUC. Retrieved 2007-04-02.

Nau, D. (1982). "An investigation of the causes of pathology in games". *Artificial Intelligence* 19: 257–278. doi:10.1016/0004-3702(82) 90002-9.

Allis, V. (1994). *Searching for Solutions in Games and Artificial Intelligence*. Ph.D. Thesis, University of Limburg, Maastricht, The Netherlands. ISBN 9090074880. http://fragrieu.free.fr/ SearchingForSolutions.pdf.

MODULE 3   ARTIFICIAL INTELLIGENCE TECHNIQUES   IN PROGRAMMING AND NATURAL LANGUAGES

Unit 1   Knowledge Representation
Unit 2   Programming Languages for Artificial Intelligence
Unit 3   Natural Language Processing

UNIT 1   KNOWLEDGE REPRESENTATION

CONTENTS

1.0   INTRODUCTION

Knowledge Representation (KR) has long been considered one of the principal elements of Artificial Intelligence, and a critical part of all problems solving [Newell, 1982]. The subfields of KR range from the purely philosophical aspects of epistemology to the more practical problems of handling huge amounts of data. This diversity is unified by the central problem of encoding human knowledge - in all its various forms - in such a way that the knowledge can be used. This goal is perhaps best summarized in the *Knowledge Representation Hypothesis:*

> Any mechanically embodied intelligent process will be comprised of structural ingredients that a) we as external observers naturally take to represent a propositional

account of the knowledge that the overall process exhibits, and b) independent of such external semantically attribution, play a formal but causal and essential role in engendering the behavior that manifests that knowledge [Smith, 1982].

A successful representation of some knowledge must, then, be in a form that is *understandable* by humans, and must cause the system using the knowledge to *behave* as if it knows it. The "structural ingredients" that accomplish these goals are typically found in the *languages* for KR, both implemented and theoretical, that have been developed over the years.

2.0   OBJECTIVES

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

- explain the meaning of knowledge representation
- describe the history of history of knowledge representation and reasoning
- list some characteristics of kr
- list 4 main features of kr language.

3.0   MAIN CONTENT

3.1   Overview of Knowledge Representation

Knowledge Representation (KR) research involves analysis of how to accurately and effectively reason and how best to use a set of symbols to represent a set of facts within a knowledge domain. A symbol vocabulary and a system of logic are combined to enable inferences about elements in the KR to create new KR sentences. Logic is used to supply formal semantics of how reasoning functions should be applied to the symbols in the KR system. Logic is also used to define how operators can process and reshape the knowledge. Examples of operators and operations include negation, conjunction, adverbs, adjectives, quantifiers and modal operators. Interpretation theory is this logic. These elements--symbols, operators, and interpretation theory--are what give sequences of symbols meaning within a KR.

A key parameter in choosing or creating a KR is its *expressivity*. The more expressive a KR, the easier and more compact it is to express a fact or element of knowledge within the semantics and grammar of that KR. However, more expressive languages are likely to require more complex logic and algorithms to construct equivalent inferences. A

highly expressive KR is also less likely to be complete and consistent. Less expressive KRs may be both complete and consistent. Auto epistemic temporal modal logic is a highly expressive KR system, encompassing meaningful chunks of knowledge with brief, simple symbol sequences (sentences). Propositional logic is much less expressive but highly consistent and complete and can efficiently produce inferences with minimal algorithm complexity. Nonetheless, only the limitations of an underlying knowledge base affect the ease with which inferences may ultimately be made (once the appropriate KR has been found). This is because a knowledge set may be exported from a knowledge model or knowledge base system (KBS) into different KRs, with different degrees of expressiveness, completeness, and consistency. If a particular KR is inadequate in some way, that set of problematic KR elements may be transformed by importing them into a KBS, modified and operated on to eliminate the problematic elements or augmented with additional knowledge imported from other sources, and then exported into a different, more appropriate KR.

In applying KR systems to practical problems, the complexity of the problem may exceed the resource constraints or the capabilities of the KR system. Recent developments in KR include the concept of the Semantic Web, and development of XML-based knowledge representation languages and standards, including Resource Description Framework (RDF), RDF Schema, Topic Maps, DARPA Agent Mark-up Language (DAML), Ontology Inference Layer (OIL), and Web Ontology Language (OWL).

There are several KR techniques such as frames, rules, tagging, and semantic networks which originated in Cognitive Science. Since knowledge is used to achieve intelligent behaviour, the fundamental goal of knowledge representation is to facilitate reasoning, drawing conclusions. A good KR must be both declarative and procedural knowledge. What is knowledge representation can best be understood in terms of five distinct roles it plays, each crucial to the task at hand:

- A knowledge representation (KR) is most fundamentally a surrogate, a substitute for the thing itself, used to enable an entity to determine consequences by thinking rather than acting, i.e., by reasoning about the world rather than taking action in it.
- It is a set of ontological commitments, i.e., an answer to the question: In what terms should I think about the world?
- It is a fragmentary theory of intelligent reasoning, expressed in terms of three components: (i) the representation's fundamental conception of intelligent reasoning; (ii) the set of inferences the

representation sanctions; and (iii) the set of inferences it recommends.

- It is a medium for pragmatically efficient computation, i.e., the computational environment in which thinking is accomplished. One contribution to this pragmatic efficiency is supplied by the guidance a representation provides for organizing information so as to facilitate making the recommended inferences.
- It is a medium of human expression, i.e., a language in which we say things about the world."

Some issues that arise in knowledge representation from an AI perspective are:

- How do people represent knowledge?
- What is the nature of knowledge?
- Should a representation scheme deal with a particular domain or should it be general purpose?
- How expressive is a representation scheme or formal language?
- Should the scheme be declarative or procedural?

There has been very little top-down discussion of the knowledge representation (KR) issues and research in this area is a well aged quillwork. There are well known problems such as "spreading activation" (this is a problem in navigating a network of nodes), "subsumption" (this is concerned with selective inheritance; e.g. an ATV can be thought of as a specialization of a car but it inherits only particular characteristics) and "classification." For example a tomato could be classified both as a fruit and a vegetable.

In the field of artificial intelligence, problem solving can be simplified by an appropriate choice of *knowledge representation*. Representing knowledge in some ways makes certain problems easier to solve. For example, it is easier to divide numbers represented in Hindu-Arabic numerals than numbers represented as Roman numerals.

3.1.1 Characteristics

A good knowledge representation covers six basic characteristics:

- Coverage, which means the KR covers a breath and depth of information. Without a wide coverage, the KR cannot determine anything or resolve ambiguities.
- Understandable by humans. KR is viewed as a natural language, so the logic should flow freely. It should support modularity and hierarchies of classes (Polar bears are bears, which are animals).

It should also have simple primitives that combine in complex forms.

- Consistency. If John closed the door, it can also be interpreted as the door was closed by John. By being consistent, the KR can eliminate redundant or conflicting knowledge.
- Efficient
- Easiness for modifying and updating.
- Supports the intelligent activity which uses the knowledge base

To gain a better understanding of why these characteristics represent a good knowledge representation, think about how an encyclopaedia (e.g. Wikipedia) is structured. There are millions of articles (coverage), and they are sorted into categories, content types, and similar topics (understandable). It redirects different titles but same content to the same article (consistency). It is efficient, easy to add new pages or update existing ones, and allows users on their mobile phones and desktops to view its knowledge base.

3.1.2 History of Knowledge Representation and Reasoning

In computer science, particularly artificial intelligence, a number of representations have been devised to structure information.

KR is most commonly used to refer to representations intended for processing by modern computers, and in particular, for representations consisting of explicit objects (the class of all elephants, or Clyde a certain individual), and of assertions or claims about them ('Clyde is an elephant', or 'all elephants are grey'). Representing knowledge in such explicit form enables computers to draw conclusions from knowledge already stored ('Clyde is grey').

Many KR methods were tried in the 1970s and early 1980s, such as heuristic question-answering, neural networks, theorem proving, and expert systems, with varying success. Medical diagnosis (e.g., Mycin) was a major application area, as were games such as chess.

In the 1980s formal computer knowledge representation languages and systems arose. Major projects attempted to encode wide bodies of general knowledge; for example the "Cyc" project (still ongoing) went through a large encyclopaedia, encoding not the information itself, but the information a reader would need in order to understand the encyclopaedia: naive physics; notions of time, causality, motivation; commonplace objects and classes of objects.

Through such work, the difficulty of KR came to be better appreciated. In computational linguistics, meanwhile, much larger databases of language information were being built, and these, along with great increases in computer speed and capacity, made deeper KR more feasible.

Several programming languages have been developed that are oriented to KR. Prolog developed in 1972, but popularized much later, represents propositions and basic logic, and can derive conclusions from known premises. KL-ONE (1980s) is more specifically aimed at knowledge representation itself. In 1995, the Dublin Core standard of metadata was conceived.

In the electronic document world, languages were being developed to represent the structure of documents, such as SGML (from which HTML descended) and later XML. These facilitated information retrieval and data mining efforts, which have in recent years begun to relate to knowledge representation.

Development of the Semantic Web, has included development of XML-based knowledge representation languages and standards, including RDF, RDF Schema, Topic Maps, DARPA Agent Markup Language (DAML), Ontology Inference Layer (OIL), and Web Ontology Language (OWL).

3.2   Knowledge Representation Languages

William Woods defines the properties of a KR Language as follows:
A KR language must unambiguously represent any interpretation of a sentence (logical adequacy), have a method for translating from natural language to that representation, and must be usable for reasoning [Woods, 1975].

Wood's definition is merely a simplification of the KR Hypothesis where "reasoning" is the only method of "engendering the behavior that manifests that knowledge." Reasoning is essential to KR, and especially to KR languages, yet even simple reasoning capabilities can lead to serious tractability problems [Brachman and Levesque, 1987], and thus must be well understood and used carefully.

One of the most important developments in the application of KR in the past 20 years has been the proposal [Minsky, 1981], study [Woods, 1975] [Brachman, 1977] [Brachman, 1979], and development [Brachman and Schmolze, 1985] [Fox, Wright, and Adam, 1985] [Bobrow and Winograd, 1985] of frame-based KR languages. While

frame-based KR languages differ in varying degrees from each other, the central tenet of these systems is a notation based on the specification of objects (concepts) and their relationships to each other. The main features of such a language are:

1. *Object-orientedness*. All the information about a specific concept is stored with that concept, as opposed, for example, to rule-based systems where information about one concept may be scattered throughout the rule base.
2. *Generalization/Specialization*. Long recognized as a key aspect of human cognition [Minsky, 1981], KR languages provide a natural way to group concepts in hierarchies in which higher level concepts represent more general, shared attributes of the concepts below.
3. *Reasoning*. The ability to state in a formal way that the existence of some piece of knowledge implies the existence of some other, previously unknown piece of knowledge is important to KR. Each KR language provides a different approach to reasoning.
4. *Classification*. Given an abstract description of a concept, most KR languages provide the ability to determine if a concept fits that description, this is actually a common special form of reasoning.

```
article-10::
title: "Domain Modeling"
author: person-1
published-in: journal-64
journal-64::
name: "AI Magazine"
volume: 10
number: 4
location: Library
Figure A A simple database representation.
```

Object orientation and generalization help to make the represented knowledge more understandable to humans; reasoning and classification help make a system behave as if it knows what is represented. Frame-based systems thus meet the goals of the KR Hypothesis.

It is important to realize both the capabilities and limitations of frame-based representations, especially as compared to other formalisms. To begin with, all symbolic KR techniques are derived in one way or

another from First Order Logic (FOL), and as a result are suited for representing knowledge that doesn't change.(Figure one Simple database representation) . Different KR systems may be able to deal with non-monotonic changes in the knowledge being represented, but the basic assumption has been that change, if present, is the exception rather than the rule.

Two other major declarative KR formalisms are *production systems* and *database systems*. Production systems allow for the simple and natural expression of *if-then rules*. However, these systems have been shown to be quite restrictive when applied to large problems, as there is no ordering of the rules, and inferences cannot be constrained away from those dealing only with the objects of interest. Production systems are subsumed by frame-based systems, which additionally provide natural inference capabilities like classification and inheritance, as well as knowledge-structuring techniques such as generalization and object orientation.

Database systems provide only for the representation of simple assertions, without inference. Rules of inference are important pieces of knowledge about a domain. For example, consider the bibliographic database in Figure 1. If someone were interested in `article-10` and wanted to know where it was, that person would have to be smart enough to realize that *an article can be found in the location of the journal in which it is published*. That sentence is a rule, it is *knowledge*. It is knowledge that cannot be expressed in a database system. The person doing the retrieval of the information in the database must have that knowledge in order to type the SQL statement that will get the proper location. In a frame-based system that knowledge can be expressed as a rule that will fire when `article-10` is accessed, thus the user is not required to know it.

Figure A A framework for Domain Modeling
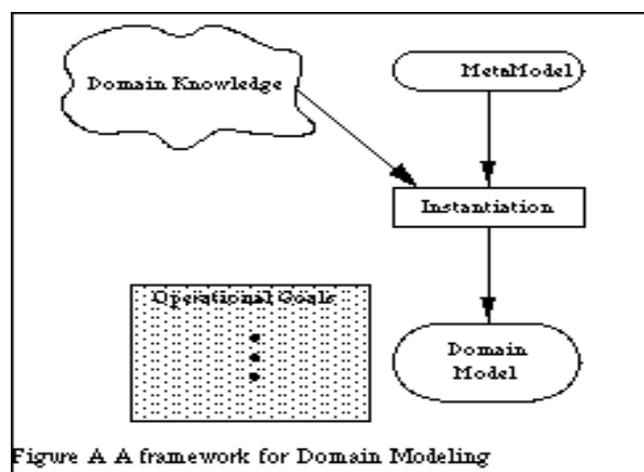
Frame-based systems are currently severely limited when dealing with *procedural knowledge* [Winograd, 1975]. An example of procedural knowledge would be Newton's Law of Gravitation - the attraction between two masses is inversely proportional to the square of their distances from each other. Given two frames representing the two bodies, with slots holding their positions and mass, the value of the gravitational attraction between them cannot be (Figure 2-A framework for Domain Modelling).

Inferred declaratively using the standard reasoning mechanisms available in frame-based KR languages, though a function or procedure in a programming language could represent the mechanism for performing this "inference" quite well. Frame-based systems that can deal with this kind of knowledge do so by adding a procedural language to its representation. The knowledge is *not* being represented in a frame-based way, it is being represented as C or (more commonly) LISP code which is accessed through a slot in the frame [Bobrow and Winograd, 1985]. This is an important distinction - there is knowledge being.

Encoded in those LISP functions that is not fully accessible. The system can reason *with* that knowledge, but not *about* it; in other words we can use some attached procedure to compute (or infer) the value of one slot based on some others, but we cannot ask how that value was obtained.

3.3   Domain *Modeling*

Domain modeling is the field in which the application of KR to specific domains is studied and performed. Figure 2 shows a framework for discussing domain modeling that seems to map well onto most examples [Iscoe, Tam, and Liu, 1991].

The amorphous shape labelled *Domain Knowledge* refers to the knowledge possessed by the domain expert that must be encoded in some fashion. This knowledge is not well defined and is fairly difficult for others to access. The box labelled *Meta-Model* refers to the KR formalism, typically a KR language that will be used as the *symbol level* [Newell, 1982] for the machine representation of this knowledge. The box labelled *instantiation* refers to the *process* of taking the domain knowledge and physically representing it using the meta-model, this process is sometimes referred to as *knowledge acquisition* [Schoen, 1991]. The box labelled *domain model* refers to the knowledge-base that results from the instantiation, and the *operational goals* are typically not represented formally, but refer to the reason the domain model was built and what it will be used for.

Specific examples of real-world domain modelling efforts and how they fit into this framework can be found in [Iscoe, 1991], and it has become clear that the most prevalent operational goal across modelling efforts today is understanding the domain of a large software system [Arango, 1989]. One thing that seems to be universally lacking in efforts with this operational goal is the realization that a software system operating within a domain *is a part of that domain*, and deserves as much attention and detail in the model as any other part. The main reason for this oversight is that there is a historical reason for distinguishing *procedural* from *declarative* knowledge [Winograd, 1975], and as a result the two are typically represented differently: domain models are represented with frame based KR languages and programs are represented with programming languages.

This traditional separation between programs and domain models causes problems during the instantiation of a domain model that includes not only knowledge of the objects and attributes, but knowledge of the procedural aspects of the processes associated with the domain as well. The problems stem from the fact that domain modelling is a discipline in which advances are made incrementally, by building upon previous systems [Simon, 1991]. Some of the most significant results are in the form of methodologies which help other domain modellers to avoid pitfalls and use techniques that work [Gruber, 1993].

The predominant methodologies for domain modelling clearly indicate that the instantiation of the model is the most time consuming part, and that the most important part of instantiation is *ontological analysis* [Alexander, Freiling, and Shulman, 1986] (which is more fully described in the next section). Ontologies for general taxonomies of objects are abundant, and there seem to be clear guidelines for developing new ones.

The problem is that for the knowledge required to represent procedural knowledge and reason about it (not with it); there are few guidelines, especially when the procedures requiring representation are implemented as software. There is not much background to draw upon, other than software information systems, as far as ontologies and methodologies for modelling what software does. Ontological analysis ended up being a large part of the effort for this research, since it had never been done before.

3.4   Ontological Analysis

The word *ontology* means "the study of the state of being." *Ontology* describes the states of being of a particular set of things. This

description is usually made up of axioms that define each thing. In knowledge representation, ontology has become the defining term for the part of a domain model that excludes the *instances*, yet describes what they can be. Ontological analysis is the process of defining this part of the model.

What makes up a specific domain ontology is restricted by the representational capabilities of the *meta-model* - the language used to construct the model. Each knowledge representation language differs in its manner and range of expression. In general, ontology consists of three parts: *concept* definitions, *role* definitions, and further inference definitions.

The concept definitions set up all the *types* of objects in the domain. In object oriented terms this is called the class definitions, and in database terms these are the entities. There can be three parts to the concept definitions:

- Concept taxonomy. The taxonomy is common to most knowledge representation languages, and through it is specified the nature of the categories in terms of generalization and specialization.
- Role defaults which specify for each concept what the default values are for any attributes.
- Role restrictions which specify for a concept any constraints on the values in a role, such as what types the values must be, how many values there can be, etc.

A role is an attribute of an object. In object-oriented terms it is a slot, in database terms (and even some KR languages) it is a relation. In the simplest case, a role for an object just has a value; the object mailbox-4 might have a role number-of-messages, for example, that would have a value which is a number. Roles also express relationships between objects. The same object might have a role called owner which relates mailbox-4 to the object person-2. Roles which represent relationships are unidirectional. A role definition may have up to three parts as well:

- The role taxonomy which specifies the generalization/specialization relationship *between* roles. For example, ontology for describing cars might include roles called has-engine, has-seats, and has-headlights, which relate objects that represent cars to objects that represent engines, seats, and headlights, resp. The role has-parts, then, could be expressed as the generalization of all these roles, and the result is that all the

values of all the more specialized roles would also be values of the more general role.

- Role inverses which provide a form of inference that allows the addition of a role in the opposite direction when the forward link is made. For example, if the inverse of has-engine was engine-of, then when the has-engine link between the object that represents the car and the object that represents the engine is made, the engine-of link will automatically be added between the engine object and the car object.

- Role restrictions. The role itself may be defined such that it can only appear between objects of certain types (domain/range restrictions), or can only appear a specified number of times (cardinality restriction). This is the same information specified in role restriction for concepts, some representation languages consider this information to be part of the role, and some consider it to be part of the concept.

The final part of ontology is the specification of additional inference that the language provides. Examples of this are forward and/or backward chaining rules, path grammars, subsumption and/or classification, demons, etc. An explanation of the inference mechanisms used in this research will be given in the next section.

3.5   Classic

Classic is a frame-based knowledge representation language that belongs to the family of *description logics* [Brachman, et al., 1991]. It is descended from KL-ONE [Brachman and Schmolze, 1985], and has been specifically designed to be mindful of the tractability of its own inferences [Brachman, et al., 1989].

Classic knowledge-bases are composed of four kinds of objects: *concepts*, *roles*, *rules*, and *individuals*. Ontology in Classic consists of concept taxonomy, role taxonomy, role inverses, role restrictions and defaults. The role restrictions and defaults are specified as part of the concept definitions. Classic rules are forward chaining rules.

3.5.1 The Classic Language

Throughout this document, it has been necessary to make explicit use of the notation and terminology of Classic to explain and describe some of the more detailed aspects of this research. This section contains a brief introduction to the language of Classic in order to make it clear precisely how one goes about describing objects. This introduction only presents the subset of Classic which was used in this research. The Classic

language is specifically designed to make it possible to describe objects in such a way that it is possible to determine automatically whether one object is subsumed by another. The peculiarities of the language arise from the goal of making this determination not only possible, but tractable.

To begin with, a typical concept description looks like this:

```
(defconcept information-filter
 (and kbeds-object
     (all information-filter-of valid-mail-recipient)
     (at-least one information-filter-of)
     (all has-filter-condition kbeds-mail-message)
     (at-least one has-filter-condition)
     (at-most one has-filter-condition)
     (all has-filter-action kbeds-filter-action)
     (at-least one has-filter-action))
 :disjoint kbeds-thing)
```

This says an `information-filter` is subsumed by (or is more specialized than, or is a subclass of) `kbeds-object` (and therefore is also described by that concept), and that all the *fillers* (in Classic a filler is the value of a particular role on a particular object) for its `information-filter-of` role must be individuals of `valid-mail-recipient`, and that there must be at least one filler for that role, and similarly for the role `has-filter-condition` except that there can also be at most one filler (in other words, there can be only one filler), and so on. The disjoint specification identifies this concept as a member of a disjoint set, which means that an individual cannot be subsumed by more than one concept in that set. The most obvious example of a disjoint set would be the gender set, containing the concepts "male person" and "female person."

In the terminology of Classic, the description above is *told* information. Told information is precisely the information that is explicitly typed into the knowledge base. This is opposed to *derived* information, which is all the information that Classic derives or infers through its various mechanisms. (Figure 3 – A Simple taxonomy of primitive concepts).

This description actually shows a *primitive* concept - one which Classic will not automatically try to classify. Classic will also not automatically try to find which individuals are subsumed by a primitive concept, this information must be told. This subtle notion may seem irrelevant, but it is the norm in most representation languages (when a concept is created the modeller explicitly names the parent concepts, and when an

individual is created, the modeller explicitly names the concept that the new individual is an instance of). It is important in Classic because there is another kind of concept, the *defined* concept, which Classic actually does automatically classify and find individuals of. For example, in figure 3 a simple taxonomy of primitive concepts is shown. Let us suppose we create a defined concept called `vegetarian-mammal` as follows: `(and mammal (all food plant))`. Next we create another defined concept called `fruit-eating-person`: `(and person (all food fruit))`. Classic will derive that `vegetarian-mammal` *subsumes* `fruit-eating-person` (why?

Figure A A simple taxonomy of primitive concepts.

because `mammal` subsumes `person` and `plant` subsumes `fruit`). If we created two individuals, `joe` and `apple`, and tell Classic that they are instances of `person` and `fruit`, resp., and further tell Classic that `apple` is a filler for `joe's food` role, Classic will derive that `joe` is an instance of `fruit-eating-person` (and therefore also

Figure A A simple taxonomy of primitive concepts.

`vegetarian-mammal`). Again, Classic will never derive that an individual is an instance of a primitive concept, it must always be told that.
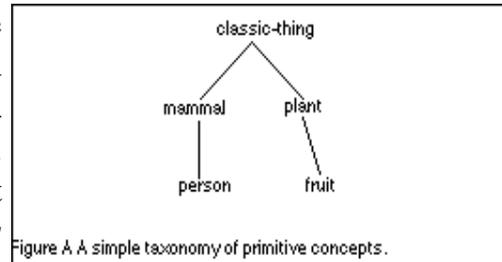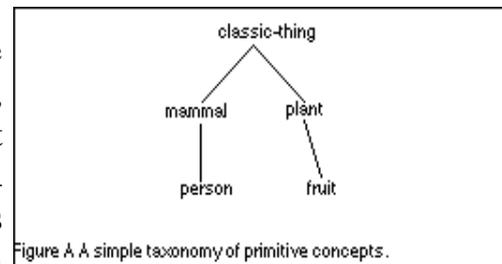
This automatic classification of individuals of defined concepts through subsumption is a simple, yet extremely powerful process. It is the key to several significant advances in software information systems described in later sections.

Another important point about Classic is the *Open World Assumption*. Classic does not assume that the information it knows is all the information there is. Returning to the example above, we have the following *told* information about two individuals, `joe: (and person (fills food apple))`, and `apple: fruit`. Classic will then add all the *derived* information it can to these individuals, yielding `joe: (and person mammal classic-thing (fills food apple))`, and `apple: (and fruit plant classic-thing)`. Where is the information about `joe` being a `fruit-eating-person`? The truth is that Classic cannot derive this yet. The definition of `fruit-eating-person` specifies that *all* the fillers for

the `food` role of an individual must be instances of `fruit`. Classic does not assume that because it knows one (or two, or zero, etc.) fillers for an individual's role, that it knows them all. In fact, Classic assumes the opposite: it assumes it does not know them all.

There are two ways for Classic to figure out that it knows all the fillers for an individual's role. The first way is for it to be told, by *closing* the role. When a role is closed on an individual, it tells Classic that there can be no more filler. In the above example, the user would have to close the `food` role on `joe` in order for Classic to derive that `joe` is an instance of `fruit-eating-person` (Classic always tries to reclassify individuals when their roles are closed). The second way is for Classic to derive that a role is closed on an individual if there is an *at-most* restriction on the role. For example, if the concept `person` (or `mammal`) additionally had `(at-most one food)` in its description, then since `joe` is told to be an instance of `person` that restriction would apply to him, and since he already has one filler in his `food` role, *and* since he can have at most one filler in his `food` role, he can have no more and the role is derived to be closed.

The final part of ontology in Classic is the rules. Classic rules come in two forms, *description* rules and *filler* rules. All classic rules have as their antecedent a named concept, and are fired on an individual when the individual is classified as an instance of the concept.

The consequent of a classic description rule is a classic description which, when the rule fires on an individual, is merged into the description of the individual. For example, if we had a description rule like: `vegetarian-mammal -->` *(at-most 0 has-prey)*, the rule would fire on `joe` when he is classified as a `fruit-eating-person` and would add the at-most restriction to the description of `joe`. Classic would then also derive that `joe`'s `has-prey` role is closed as well.

The consequent of a classic filler rule is the name of a role and a LISP function that will be invoked when the rule fires. The function is passed the individual the rule fired on and the role named in the consequent, and returns a list of new fillers for that role and individual. One useful application for filler rules is to create inter-role dependencies. For example, the concept `rectangle` has three roles: `length`, `width`, and `area`. We could define a function for calculating the area in LISP as follows:

```
(defun calculate-area (rect role)
```

```
(let ((length (car (cl-fillers rect @length)))
     (width (car (cl-fillers rect @width))))
  (* length width)))
```

And then define a filler rule: rectangle --> *area calculate-area*, the filler for the area role would automatically be generated based on the fillers in the length and width roles.

3.5.2 Enhancements to Classic

It was necessary to extend Classic in several ways in order to support this research. Each extension had a different motivation, which may not be entirely clear until that aspect of the research is discussed. These extensions are explained here, however, so that the sections involving the research do not need to sidetrack into explanations of the underlying support.

The first extension to Classic was a facility for supporting what some other representation languages call *path grammars* or role *transitivity* [Fox, Wright, and Adam, 1985]. A very common form of inference in frame-based representations is one in which the fillers for one role in a class of individuals can always be found by following the same *role path*. For example, an individual representing an article in a journal might have a role called `published-in` which is filled with an individual representing a journal. Journal individuals could have a role called `location` which is filled with some string indicating the place where the journal is physically located. It makes sense that the article individual should also have a location that is *the same as the location of the journal it is published in*, and this can be represented as a path rule. A path rule, like all Classic rules, has for its antecedent a concept name, and for its consequent a role and a role path (an ordered list of roles). When a rule fires on an individual, classic follows the role path to the end, and fills the role specified in the rule with all the values it finds at the end of the path. In the journal article example, the path rule would be: `article --> location (published-in location)`. An individual of `article` might be described: `(and article (fills published-in journal-10))`, and the `journal` individual `journal-10`: `(and journal (fills location "Shelf 10"))`. Classic would fire the path rule on the individual of `article` and follow the path: the first part of the path is `published-in`, which gets us to `article-10`, and the next part of the path is `location` which gets us to the string `"Shelf 10."` This value is then derived to be the filler for the `location` role of the individual of `article`. If a particular path ends "early," that is, the path leads to an intermediate individual that has no fillers for the next

role in the path, no fillers are returned for that particular branch of the path.

The path rule facility was further expanded to allow for the expression of *specialization overrides*. A specialization override is a type of inference in which a value that would normally be derived to fill a role is blocked if and only if there is already a value filling the role that is more specialized. The most common example of this is in object-oriented languages, a class inherits all the methods of its superclass, except the ones that are already defined by the class.
The next enhancement to Classic was a facility for dumping individuals into a file; in essence there is no way to save the current state of a classic knowledge-base. While this may not sound like a significant extension, there is one aspect of dumping (or, more accurately, of loading) a knowledge-base that is very intricate: the order in which roles are closed. When a role is told to be closed on an individual, it means there can be no more filler for that role - told or derived. However, when derived role filler depends on filler or fillers in other individuals, the role cannot be closed until the fillers it depends on are closed. There is an implicit ordering of role closing based on all the ways Classic can derive information.

The most significant enhancement to Classic was the addition of a facility for representing spanning objects [Welty and Ferrucci, 1994]. In reality, this enhancement is in the process of being made to Classic by its support team, and the spanning object facility used for this research was actually applied to the "dumped" knowledge-based - that is, the spanning functions worked by generating a text file containing Classic descriptions, then the knowledge-base was cleared and the text file could be loaded in. Until support for multiple universes of discourse is added to Classic (which will happen in the next major release), this was the only way to proceed. The only limitation this presented was an inability to change the first universe from the second.

4.0   CONCLUSION

Knowledge representation (KR) is an area of artificial intelligence research aimed at representing knowledge in symbols to facilitate inferencing from those knowledge elements, creating new elements of knowledge. The KR can be made to be independent of the underlying knowledge model or knowledge base system (KBS) such as a semantic network.

5.0   SUMMARY

In this unit, you learnt that:

- Knowledge Representation (KR) research involves analysis of how to accurately and effectively reason and how best to use a set of symbols to represent a set of facts within a knowledge domain.
- A good knowledge representation covers six basic characteristics
- In computer science, particularly artificial intelligence, a number of representations have been devised to structure information.
- A KR language must unambiguously represent any interpretation of a sentence (logical adequacy), have a method for translating from natural language to that representation, and must be usable for reasoning [Woods, 1975].
- Classic is a frame-based knowledge representation language that belongs to the family of *description logics* [Brachman, et al., 1991]

6.0   TUTOR-MARKED ASSIGNMENT

1.   List four (4) Characteristics of Knowledge representation.
2.   Explain Knowledge representation.
3.   Explain Domain Modeling in KR.

7.0   REFERENCES/FURTHER READING

Walker, A. McCord, M., John, F. Sowa, & Walter, G. Wilson (1990). Knowledge *Systems and Prolog* (Second Edition). Addison-Wesley.

Arthur, B. M. (1998). *Knowledge Representation.* Lawrence Erlbaum Associates.

Chein, M. ; Mugnier, M.L. (2009). *Graph-based Knowledge Representation: Computational Foundations of Conceptual Graphs*. Springer. ISBN 978-1-84800-285-2.

Davis, R.; Shrobe, H. E. Representing Structure and Behavior of Digital Hardware, IEEE Computer, Special Issue on Knowledge Representation, 16(10):75-82.

Hermann Helbig K. (2006). k*nowledge Representation and the Semantics of Natural Language*. Springer, Berlin, Heidelberg, New York.

Jean-Luc, Hainaut, Jean-Marc, H.; Vincent Englebert, Henrard, J., Roland, D. Understanding Implementations of IS-A Relations. ER 1996: 42-57.

Jeen Broekstraa, Michel Klein, Stefan Deckerc, Dieter Fenselb, Frank van Harmelenb and Ian Horrocks Enabling knowledge representation on the Web by extending RDF Schema, , April 16 2002.

John F. Sowa (2000). *Knowledge Representation*: Logical, Philosophical, and Computational Foundations. New York: Brooks/Cole.

Philippe Martin "Knowledge representation in RDF/XML, KIF, Frame-CG and Formalized-English", Distributed System Technology Centre, QLD, Australia, July 15-19, 2002

Pople H, Heuristic Methods for imposing structure on ill-structured problems, in AI in Medicine, Szolovits (ed.). AAAS Symposium 51, Boulder: Westview Press.

Randall Davis, Howard Shrobe, and Peter Szolovits; What Is a Knowledge Representation? AI Magazine, 14(1):17-33,1993

Ronald Fagin, Joseph Y. Halpern, Yoram Moses, Moshe Y. Vardi *Reasoning About Knowledge*, MIT Press, 1995, ISBN 0-262-06162-7.

Ronald J. Brachman, Hector J. Levesque (eds). *Readings in Knowledge Representation*, Morgan Kaufmann, 1985, ISBN 0-934613-01-X

Ronald J. Brachman, Hector J. Levesque *Knowledge Representation and Reasoning*, Morgan Kaufmann, 2004 ISBN 978-1-55860-932-7

Ronald J. Brachman; What IS-A is and isn't. An Analysis of Taxonomic Links in Semantic Networks; IEEE Computer, 16 (10); October 1983

Russell, Stuart J.; Norvig, Peter (2010). Artificial Intelligence: A Modern Approach (3rd ed.). Upper Saddle River, New Jersey: Prentice Hall, ISBN 0-13-604259-7, p. 437-439.

UNIT 2 PROGRAMMING LANGUAGES FOR ARTIFICIAL INTELLIGENCE

CONTENTS

1.0 INTRODUCTION

Artificial intelligence researchers have developed several specialized programming languages for artificial intelligence:

- IPL was the first language developed for artificial intelligence. It includes features intended to support programs that could perform general problem solving, including lists, associations, schemas (frames), dynamic memory allocation, data types, recursion, associative retrieval, functions as arguments, generators (streams), and cooperative multitasking.

- Lisp is a practical mathematical notation for computer programs based on lambda calculus. Linked lists are one of Lisp languages'

major data structures, and Lisp source code is itself made up of lists. As a result, Lisp programs can manipulate source code as a data structure, giving rise to the macro systems that allow programmers to create new syntax or even new domain-specific programming languages embedded in Lisp. There are many dialects of Lisp in use today, among them are Common Lisp, Scheme, and Clojure.

- Prolog is a declarative language where programs are expressed in terms of relations, and execution occurs by running *queries* over these relations. Prolog is particularly useful for symbolic reasoning, database and language parsing applications. Prolog is widely used in AI today.

- A STRIP is a language for expressing automated planning problem instances. It expresses an initial state, the goal states, and a set of actions. For each action preconditions (what must be established before the action is performed) and post conditions (what is established after the action is performed) are specified.

- Planner is a hybrid between procedural and logical languages. It gives a procedural interpretation to logical sentences where implications are interpreted with pattern-directed inference.

AI applications are also often written in standard languages like C++ and languages designed for mathematics, such as MATLAB and Lush. This unit will deal only on IPL, Lisp and Prolog.

2.0   OBJECTIVES

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

- describe the history of IPL
- discuss the similarities between lisp and prolog programming
- list the areas where lisp can be used.

3.0   MAIN CONTENT

3.1   IPL Programming Language

Information Processing Language (IPL) is a programming language developed by Allen Newell, Cliff Shaw, and Herbert Simon at RAND Corporation and the Carnegie Institute of Technology from about 1956. Newell had the role of language specifier-application programmer, Shaw was the system programmer and Simon took the role of application programmer-user.

The language includes features intended to support programs that could perform general problem solving, including lists, associations, schemas (frames), dynamic memory allocation, data types, recursion, associative retrieval, functions as arguments, generators (streams), and cooperative multitasking. IPL pioneered the concept of list processing, albeit in an assembly-language style.

3.1.1 A taste of IPL

An IPL computer has:

1.   A set of *symbols*. All symbols are addresses, and name cells. Unlike symbols in later languages, symbols consist of a character followed by a number, and are written H1, A29, 9-7, 9-100.

- Cell names beginning with a letter are *regional*, and are absolute addresses.
- Cell names beginning with "9-" are *local*, and are meaningful within the context of a single list. One list's 9-1 is independent of another list's 9-1.
- Other symbols (e.g., pure numbers) are *internal*.

2.   A set of *cells*. Lists are built from several cells holding mutual references. Cells have several fields:

- P, a 3-bit field used for an operation code when the cell is used as an instruction and unused when the cell is data.
- Q, a 3-valued field used for indirect reference when the cell is used as an instruction and unused when the cell is data.
- SYMB, a symbol used as the value in the cell.

3.   A set of *primitive processes*, which would be termed *primitive functions* in modern languages.

3.1.2 History of IPL

The first application of IPL was to demonstrate that the theorems in *Principia Mathematica* which were laboriously proven by hand, by Bertrand Russell and Alfred North Whitehead, could in fact be proven by computation. According to Simon's autobiography *Models of My Life*, this first application was developed first by hand simulation, using his children as the computing elements, while writing on and holding up note cards as the registers which contained the state variables of the program.

IPL was used to implement several early artificial intelligence programs, also by the same authors: the Logic Theory Machine (1956), the General Problem Solver (1957), and their computer chess program NSS (1958). Several versions of IPL were created: IPL-I (never implemented), IPL-II (1957 for JOHNNIAC), IPL-III (existed briefly), IPL-IV, IPL-V (1958, for IBM 650, IBM 704, IBM 7090, many others. Widely used), IPL-VI. However the language was soon displaced by Lisp, which had far more powerful features, a simpler syntax, and the benefit of automatic garbage collection.

3.2   Lisp Programming Language

Lisp (or LISP) is a family of computer programming languages with a long history and a distinctive, fully parenthesized syntax. Originally specified in 1958, Lisp is the second-oldest high-level programming language in widespread use today; only FORTRAN is older (by one year). Like FORTRAN, Lisp has changed a great deal since its early days, and a number of dialects have existed over its history. Today, the most widely known general-purpose Lisp dialects are Common Lisp, Scheme, and Clojure.

Lisp was originally created as a practical mathematical notation for computer programs, influenced by the notation of Alonzo Church's lambda calculus. It quickly became the favored programming language for artificial intelligence (AI) research. As one of the earliest programming languages, Lisp pioneered many ideas in computer science, including tree data structures, automatic storage management, dynamic typing, and the self-hosting compiler.

The name *LISP* derives from "LISt Processing". Linked lists are one of Lisp languages' major data structures, and Lisp source code is itself made up of lists. As a result, Lisp programs can manipulate source code as a data structure, giving rise to the macro systems that allow programmers to create new syntax or even new domain-specific languages embedded in Lisp.

The interchange ability of code and data also gives Lisp its instantly recognizable syntax. All program code is written as *s-expressions*, or parenthesized lists. A function call or syntactic form is written as a list with the function or operator's name first, and the arguments following; for instance, a function f that takes three arguments might be called using (f arg1 arg2 arg3).

3.2.1 History

Interest in artificial intelligence first surfaced in the mid 1950. Linguistics, psychology, and mathematics were only some areas of application for AI. Linguists were concerned with natural language processing, while psychologists were interested in modelling human information and retrieval. Mathematicians were more interested in automating the theorem proving process. The common need among all of these applications was a method to allow computers to process symbolic data in lists.

IBM was one of the first companies interested in AI in the 1950s. At the same time, the FORTRAN project was still going on. Because of the high cost associated with producing the first FORTRAN compiler, they decided to include the list processing functionality into FORTRAN. The FORTRAN List Processing Language (FLPL) was designed and implemented as an extension to FORTRAN.

In 1958 John McCarthy took a summer position at the IBM Information Research Department. He was hired to create a set of requirements for doing symbolic computation. The first attempt at this was differentiation of algebraic expressions. This initial experiment produced a list of language requirements, most notably was recursion and conditional expressions. At the time, not even FORTRAN (the only high-level language in existence) had these functions.

It was at the 1956 Dartmouth Summer Research Project on Artificial Intelligence that John McCarthy first developed the basics behind Lisp. His motivation was to develop a list processing language for Artificial Intelligence. By 1965 the primary dialect of Lisp was created (version 1.5). By 1970 special-purpose computers known as Lisp Machines, were designed to run Lisp programs. 1980 was the year that object-oriented concepts were integrated into the language. By 1986, the X3J13 group formed to produce a draft for ANSI Common Lisp standard. Finally in 1992, X3J13 group published the American National Standard for Common Lisp.

Since 2000

After having declined somewhat in the 1990s, Lisp has recently experienced a resurgence of interest. Most new activity is focused around open source implementations of Common Lisp, and includes the development of new portable libraries and applications. This interest can be measured partly by sales from the print version of *Practical Common Lisp* by Peter Seibel, a tutorial for new Lisp programmers published in

2004. It was briefly Amazon.com's second most popular programming book. It is available free online.http://en.wikipedia.org/wiki/Lisp_(programming_language) - cite_note-15

Many new Lisp programmers were inspired by writers such as Paul Graham and Eric S. Raymond to pursue a language others considered antiquated. New Lisp programmers often describe the language as an eye-opening experience and claim to be substantially more productive than in other languages. This increase in awareness may be contrasted to the "AI winter" and Lisp's brief gain in the mid-1990s.

Dan Weinreb lists in his survey of Common Lisp implementations eleven actively maintained Common Lisp implementations. Scieneer Common Lisp is a new commercial implementation forked from CMUCL with a first release in 2002.

The open source community has created new supporting infrastructure: Cliki is a wiki that collects Common Lisp related information, the Common Lisp directory lists resources, #lisp is a popular IRC channel (with support by a Lisp-written Bot), lisppaste supports the sharing and commenting of code snippets, Planet Lisp collects the contents of various Lisp-related Blogs, on LispForum user discuss Lisp topics, Lisp jobs is a service for announcing job offers and there is a new weekly news service (Weekly Lisp News). Common-lisp.net is a hosting site for open source Common Lisp projects.

50 years of Lisp (1958–2008) has been celebrated at LISP50@OOPSLA. There are several regular local user meetings (Boston, Vancouver, Hamburg,), Lisp Meetings (European Common Lisp Meeting, European Lisp Symposium) and an International Lisp Conference.

The Scheme community actively maintains over twenty implementations. Several significant new implementations (Chicken, Gambit, Gauche, Ikarus, Larceny, and Ypsilon) have been developed in the last few years. The Revised Report on the Algorithmic Language Scheme standard of Scheme was widely accepted in the Scheme community. The Scheme Requests for Implementation process has created a lot of quasi standard libraries and extensions for Scheme. User communities of individual Scheme implementations continue to grow. A new language standardization process was started in 2003 and led to the RRS Scheme standard in 2007. Academic use of Scheme for teaching computer science seems to have declined somewhat. Some universities are no longer using Scheme in their computer science introductory courses.

There are several new dialects of Lisp: Arc, Nu, and Clojure.

3.2.2 Connection to artificial intelligence

Since its inception, Lisp was closely connected with the artificial intelligence research community, especially on PDP-10http://en. wikipedia.org/wiki/Lisp_(programming_language)- cite_note-5 systems. Lisp was used as the implementation of the programming language Micro Planner which was used in the famous AI system SHRDLU. In the 1970s, as AI research spawned commercial offshoots, the performance of existing Lisp systems became a growing issue.

3.2.3 Areas of Application

Lisp totally dominated Artificial Intelligence applications for a quarter of a century, and is still the most widely used language for AI. In addition to its success in AI, Lisp pioneered the process of *Functional Programming*. Many programming language researchers believe that functional programming is a much better approach to software development, than the use of Imperative Languages (Pascal, C++, etc).

Below is a short list of the areas where Lisp has been used:

- Artificial Intelligence

- AI Robots
- Computer Games (Craps, Connect-4, BlackJack)
- Pattern Recognition

- Air Defense Systems
- Implementation of Real-Time, embedded Knowledge-Based Systems
- List Handling and Processing
- Tree Traversal (Breath/Depth First Search)
- Educational Purposes (Functional Style Programming)

3.2.4 Syntax and semantics

Symbolic expressions

Lisp is an expression-oriented language. Unlike most other languages, no distinction is made between "expressions" and "statements"; all code and data are written as expressions. When an expression is *evaluated*, it produces a value (in Common Lisp, possibly multiple values), which then can be embedded into other expressions. Each value can be any data type.

106

McCarthy's 1958 paper introduced two types of syntax: S-expressions (Symbolic expressions, also called "sexps"), which mirror the internal representation of code and data; and M-expressions (Meta Expressions), which express functions of S-expressions. M-expressions never found favour, and almost all Lisps today use S-expressions to manipulate both code and data.

The use of parentheses is Lisp's most immediately obvious difference from other programming language families. As a result, students have long given Lisp nicknames such as *Lost in Stupid Parentheses*, or *Lots of Irritating Superfluous Parentheses*.[23] However, the S-expression syntax is also responsible for much of Lisp's power: the syntax is extremely regular, which facilitates manipulation by computer. However, the syntax of Lisp is not limited to traditional parentheses notation. It can be extended to include alternative notations. XMLisp, for instance, is a Common Lisp extension that employs the metaobject-protocol to integrate S-expressions with the Extensible Markup Language (XML).

The reliance on expressions gives the language great flexibility. Because Lisp functions are themselves written as lists, they can be processed exactly like data. This allows easy writing of programs which manipulate other programs (metaprogramming). Many Lisp dialects exploit this feature using macro systems, which enables extension of the language almost without limit.

3.3   Prolog Programming Language

Prolog is a general purpose logic programming language associated with artificial intelligence and computational linguistics.

Prolog has its roots in first-order logic, a formal logic, and unlike many other programming languages, Prolog is declarative: the program logic is expressed in terms of relations, represented as facts and rules. A computation is initiated by running a *query* over these relations.

The language was first conceived by a group around Alain Colmerauer in Marseille, France, in the early 1970s and the first Prolog system was developed in 1972 by Colmerauer with Philippe Roussel.

Prolog was one of the first logic programming languages, and remains among the most popular such languages today, with many free and commercial implementations available. While initially aimed at natural language processing, the language has since then stretched far into other areas like theorem proving, expert systems, games, automated

answering systems, ontologies and sophisticated control systems. Modern Prolog environments support creating graphical user interfaces, as well as administrative and networked applications.

3.3.1 History of Prolog

The name *Prolog* was chosen by Philippe Roussel as an abbreviation for *programmation en logique* (French for *programming in logic*). It was created around 1972 by Alain Colmerauer with Philippe Roussel, based on Robert Kowalski's procedural interpretation of Horn clauses. It was motivated in part by the desire to reconcile the use of logic as a declarative knowledge representation language with the procedural representation of knowledge that was popular in North America in the late 1960s and early 1970s. According to Robert Kowalski, the first Prolog system was developed in 1972 by Alain Colmerauer and Phillipe Roussel. The first implementations of Prolog were interpreters; however, David H. D. Warren created the Warren Abstract Machine, an early and influential Prolog compiler which came to define the "Edinburgh Prolog" dialect which served as the basis for the syntax of most modern implementations.

Much of the modern development of Prolog came from the impetus of the Fifth Generation Computer Systems project (FGCS), which developed a variant of Prolog named *Kernel Language* for its first operating system.

Pure Prolog was originally restricted to the use of a resolution theorem prove with Horn clauses of the form:

3.3.2 Prolog Syntax and Semantics

In Prolog, program logic is expressed in terms of relations, and a computation is initiated by running a *query* over these relations. Relations and queries are constructed using Prolog's single data type, the *term*. Relations are defined by *clauses*. Given a query, the Prolog engine attempts to find a resolution refutation of the negated query. If the negated query can be refuted, i.e., an instantiation for all free variables is found that makes the union of clauses and the singleton set consisting of the negated query false, it follows that the original query, with the found instantiation applied, is a logical consequence of the program. This makes Prolog (and other logic programming languages) particularly useful for database, symbolic mathematics, and language parsing applications. Because Prolog allows impure predicates, checking the truth value of certain special predicates may have some deliberate side effect, such as printing a value to the screen. Because of this, the

programmer is permitted to use some amount of conventional imperative programming when the logical paradigm is inconvenient. It has a purely logical subset, called "pure Prolog", as well as a number of extra logical features.

3.3.2.1 Data Types

Prolog's single data type is the *term*. Terms are atoms, *numbers*, *variables* or *compound terms*.

- An atom is a general-purpose name with no inherent meaning. Examples of atoms include x, blue, 'Taco', and 'some atom'.
- Numbers can be floats or integers.
- Variables are denoted by a string consisting of letters, numbers and underscore characters, and beginning with an upper-case letter or underscore. Variables closely resemble variables in logic in that they are placeholders for arbitrary terms.
- A compound term is composed of an atom called a "functor" and a number of "arguments", which are again terms. Compound terms are ordinarily written as a functor followed by a comma-separated list of argument terms, which is contained in parentheses. The number of arguments is called the term's arity. An atom can be regarded as a compound term with arity zero. Examples of compound terms are truck_year('Mazda', 1986) and 'Person_Friends'(zelda,[tom,jim]).

Special cases of compound terms:

- A *List* is an ordered collection of terms. It is denoted by square brackets with the terms separated by commas or in the case of the empty list, []. For example [1,2,3] or [red,green,blue].
- *Strings*: A sequence of characters surrounded by quotes is equivalent to a list of (numeric) character codes, generally in the local character encoding, or Unicode if the system supports Unicode. For example, "to be, or not to be".

3.3.2.2 Rules and Facts

Prolog programs describe relations, defined by means of clauses. Pure Prolog is restricted to Horn clauses. There are two types of clauses: facts and rules. A rule is of the form

Head: - Body.

and is read as "Head is true if Body is true". A rule's body consists of calls to predicates, which are called the rule's goals. The built-in predicate ,/2 (meaning a 2-arity operator with name ,) denotes conjunction of goals, and ;/2 denotes disjunction. Conjunctions and disjunctions can only appear in the body, not in the head of a rule.

Clauses with empty bodies are called facts. An example of a fact is:
cat(tom).
which is equivalent to the rule?
cat(tom) :- true.
The built-in predicate true/0 is always true.
Given the above fact, one can ask:
is tom a cat?
?- cat(tom).
Yes
what things are cats?
?- cat(X).
X = tom
Clauses with bodies are called rules. An example of a rule is:
animal(X):- cat(X).
If we add that rule and ask *what things are animals?*
?- animal(X).
X = tom

Due to the relational nature of many built-in predicates, they can typically be used in several directions. For example, length/2 can be used to determine the length of a list (length(List, L), given a list) as well as to generate a list skeleton of a given length (length(X, 5)), and also to generate both list skeletons and their lengths together (length(X, L)). Similarly, append/3 can be used both to append two lists (append(ListA, ListB, X) given lists ListA and ListB) as well as to split a given list into parts (append(X, Y, List), given a list List). For this reason, a comparatively small set of library predicates suffices for many Prolog programs.

As a general purpose language, Prolog also provides various built-in predicates to perform routine activities like input/output, using graphics and otherwise communicating with the operating system. These predicates are not given a relational meaning and are only useful for the side-effects they exhibit on the system. For example, the predicate write/1 displays a term on the screen.

110

3.3.2.3 Evaluation

Execution of a Prolog program is initiated by the user's posting of a single goal, called the query. Logically, the Prolog engine tries to find a resolution refutation of the negated query. The resolution method used by Prolog is called SLD resolution. If the negated query can be refuted, it follows that the query, with the appropriate variable bindings in place, is a logical consequence of the program. In that case, all generated variable bindings are reported to the user, and the query is said to have succeeded. Operationally, Prolog's execution strategy can be thought of as a generalization of function calls in other languages, one difference being that multiple clause heads can match a given call. In that case, the system creates a choice-point, unifies the goal with the clause head of the first alternative, and continues with the goals of that first alternative. If any goal fails in the course of executing the program, all variable bindings that were made since the most recent choice-point was created are undone, and execution continues with the next alternative of that choice-point. This execution strategy is called chronological backtracking. For example:

mother_child(trude, sally).
father_child(tom, sally).
father_child(tom, erica).
father_child(mike, tom).

sibling(X, Y)   :- parent_child(Z, X), parent_child(Z, Y).

parent_child(X, Y) :- father_child(X, Y).
parent_child(X, Y) :- mother_child(X, Y).
This results in the following query being evaluated as true:
?- sibling(sally, erica).
Yes

This is obtained as follows: Initially, the only matching clause-head for the query sibling(sally, erica) is the first one, so proving the query is equivalent to proving the body of that clause with the appropriate variable bindings in place, i.e., the conjunction (parent_child(Z,sally), parent_child(Z,erica)). The next goal to be proved is the leftmost one of this conjunction, i.e., parent_child(Z, sally). Two clause heads match this goal. The system creates a choice-point and tries the first alternative, whose body is father_child(Z, sally). This goal can be proved using the fact father_child(tom, sally), so the binding Z = tom is generated, and the next goal to be proved is the second part of the above conjunction: parent_child(tom, erica). Again, this can be proved by the corresponding fact. Since all goals could be proved, the query succeeds.

Since the query contained no variables, no bindings are reported to the user. A query with variables, like:

?- father_child(Father, Child).
enumerates all valid answers on backtracking.
Notice that with the code as stated above, the query?- sibling(sally, sally). Also succeeds. One would insert additional goals to describe the relevant restrictions, if desired.

3.3.2.4 Loops and recursion

Iterative algorithms can be implemented by means of recursive predicates.

3.3.2.5 Negation

The built-in Prolog predicate \+/1 provides negation as failure, which allows for non-monotonic reasoning. The goal \+ legal(X) in the rule
illegal(X) :- \+ legal(X).

is evaluated as follows: Prolog attempts to prove the legal(X). If a proof for that goal can be found, the original goal (i.e., \+ legal(X)) fails. If no proof can be found, the original goal succeeds. Therefore, the \+/1 prefix operator is called the "not provable" operator, since the query ?- \+ Goal. succeeds if Goal is not provable. This kind of negation is sound if its argument is "ground" (i.e. contains no variables). Soundness is lost if the argument contains variables and the proof procedure is complete. In particular, the query ?- illegal(X). can now not be used to enumerate all things that are illegal.

3.3.2.6 Examples

Here follow some example programs written in Prolog.
 Hello world
An example of a query:
?- write('Hello world!'), nl.
Hello world!
true.

?-

3.3.2.7 Criticism

Although Prolog is widely used in research and education, Prolog and other logic programming languages have not had a significant impact on

the computer industry in general. Most applications are small by industrial standards, with few exceeding 100,000 lines of code. Programming in the large is considered to be complicated because not all Prolog compilers support modules, and there are compatibility problems between the module systems of the major Prolog compilers. Portability of Prolog code across implementations has also been a problem, but developments since 2007 have meant: "the portability within the family of Edinburgh/Quintus derived Prolog implementations is good enough to allow for maintaining portable real-world applications."

Software developed in Prolog has been criticised for having a high performance penalty compared to conventional programming languages. However, advances in implementation methods have reduced the penalties to as little as 25%-50% for some applications.

3.3.2.8 Types

Prolog is an untyped language. Attempts to introduce types date back to the 1980s, and as of 2008 there are still attempts to extend Prolog with types. Type information is useful not only for type safety but also for reasoning about Prolog programs.

4.0   CONCLUSION

IPL, Lisp and Prolog considered in this unit are among other specialized programming languages for artificial intelligence.

5.0   SUMMARY

In this unit, you learnt that:

- IPL is the pi1ered concept of list processing.
- Lisp is the second-oldest high-level programming language in widespread use today
- Prolog was 1 of the first logic programming languages, and remains among the most popular such languages today, with many free and commercial implementations available.

6.0   TUTOR-MARKED ASSIGNMENT

1.   Describe Prolog programming Language.
2.   Describe Lisp programming Language.
3.   List three (3) areas where Lisp can be used.

7.0   REFERENCES/FURTHER READING

Crevier, D. (1993). *AI: The Tumultuous Search for Artificial Intelligence*. New York, NY: BasicBooks, ISBN 0-465-02997-3

McCarthy, J. (1979). *History of Lisp.* "LISP prehistory - Summer 1956 through Summer 1958."

Nilsson, N. (1998). *Artificial Intelligence: A New Synthesis*. Morgan: Kaufmann Publishers, ISBN 978-1-55860-467-4.

Poole, D. Mackworth, A. Goebel, R. (1998). *Computational Intelligence: A Logical Approach*. New York: Oxford University Press, ISBN 0195102703, http://www.cs.ubc.ca/spider/poole/ci.html

Russell, S. J.& Norvig, Peter (2003). *Artificial Intelligence: A Modern Approach* (2nd ed.), Upper Saddle River, New Jersey: Prentice Hall, ISBN 0-13-790395-2, http://aima.cs.berkeley.edu/

Sebesta, R.W. (1996). *Concepts of Programming Languages,* (Third Edition). Menlo Park, California: Addison-Wesley Publishing Company.

UNIT 3   NATURAL LANGUAGE PROCESSING

CONTENTS

1.0   INTRODUCTION

Natural language processing (NLP) is a field of computer science and linguistics concerned with the interactions between computers and human (natural) languages; it began as a branch of artificial intelligence. In theory, natural language processing is a very attractive method of human–computer interaction. Natural language understanding is sometimes referred to as an AI-complete problem because it seems to require extensive knowledge about the outside world and the ability to manipulate it.

An automated online assistant providing customer service on a web page is an example of an application where natural language processing is a major comp1nt.

2.0   OBJECTIVES

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

- describe the history of natural language processing
- list major tasks in NLP
- mention different types of evaluation of NPL.

3.0   MAIN CONTENT

3.1   History of natural language processing

The history of NLP generally starts in the 1950s, although work can be found from earlier periods. In 1950, Alan Turing published his famous article "Computing Machinery and Intelligence" which proposed what is now called the Turing test as a criterion of intelligence. This criterion depends on the ability of a computer program to impersonate a human in a real-time written conversation with a human judge, sufficiently well that the judge is unable to distinguish reliably — on the basis of the conversational content alone — between the program and a real human. The Georgetown experiment in 1954 involved fully automatic translation of more than sixty Russian sentences into English. The authors claimed that within three or five years, machine translation would be a solved problem. However, real progress was much slower, and after the ALPAC report in 1966, which found that ten years long research had failed to fulfill the expectations, funding for machine translation was dramatically reduced. Little further research in machine translation was conducted until the late 1980s, when the first statistical machine translation systems were developed.

Some notably successful NLP systems developed in the 1960s were SHRDLU, a natural language system working in restricted "blocks worlds" with restricted vocabularies, and ELIZA, a simulation of a Rogerian psychotherapist, written by Joseph Weizenbaum between 1964 to 1966. Using almost no information about human thought or emotion, ELIZA sometimes provided a startlingly human-like interaction. When the "patient" exceeded the very small knowledge base, ELIZA might provide a generic response, for example, responding to "My head hurts" with "Why do you say your head hurts?"

During the 70's many programmers began to write 'conceptual ontologies', which structured real-world information into computer-understandable data. Examples are MARGIE (Schank, 1975), SAM (Cullingford, 1978), PAM (Wilensky, 1978), TaleSpin (Meehan, 1976), QUALM (Lehnert, 1977), Politics (Carbonell, 1979), and Plot Units (Lehnert 1981). During this time, many chatterbots were written including PARRY, Racter, and Jabberwacky.

Up to the 1980s, most NLP systems were based on complex sets of hand-written rules. Starting in the late 1980s, however, there was a revolution in NLP with the introduction of machine learning algorithms for language processing. This was due both to the steady increase in computational power resulting from Moore's Law and the gradual

lessening of the dominance of Chomskyan theories of linguistics (e.g. transformational grammar), whose theoretical underpinnings discouraged the sort of corpus linguistics that underlies the machine-learning approach to language processing. Some of the earliest-used machine learning algorithms, such as decision trees, produced systems of hard if-then rules similar to existing hand-written rules. Increasingly, however, research has focused on statistical models, which make soft, probabilistic decisions based on attaching real-valued weights to the features making up the input data. Such models are generally more robust when given unfamiliar input, especially input that contains errors (as is very common for real-world data), and produce more reliable results when integrated into a larger system comprising multiple subtasks.

Many of the notable early successes occurred in the field of machine translation, due especially to work at IBM Research, where successively more complicated statistical models were developed. These systems were able to take advantage of existing multilingual textual corpora that had been produced by the Parliament of Canada and the European Union as a result of laws calling for the translation of all governmental proceedings into all official languages of the corresponding systems of government. However, most other systems depended on corpora specifically developed for the tasks implemented by these systems, which was (and often continues to be) a major limitation in the success of these systems. As a result, a great deal of research has gone into methods of more effectively learning from limited amounts of data.

Recent research has increasingly focused on unsupervised and semi-supervised learning algorithms. Such algorithms are able to learn from data that has not been hand-annotated with the desired answers, or using a combination of annotated and non-annotated data. Generally, this task is much more difficult than supervised learning, and typically produces less accurate results for a given amount of input data. However, there is an enormous amount of non-annotated data available (including, among other things, the entire content of the World Wide Web), which can often make up for the inferior results.

3.2   NLP Using Machine Learning

As described above, modern approaches to natural language processing (NLP) are grounded in machine learning. The paradigm of machine learning is different from that of most prior attempts at language processing. Prior implementations of language-processing tasks typically involved the direct hand coding of large sets of rules. The machine-learning paradigm calls instead for using general learning

algorithms — often, although not always, grounded in statistical inference — to automatically learn such rules through the analysis of large corpora of typical real-world examples. A corpus (plural, "corpora") is a set of documents (or sometimes, individual sentences) that have been hand-annotated with the correct values to be learned.

As an example, consider the task of part of speech tagging, i.e. determining the correct part of speech of each word in a given sentence, typically one that has never been seen before. A typical machine-learning-based implementation of a part of speech tagger proceeds in two steps, a training step and an evaluation step. The first step — the training step — makes use of a corpus of training data, which consists of a large number of sentences, each of which has the correct part of speech attached to each word. (An example of such a corpus in common use is the Penn Treebank. This includes (among other things) a set of 500 texts from the Brown Corpus, containing examples of various genres of text, and 2500 articles from the Wall Street Journal.) This corpus is analyzed and a learning model is generated from it, consisting of automatically created rules for determining the part of speech for a word in a sentence, typically based on the nature of the word in question, the nature of surrounding words, and the most likely part of speech for those surrounding words. The model that is generated is typically the best model that can be found that simultaneously meets two conflicting objectives: To perform as well as possible on the training data, and to be as simple as possible (so that the model avoids over fitting the training data, i.e. so that it generalizes as well as possible to new data rather than only succeeding on sentences that have already been seen). In the second step (the evaluation step), the model that has been learned is used to process new sentences. An important part of the development of any learning algorithm is testing the model that has been learned on new, previously unseen data. It is critical that the data used for testing is not the same as the data used for training; otherwise, the testing accuracy will be unrealistically high.

Many different classes of machine learning algorithms have been applied to NLP tasks. In common to all of these algorithms is that they take as input a large set of "features" that are generated from the input data. As an example, for a part-of-speech tagger, typical features might be the identity of the word being processed, the identity of the words immediately to the left and right, the part-of-speech tag of the word to the left, and whether the word being considered or its immediate neighbors are content words or function words. The algorithms differ, however, in the nature of the rules generated. Some of the earliest-used algorithms, such as decision trees, produced systems of hard if-then rules similar to the systems of hand-written rules that were then

common. Increasingly, however, research has focused on statistical models, which make soft, probabilistic decisions based on attachingreal-valued weights to each input feature. Such models have the advantage that they can express the relative certainty of many different possible answers rather than only one, producing more reliable results when such a model is included as a component of a larger system. In addition, models that make soft decisions are generally more robust when given unfamiliar input, especially input that contains errors (as is very common for real-world data).

Systems based on machine-learning algorithms have many advantages over hand-produced rules:

- The learning procedures used during machine learning automatically focus on the most common cases, whereas when writing rules by hand it is often not obvious at all where the effort should be directed.

- Automatic learning procedures can make use of statistical inference algorithms to produce models that are robust to unfamiliar input (e.g. containing words or structures that have not been seen before) and to erroneous input (e.g. with misspelled words or words accidentally omitted). Generally, handling such input gracefully with hand-written rules — or more generally, creating systems of hand-written rules that make soft decisions — is extremely difficult and error-prone.

- Systems based on automatically learning the rules can be made more accurate simply by supplying more input data. However, systems based on hand-written rules can only be made more accurate by increasing the complexity of the rules, which is a much more difficult task. In particular, there is a limit to the complexity of systems based on hand-crafted rules, beyond which the systems become more and more unmanageable. However, creating more data to input to machine-learning systems simply requires a corresponding increase in the number of man-hours worked, generally without significant increases in the complexity of the annotation process.

3.3   Major tasks in NLP

The following is a list of some of the most commonly researched tasks in NLP. Note that some of these tasks have direct real-world applications, while others more commonly serve as subtasks that are used to aid in solving larger tasks. What distinguishes these tasks from other potential and actual NLP tasks is not only the volume of research devoted to them but the fact that for each one there is typically a well-

defined problem setting, a standard metric for evaluating the task, standard corpora on which the task can be evaluated, and competitions devoted to the specific task.

- **Automatic summarization:** Produce a readable summary of a chunk of text. Often used to provide summaries of text of a known type, such as articles in the financial section of a newspaper.

- **Co reference resolution:** Given a sentence or larger chunk of text, determine which words ("mentions") refer to the same objects ("entities"). Anaphora resolution is a specific example of this task, and is specifically concerned with matching up pronouns with the nouns or names that they refer to. The more general task of co reference resolution also includes identify so-called "bridging relationships" involving referring expressions. For example, in a sentence such as "He entered John's house through the front door", "the front door" is a referring expression and the bridging relationship to be identified is the fact that the door being referred to is the front door of John's house (rather than of some other structure that might also be referred to).

- **Discourse analysis:** This rubric includes a number of related tasks. One task is identifying the discourse structure of connected text, i.e. the nature of the discourse relationships between sentences (e.g. elaboration, explanation, contrast). Another possible task is recognizing and classifying the speech acts in a chunk of text (e.g. yes-no question, content question, statement, assertion, etc.).

- **Machine translation:** Automatically translate text from one human language to another. This is one of the most difficult problems, and is a member of a class of problems colloquially termed "AI-complete", i.e. requiring all of the different types of knowledge that humans possess (grammar, semantics, facts about the real world, etc.) in order to solve properly.

- **Morphological segmentation:** Separate words into individual morphemes and identify the class of the morphemes. The difficulty of this task depends greatly on the complexity of the morphology (i.e. the structure of words) of the language being considered. English has fairly simple morphology, especially inflectional morphology, and thus it is often possible to ignore this task entirely and simply model all possible forms of a word (e.g. "open, opens, opened, and opening") as separate words. In languages such as Turkish, however, such an approach is not possible, as each dictionary entry has thousands of possible word forms.

- **Named entity recognition (NER):** Given a stream of text, determine which items in the text map to proper names, such as people or places, and what the type of each such name is (e.g. person, location, organization). Note that, although capitalization can aid in recognizing named entities in languages such as English, this information cannot aid in determining the type of named entity, and in any case is often inaccurate or insufficient. For example, the first word of a sentence is also capitalized, and named entities often span several words, only some of which are capitalized. Furthermore, many other languages in non-Western scripts (e.g. Chinese or Arabic) do not have any capitalization at all, and even languages with capitalization may not consistently use it to distinguish names. For example, German capitalizes all nouns, regardless of whether they refer to names, and French and Spanish do not capitalize names that serve as adjectives.
- **Natural language generation:** Convert information from computer databases into readable human language.
- **Natural language understanding:** Convert chunks of text into more formal representations such as first-order logic structures that are easier for computer programs to manipulate. Natural language understanding involves the identification of the intended semantic from the multiple possible semantics which can be derived from a natural language expression which usually takes the form of organized notations of natural languages concepts. Introduction and creation of language metamodel and ontology are efficient however empirical solutions. An explicit formalization of natural languages semantics without confusions with implicit assumptions such as closed world assumption (CWA) vs. open world assumption, or subjective Yes/No vs. objective True/False is expected for the construction of a basis of semantics formalization.
- **Optical character recognition (OCR):** Given an image representing printed text, determine the corresponding text.
- **Part-of-speech tagging:** Given a sentence, determine the part of speech for each word. Many words, especially common ones, can serve as multiple parts of speech. For example, "book" can be a noun ("the book on the table") or verb ("to book a flight"); "set" can be a noun, verb or adjective; and "out" can be any of at least five different parts of speech. Note that some languages have more such ambiguity than others. Languages with little inflectional morphology, such as English are particularly prone to such ambiguity. Chinese is prone to such ambiguity because it is a tonal language during verbalization. Such inflection is not readily conveyed via the entities employed within the orthography to convey intended meaning.

- **Parsing:** Determine the parse tree (grammatical analysis) of a given sentence. The grammar for natural languages is ambiguous and typical sentences have multiple possible analyses. In fact, perhaps surprisingly, for a typical sentence there may be thousands of potential parses (most of which will seem completely nonsensical to a human).

- **Question answering:** Given a human-language question, determine its answer. Typical questions have a specific right answer (such as "What is the capital of Canada?"), but sometimes open-ended questions are also considered (such as "What is the meaning of life?").

- **Relationship extraction:** Given a chunk of text, identify the relationships among named entities (e.g. who is the wife of whom).

- **Sentence breaking (also known as sentence boundary disambiguation):** Given a chunk of text, find the sentence boundaries. Sentence boundaries are often marked by periods or other punctuation marks, but these same characters can serve other purposes (e.g. marking abbreviations).

- **Sentiment analysis:** Extract subjective information usually from a set of documents, often using online reviews to determine "polarity" about specific objects. It is especially useful for identifying trends of public opinion in the social media, for the purpose of marketing.

- **Speech recognition:** Given a sound clip of a person or people speaking, determine the textual representation of the speech. This is the opposite of text to speech and is one of the extremely difficult problems colloquially termed "AI-complete" (see above). In natural speech there are hardly any pauses between successive words, and thus speech segmentation is a necessary subtask of speech recognition (see below). Note also that in most spoken languages, the sounds representing successive letters blend into each other in a process termed coarticulation, so the conversion of the analog signal to discrete characters can be a very difficult process.

- **Speech segmentation:** Given a sound clip of a person or people speaking, separate it into words. A subtask of speech recognition and typically grouped with it.

- Topic segmentation and recognition: Given a chunk of text, separate it into segments each of which is devoted to a topic, and identify the topic of the segment.

- **Word segmentation:** Separate a chunk of continuous text into separate words. For a language like English, this is fairly trivial, since words are usually separated by spaces. However, some written languages like Chinese, Japanese and Thai do not mark

word boundaries in such a fashion, and in those languages text segmentation is a significant task requiring knowledge of the vocabulary and morphology of words in the language.

- **Word sense disambiguation:** Many words have more than one meaning; we have to select the meaning which makes the most sense in context. For this problem, we are typically given a list of words and associated word senses, e.g. from a dictionary or from an online resource such as WordNet.

In some cases, sets of related tasks are grouped into subfields of NLP that are often considered separately from NLP as a whole. Examples include:

- **Information retrieval (IR):** This is concerned with storing, searching and retrieving information. It is a separate field within computer science (closer to databases), but IR relies on some NLP methods (for example, stemming). Some current research and applications seek to bridge the gap between IR and NLP.
- **Information extraction (IE):** This is concerned in general with the extraction of semantic information from text. This covers tasks such as named entity recognition, coreference resolution, relationship extraction, etc.
- **Speech processing:** This covers speech recognition, text-to-speech and related tasks.

Other tasks include:

- Stemming
- Text simplification
- Text-to-speech
- Text-proofing
- Natural language search
- Query expansion
- Truecasing

3.4   Statistical Natural Language Processing

Statistical natural-language processing uses stochastic, probabilistic and statistical methods to resolve some of the difficulties discussed above, especially those which arise because longer sentences are highly ambiguous when processed with realistic grammars, yielding thousands or millions of possible analyses. Methods for disambiguation often involve the use of corpora and Markov models. Statistical NLP comprises all quantitative approaches to automated language processing,

including probabilistic modeling, information theory, and linear algebra. The technology for statistical NLP comes mainly from machine learning and data mining, both of which are fields of artificial intelligence that involve learning from data.

3.5   Evaluation of natural language processing

3.5.1 Objectives

The goal of NLP evaluation is to measure one or more qualities of an algorithm or a system, in order to determine whether (or to what extent) the system answers the goals of its designers, or meets the needs of its users. Research in NLP evaluation has received considerable attention, because the definition of proper evaluation criteria is one way to specify precisely an NLP problem, going thus beyond the vagueness of tasks defined only as language understanding or language generation. A precise set of evaluation criteria, which includes mainly evaluation data and evaluation metrics, enables several teams to compare their solutions to a given NLP problem.

3.5.2 Short history of evaluation in NLP

The first evaluation campaign on written texts seems to be a campaign dedicated to message understanding in 1987 (Pallet 1998). Then, the Parseval/GEIG project compared phrase-structure grammars (Black 1991). A series of campaigns within Tipster project were realized on tasks like summarization, translation and searching (Hirschman 1998). In 1994, in Germany, the Morpholympics compared German taggers. Then, the Senseval and Romanseval campaigns were conducted with the objectives of semantic disambiguation. In 1996, the Sparkle campaign compared syntactic parsers in four different languages (English, French, German and Italian). In France, the Grace project compared a set of 21 taggers for French in 1997 (Adda 1999). In 2004, during the Technolangue/Easy project, 13 parsers for French were compared. Large-scale evaluation of dependency parsers were performed in the context of the CoNLL shared tasks in 2006 and 2007. In Italy, the EVALITA campaign was conducted in 2007 and 2009 to compare various NLP and speech tools for Italian; the 2011 campaign is in full progress - EVALITA web site. In France, within the ANR-Passage project (end of 2007), 10 parsers for French were compared - passage web site.

3.5.3 Different types of evaluation

Depending on the evaluation procedures, a number of distinctions are traditionally made in NLP evaluation:

- Intrinsic vs. extrinsic evaluation

Intrinsic evaluation considers an isolated NLP system and characterizes its performance mainly with respect to a gold standard result, pre-defined by the evaluators. Extrinsic evaluation, also called evaluation in use considers the NLP system in a more complex setting, either as an embedded system or serving a precise function for a human user. The extrinsic performance of the system is then characterized in terms of its utility with respect to the overall task of the complex system or the human user. For example, consider a syntactic parser that is based on the output of some new part of speech (POS) tagger. An intrinsic evaluation would run the POS tagger on some labelled data, and compare the system output of the POS tagger to the gold standard (correct) output. An extrinsic evaluation would run the parser with some other POS tagger, and then with the new POS tagger, and compare the parsing accuracy.

- Black-box vs. glass-box evaluation

Black-box evaluation requires one to run an NLP system on a given data set and to measure a number of parameters related to the quality of the process (speed, reliability, resource consumption) and, most importantly, to the quality of the result (e.g. the accuracy of data annotation or the fidelity of a translation). Glass-box evaluation looks at the design of the system, the algorithms that are implemented, the linguistic resources it uses (e.g. vocabulary size), etc. Given the complexity of NLP problems, it is often difficult to predict performance only on the basis of glass-box evaluation, but this type of evaluation is more informative with respect to error analysis or future developments of a system.

- Automatic vs. manual evaluation

In many cases, automatic procedures can be defined to evaluate an NLP system by comparing its output with the gold standard (or desired) one. Although the cost of producing the gold standard can be quite high, automatic evaluation can be repeated as often as needed without much additional costs (on the same input data). However, for many NLP problems, the definition of a gold standard is a complex task, and can prove impossible when inter-annotator agreement is insufficient. Manual

evaluation is performed by human judges, which are instructed to estimate the quality of a system, or most often of a sample of its output, based on a number of criteria. Although, thanks to their linguistic competence, human judges can be considered as the reference for a number of language processing tasks, there is also considerable variation across their ratings. This is why automatic evaluation is sometimes referred to as objective evaluation, while the human kind appears to be more subjective.

3.5.4 Shared tasks (Campaigns)

- BioCreative
- Message Understanding Conference
- Technolangue/Easy
- Text Retrieval Conference
- Evaluation exercises on Semantic Evaluation (SemEval)

4.0   CONCLUSION

Systems based on machine-learning algorithms have many advantages over hand-produced rules.

5.0 SUMMARY

In this unit, you learnt:

- NLP using machine learning
- History of natural language processing
- Major tasks in NLP
- Statistical Natural Language Processing
- Evaluation of natural language processing

6.0   TUTOR- MARKED ASSIGNMENT

1.   List four major tasks in NLP.
2.   Describe the history of natural language processing.
3.   Mention different types of evaluation of NPL.

7.0 REFERENCES/FURTHER READING

Bates, M. (1995). Models of Natural Language Understanding. Proceedings of the National Academy of Sciences of the United States of America, Vol. 92, No. 22 (Oct. 24, 1995), pp. 9977–9982.

Christopher, D. Manning, Hinrich Schütze (1999). Foundations of Statistical Natural Language Processing, MIT Press, ISBN 978-0-262-13360-9, p. xxxi.

Yucong, D. & Christophe C. (2011). Formalizing Semantic of Natural Language through Conceptualization from Existence. International Journal of Innovation, Management and Technology (2011) 2 (1), pp. 37-42.

MODULE 4   ARTIFICIAL INTELLIGENCE AND ITS APPLICATIONS

Unit 1   Expert System
Unit 2   Robotics

UNIT 1   EXPERT SYSTEM

CONTENTS

1.0   INTRODUCTION

Expert system is a computer program that uses artificial intelligence to solve problems within a specialized domain that ordinarily requires human expertise. The first expert system was developed in 1965 by Edward Feigenbaum and Joshua Lederberg of Stanford University in California, U.S. Dendral, as their expert system was later known, was

designed to analyze chemical compounds. Expert systems now have commercial applications in fields as diverse as medical diagnosis, petroleum engineering, financial investing make financial forecasts and schedule routes for delivery vehicles.

2.0   OBJECTIVES

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

- explain an expert system
- distinction between expert systems and traditional problem solving programs
- explain the term "knowledge base".

3.0   MAIN CONTENT

3.1   What is an Expert System?

It is a computer application that performs a task that would otherwise be performed by a human expert. Some expert systems are designed to take the place of human experts, while others are designed to aid them.

To design an expert system, one needs a *knowledge engineer,* an individual who studies how human experts make decisions and translates the rules into terms that a computer can understand. In order to accomplish feats of apparent intelligence, an expert system relies on two components: a knowledge base and an inference engine.

3.1.1 Comparison to problem-solving systems

The principal distinction between expert systems and traditional problem solving.

Programs are the way in which the problem related expertise is coded. In traditional applications, problem-related expertise is encoded in both program and data structures. In the expert system approach all of the problem expertise is encoded mostly in data structures.

In an example related to tax advice, the traditional approach has data structures that describe the taxpayer and tax tables, and a program that contains rules (encoding expert knowledge) that relate information about the taxpayer to tax table choices. In contrast, in the expert system approach, the latter information is also encoded in data structures. The collective data structures are called the knowledge base . The program (inference engine) of an expert system is relatively independent of the

problem domain (taxes) and processes the rules without regard to the problem area they describe.

This organization has several benefits:

- New rules can be added to the knowledge base or altered without needing to rebuild the program. This allows changes to be made rapidly to a system (e.g., after it has been shipped to its customers, to accommodate very recent changes in state or federal tax codes).
- Rules are arguably easier for (non-programmer) domain experts to create and modify than writing code. Commercial rule engines typically come with editors that allow rule creation/modification through a graphical user interface, which also performs actions such as consistency and redundancy checks.

Modern rule engines allow a hybrid approach: some allow rules to be "compiled" into a form that is more efficiently machine-executable. Also, for efficiency concerns, rule engines allow rules to be defined more expressively and concisely by allowing software developers to create functions in a traditional programming language such as Java, which can then be invoked from either the condition or the action of a rule. Such functions may incorporate domain-specific (but reusable) logic.

3.2   Knowledge Base

A knowledge base (abbreviated KB, kb or Δ) is a special kind of database for knowledge management, providing the means for the computerized collection, organization, and retrieval of knowledge. Also, it is a collection of data representing related experiences which their results is related to their problems and solutions.

Facts for a knowledge base must be acquired from human experts through interviews and observations. This knowledge is then usually represented in the form of "if-then" rules (production rules): "If some condition is true then the following inference can be made (or some action taken)." The knowledge base of a major expert system includes thousands of rules. A probability factor is often attached to the conclusion of each production rule, because the conclusion is not a certainty. For example, a system for the diagnosis of eye diseases might indicate, based on information supplied to it, a 90 percent probability that a person has glaucoma, and it might also list conclusions with lower probabilities. An expert system may display the sequence of rules through which it arrived at its conclusion; tracing this flow helps the

user to appraise the credibility of its recommendation and is useful as a learning tool for students.

Human experts frequently employ heuristic rules, or "rules of thumb," in addition to simple production rules. For example, a credit manager might know that an applicant with a poor credit history, but a clean record since acquiring a new job, might actually be a good credit risk. Expert systems have incorporated such heuristic rules and increasingly have the ability to learn from experience. Nevertheless, expert systems remain supplements, rather than replacements, for human experts.

3.2.1 Types Of Knowledge Base

Knowledge bases are essentially closed or open information repositories and can be categorized under two main headings:

- Machine-readable knowledge bases store knowledge in a computer-readable form, usually for the purpose of having automated deductive reasoning applied to them. They contain a set of data, often in the form of rules that describe the knowledge in a logically consistent manner. An ontology can define the structure of stored data - what types of entities are recorded and what their relationships are. Logical operators, such as *And* (conjunction), *Or* (disjunction), *material implication* and *negation* may be used to build it up from simpler pieces of information. Consequently, classical deduction can be used to reason about the knowledge in the knowledge base. Some machine-readable knowledge bases are used with artificial intelligence, for example as part of an expert system that focuses on a domain like prescription drugs or customs law. Such knowledge bases are also used by the semantic web.
- Human-readable knowledge bases are designed to allow people to retrieve and use the knowledge they contain. They are commonly used to complement a help desk or for sharing information among employees within an organization. They might store troubleshooting information, articles, white papers, user manuals, knowledge tags, or answers to frequently asked questions. Typically, a search engine is used to locate information in the system, or users may browse through a classification scheme.

A text based system that can include groups of documents including hyperlinks between them is known as Hypertext Systems. Hypertext systems support the decision process by relieving the user of the significant effort it takes to relate and remember things." Knowledge

bases can exist on both computers and mobile phones in a hypertext format. Knowledge base analysis and design (also known as KBAD) is an approach that allows people to conduct analysis and design in a way that result in a knowledge base, which can later be used to make informative decisions. This approach was first implemented by Dr. Steven H. Dam

3.3   Inference Engine

In computer science, and specifically the branches of knowledge engineering and artificial intelligence, an inference engine is a computer program that tries to derive answers from a knowledge base. It is the "brain" that expert systems use to reason about the information in the knowledge base for the ultimate purpose of formulating new conclusions. Inference engines are considered to be a special case of reasoning engines, which can use more general methods of reasoning.

3.3.1 Architecture

The separation of inference engines as a distinct software component stems from the typical production system architecture. This architecture relies on a data store:

1.   An interpreter. The interpreter executes the chosen agenda items by applying the corresponding base rules.
2.   A scheduler. The scheduler maintains control over the agenda by estimating the effects of applying inference rules in light of item priorities or other criteria on the agenda.
3.   A consistency enforcer. The consistency enforcer attempts to maintain a consistent representation of the emerging solution

3.3.2 The Recognize-Act Cycle

The inference engine can be described as a form of finite state machine with a cycle consisting of three action states: *match rules*, *select rules*, and *execute rules*. Rules are represented in the system by a notation called predicate logic.

In the first state, match rules, the inference engine finds all of the rules that are satisfied by the current contents of the data store. When rules are in the typical *condition-action* form, this means testing the conditions against the working memory. The rule matching that are found are all candidates for execution: they are collectively referred to as the *conflict set*. Note that the same rule may appear several times in the conflict set

if it matches different subsets of data items. The pair of a rule and a subset of matching data items are called an *instantiation* of the rule.
In many applications, where large volumes of data are concerned and/or when performance time considerations are critical, the computation of the conflict set is a non-trivial problem.

3.3.3 Data-Driven Computation versus Procedural Control

The inference engine control is based on the frequent re - evaluation of the data store states, not on any static control structure of the program. The computation is often qualified as *data-driven* or *pattern-directed* in contrast to the more traditional procedural control. Rules can communicate with one another only by way of the data, whereas in traditional programming languages procedures and functions explicitly call one another. Unlike instructions, rules are not executed sequentially and it is not always possible to determine through inspection of a set of rules which rule will be executed first or cause the inference engine to terminate.

In contrast to a procedural computation, in which knowledge about the problem domain is mixed in with instructions about the flow of control—although object-oriented programming languages mitigate this entanglement—the inference engine model allows a more complete separation of the knowledge (in the rules) from the control (the inference engine).

3.3.4 Inference Rules

An inference rule is a conditional statement with two parts namely; if clause and a then clause.

This rule is what gives expert systems the ability to find solutions to diagnostic and prescriptive problems. An example of an inference rule is:

If the restaurant choice includes French and the occasion is romantic,
Then the restaurant choice is definitely Paul Bocuse.

An expert system's rule base is made up of many such inference rules. They are entered as separate rules and it is the inference engine that uses them together to draw conclusions. Because each rule is a unit, rules may be deleted or added without affecting other rules - though it should affect which conclusions are reached. One advantage of inference rules over traditional programming is that inference rules use reasoning which more closely resembles human reasoning.

Thus, when a conclusion is drawn, it is possible to understand how this conclusion was reached. Furthermore, because the expert system uses knowledge in a form similar to the that of the expert, it may be easier to retrieve this information directly from the expert.

3.3.5 Chaining

Two methods of reasoning when using inference rules are forward chaining and backward chaining.

Forward chaining starts with the data available and uses the inference rules to extract more data until a desired goal is reached. An inference engine using forward chaining searches the inference rules until it finds one in which the if clause is known to be true . It then concludes the then clause and adds this information to its data. It continues to do this until a goal is reached. Because the data available determines which inference rules are used, this method is also classified as data driven. Backward chaining starts with a list of goals and works backwards to see if there is data which will allow it to conclude any of these goals. An inference engine using backward chaining would search the inference rules until it finds one which has a then clause that matches a desired goal. If the if clause of that inference rule is not known to be true, then it is added to the list of goals. For example, suppose a rule base contains:

(1)   IF X is green THEN X is a frog. (Confidence Factor: +1%)
(2)   IF X is NOT green THEN X is NOT a frog. (Confidence Factor: +99%)
(3)   IF X is a frog THEN X hops. (Confidence Factor: +50%)
(4)   IF X is NOT a frog THEN X does NOT hop. (Confidence Factor +50%)

Suppose a goal is to conclude that Fritz hops. Let X = "Fritz". The rule base would be searched and rule (3) would be selected because its conclusion (the then clause) matches the goal. It is not known that Fritz is a frog, so this "if" statement is added to the goal list. The rule base is again searched and this time rule (1) is selected because its then clause matches the new goal just added to the list. This time, the if clause (Fritz is green) is known to be true and the goal that Fritz hops is concluded. Because the list of goals determines which rules are selected and used, this method is called goal driven.

However, note that if we use confidence factors in even a simplistic fashion - for example, by multiplying them together as if they were like soft probabilities - we get a result that is known with a confidence factor of only one-half of 1%. (This is by multiplying 0.5 x 0.01 = 0.005). This

is useful, because without confidence factors, we might erroneously conclude with certainty that a sea turtle named Fritz hops just by virtue of being green. In Classical logic or Aristotelian term logic systems, there are no probabilities or confidence factors; all facts are regarded as certain. An ancient example from Aristotle states, "Socrates is a man. All men are mortal. Thus Socrates is mortal."

In real world applications, few facts are known with absolute certainty and the opposite of a given statement may be more likely to be true ("Green things in the pet store are not frogs, with the probability or confidence factor of 99% in my pet store survey"). Thus it is often useful when building such systems to try and prove both the goal and the opposite of a given goal to see which is more likely.

3.4   Certainty Factors

One method of operation of expert systems is through a quasi-probabilistic approach with certainty factors: A human, when reasoning, does not always make statements with 100% confidence: he might venture, "If Fritz is green, then he is probably a frog" (after all, he might be a chameleon). This type of reasoning can be imitated using numeric values called confidences. For example, if it is known that Fritz is green, it might be concluded with 0.85 confidence that he is a frog; or, if it is known that he is a frog, it might be concluded with 0.95 confidence that he hops. These Certainty factor (CF) numbers quantify uncertainty in the degree to which the available evidence supports a hypothesis. They represent a degree of confirmation, and are not probabilities in a Bayesian sense. The CF calculus, developed by Shortliffe & Buchanan, increases or decreases the CF associated with a hypothesis as each new piece of evidence becomes available. It can be mapped to a probability update, although degrees of confirmation are not expected to obey the laws of probability. It is important to note, for example, that evidence for hypothesis H may have nothing to contribute to the degree to which Noth is confirmed or disconfirmed (e.g., although a fever lends some support to a diagnosis of infection, fever does not disconfirm alternative hypotheses) and that the sum of CFs of many competing hypotheses may be greater than one (i.e., many hypotheses may be well confirmed based on available evidence).

The CF approach to a rule-based expert system design does not have a widespread following, in part because of the difficulty of meaningfully assigning CFs a priori. (The above example of green creatures being likely to be frogs is excessively naive.) Alternative approaches to quasi-probabilistic reasoning in expert systems involve fuzzy logic, which has a firmer mathematical foundation. Also, rule-engine shells such as

Drools and Jess do not support probability manipulation: they use an alternative mechanism called salience, which is used to prioritize the order of evaluation of activated rules.

In certain areas, as in the tax-advice scenarios discussed below, probabilistic approaches are not acceptable. For instance, a 95% probability of being correct means a 5% probability of being wrong. The rules that are defined in such systems have no exceptions: they are only a means of achieving software flexibility when external circumstances change frequently. Because rules are stored as data, the core software does not need to be rebuilt each time changes to federal and state tax codes are announced.

3.5   Real-Time Adaption

Industrial processes, data networks, and many other systems change their state and even their structure over time. Real time expert systems are designed to reason over time and change conclusions as the monitored system changes. Most of these systems must respond to constantly changing input data, arriving automatically from other systems such as process control systems or network management systems.

Representation includes features for defining changes in belief of data or conclusions over time. This is necessary because data becomes stale. Approaches to this can include decaying belief functions, or the simpler validity interval that simply lets data and conclusions expire after specified time period, falling to "unknown" until refreshed. An often-cited example (attributed to real time expert system pioneer Robert L. Moore) is a hypothetical expert system that might be used to drive a car. Based on video input, there might be an intermediate conclusion that a stop light is green and a final conclusion that it is OK to drive through the intersection. But that data and the subsequent conclusions have a very limited lifetime. You would not want to be a passenger in a car driven based on data and conclusions that were, say, an hour old.

The inference engine must track the times of each data input and each conclusion, and propagate new information as it arrives. It must ensure that all conclusions are still current. Facilities for periodically scanning data, acquiring data on demand, and filtering noise, become essential parts of the overall system. Facilities to reason within a fixed deadline are important in many of these applications.

An overview of requirements for a real-time expert system shell is given in. Examples of real time expert system applications are given in and.

Several conferences were dedicated to real time expert system applications in the chemical process industries, including.

3.5.1 Ability to Make Relevant Inquiries

An additional skill of an expert system is the ability to give relevant inquiries based on previous input from a human user, in order to give better replies or other actions, as well as working faster, which also pleases an impatient or busy human user - it allows a priori volunteering of information that the user considers important.

Also, the user may choose not to respond to every question, forcing the expert system to function in the presence of partial information.

Commercially viable systems will try to optimize the user experience by presenting options for commonly requested information based on a history of previous queries of the system using technology such as forms, augmented by keyword-based search. The gathered information may be verified by a confirmation step (e.g., to recover from spelling mistakes), and now act as an input into a forward-chaining engine. If confirmatory questions are asked in a subsequent phase, based on the rules activated by the obtained information, they are more likely to be specific and relevant. Such abilities can largely be achieved by control flow structures.

In an expert system, implementing the ability to learn from a stored history of its previous use involves employing technologies considerably different from that of rule engines, and is considerably more challenging from a software-engineering perspective. It can, however, make the difference between commercial success and failure. A large part of the revulsion that users felt towards Microsoft's Office Assistant was due to the extreme naivete of its rules ("It looks like you are typing a letter: would you like help?") and its failure to adapt to the user's level of expertise over time (e.g. a user who regularly uses features such as Styles, Outline view, Table of Contents or cross-references is unlikely to be a beginner who needs help writing a letter).

3.6   Explanation System

Another major distinction between expert systems and traditional systems is illustrated by the following answer given by the system when the user answers a question with another question, "Why", as occurred in the above example. The answer is:

A. I am trying to determine the type of restaurant to suggest. So far Indian is not a likely choice. It is possible that French is a likely choice. If I know that if the diner is a wine drinker, and the preferred wine is French, then there is strong evidence that the restaurant choice should include French.

It is very difficult to implement a general explanation system (answering questions like "Why" and "How") in a traditional computer program. An expert system can generate an explanation by retracing the steps of its reasoning. The response of the expert system to the question "Why" exposes the underlying knowledge structure. It is a rule; a set of antecedent conditions which, if true, allow the assertion of a consequent. The rule references values, and tests them against various constraints or asserts constraints onto them. This, in fact, is a significant part of the knowledge structure. There are values, which may be associated with some organizing entity. For example, the individual diner is an entity with various attributes (values) including whether they drink wine and the kind of wine. There are also rules, which associate the currently known values of some attributes with assertions that can be made about other attributes. It is the orderly processing of these rules that dictates the dialogue itself.

3.7   Knowledge Engineering

The building, maintaining and development of expert systems are known as knowledge engineering. Knowledge engineering is a "discipline that involves integrating knowledge into computer systems in order to solve complex problems normally requiring a high level of human.

There are generally three individuals having an interaction in an expert system. Primary among these is the end-user, the individual who uses the system for its problem solving assistance. In the construction and maintenance of the system there are two other roles: the problem domain expert who builds the system and supplies the knowledge base, and a knowledge engineer who assists the experts in determining the representation of their knowledge, enters this knowledge into an explanation module and who defines the inference technique required to solve the problem. Usually the knowledge engineer will represent the problem solving activity in the form of rules. When these rules are created from domain expertise, the knowledge base stores the rules of the expert system.

3.8   General Types of Problems Solved

Expert systems are most valuable to organizations that have a high-level of know-how experience and expertise that cannot be easily transferred to other members. They are designed to carry the intelligence and information found in the intellect of experts and provide this knowledge to other members of the organization for problem-solving purposes.

Typically, the problems to be solved are of the sort that would normally be tackled by a professional, such as a medical professional in the case of clinical decision support systems. Real experts in the problem domain (which will typically be very narrow, for instance "diagnosing skin conditions in teenagers") are asked to provide "rules of thumb" on how they evaluate the problem — either explicitly with the aid of experienced systems developers, or sometimes implicitly, by getting such experts to evaluate test cases and using computer programs to examine the test data and derive rules from that (in a strictly limited manner). Generally, expert systems are used for problems for which there is no single "correct" solution which can be encoded in a conventional algorithm — one would not write an expert system to find the shortest paths through graphs, or to sort data, as there are simpler ways to do these tasks.

Simple systems use simple true/false logic to evaluate data. More sophisticated systems are capable of performing at least some evaluation, taking into account real-world uncertainties, using such methods as fuzzy logic. Such sophistication is difficult to develop and still highly imperfect.

3.9   Different Types of Expert System are

- Rule-Based expert system
- Frames-Based expert system
- Hybird system
- Model-based expert system
- Ready-made system
- Real-Time expert system

3.10   Examples of Applications

Expert systems are designed to facilitate tasks in the fields of accounting, medicine, process control, financial service, production, human resources among others. Typically, the problem area is complex enough that a more simple traditional algorithm cannot provide a proper solution, The foundation of a successful expert system depends on a

139

series of technical procedures and development that may be designed by technicians and related experts. As such, expert systems do not typically provide a definitive answer, but provide probabilistic recommendations. An example of the application of expert systems in the financial field is expert systems for mortgages. Loan departments are interested in expert systems for mortgages because of the growing cost of labour, which makes the handling and acceptance of relatively small loans less profitable. They also see a possibility for standardised, efficient handling of mortgage loan by applying expert systems, appreciating that for the acceptance of mortgages there are hard and fast rules which do not always exist with other types of loans. Another common application in the financial area for expert systems is in trading recommendations in various marketplaces. These markets involve numerous variables and human emotions which may be impossible to deterministically characterize, thus expert systems based on the rules of thumb from experts and simulation data are used. Expert system of this type can range from ones providing regional retail recommendations, like Wishabi, to ones used to assist monetary decisions by financial institutions and governments. Another 1970s and 1980s application of expert systems, which we today would simply call AI, was in computer games. For example, the computer baseball games Earl Weaver Baseball and Tony La Russa Baseball each had highly detailed simulations of the game strategies of those two baseball managers. When a human played the game against the computer, the computer queried the Earl Weaver or Tony La Russa Expert System for a decision on what strategy to follow. Even those choices where some randomness was part of the natural system (such as when to throw a surprise pitch-out to try to trick a runner trying to steal a base) were decided based on probabilities supplied by Weaver or La Russa. Today we would simply say that "the game's AI provided the opposing manager's strategy."

3.11   Advantages

- Compared to traditional programming techniques, expert-system approaches provide the added flexibility (and hence easier modifiability) with the ability to model rules as data rather than as code. In situations where an organization's IT department is overwhelmed by a software-development backlog, rule-engines, by facilitating turnaround, provide a means that can allow organizations to adapt more readily to changing needs.
- In practice, modern expert-system technology is employed as an adjunct to traditional programming techniques, and this hybrid approach allows the combination of the strengths of both approaches. Thus, rule engines allow control through programs

(and user interfaces) written in a traditional language, and also incorporate necessary functionality such as inter-operability with existing database technology.

3.12 Disadvantages

- The Garbage In, Garbage Out (GIGO) phenomenon: A system that uses expert-system technology provides no guarantee about the quality of the rules on which it operates. All self-designated "experts" are not necessarily so, and one notable challenge in expert system design is in getting a system to recognize the limits to its knowledge.
- Expert systems are notoriously narrow in their domain of knowledge— as an amusing example, a researcher used the "skin disease" expert system to diagnose his rust bucket car as likely to have developed measles — and the systems are thus prone to making errors that humans would easily spot. Additionally, once some of the mystique had worn off, most programmers realized that simple expert systems were essentially just slightly more elaborate versions of the decision logic they had already been using. Therefore, some of the techniques of expert systems can now be found in most complex programs without drawing much recognition.
- An expert system or rule-based approach is not optimal for all problems, and considerable knowledge is required so as to not misapply the systems.
- Ease of rule creation and rule modification can be double-edged. A system can be sabotaged by a non-knowledgeable user who can easily add worthless rules or rules that conflict with existing ones. Reasons for the failure of many systems include the absence of (or neglect to employ diligently) facilities for system audit, detection of possible conflict, and rule lifecycle management (e.g. version control, or thorough testing before deployment). The problems to be addressed here are as much technological as organizational.

An example and a good demonstration of the limitations of an expert system is the Windows operating system troubleshooting software located in the "help" section in the taskbar menu. Obtaining technical operating system support is often difficult for individuals not closely involved with the development of the Operating System. Microsoft has designed their expert system to provide solutions, advice, and suggestions to common errors encountered while using their operating systems.

4.0   CONCLUSION

Expert systems now have commercial applications in fields as diverse as medical diagnosis, petroleum engineering, financial investing make financial forecasts and schedule routes for delivery vehicles.

5.0   SUMMARY

In this unit, you learnt:

- Definition of an Expert System
- Knowledge Base and Types of Knowledge Base
- Inference Engine
- Certainty factors
- Real-time adaption
- Knowledge Engineering
- General types of problems solved
- Different types of expert system

6.0   TUTOR-MARKED ASSIGNMENT

i.   Explain expert system.
ii.   Mention and explain two methods of reasoning when using inference rules.
iii. Describe two type of knowledge bases.
iv.   Mention two advantages of expert system.

7.0   REFERENCES/FURTHER READING

Argumentation in Artificial Intelligence by Iyad Rahwan, Guillermo R. Simari.

OWL DL Semantics. http://www.obitko.com/tutorials/ontologies-semantic-web/owl-dl-semantics.html.

Marakas, George. Decision Support Systems in the 21st Century. Prentice Hall, 1999, p.29.

UNIT 2   ROBOTICS

CONTENTS

1.0   INTRODUCTION

Robotics is the branch of technology that deals with the design, construction, operation, structural disposition, manufacture and application of robots. Robotics is related to the sciences of electronics, engineering, mechanics, and software.

2.0   OBJECTIVES

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

•   explain the word robotics
•   list 4 types of robotics you know
•   describe the history of robotics.

3.0   MAIN CONTENT

3.1   What is a Robot?

The word robotics was derived from the word robot, which was introduced to the public by Czech writer Karel Čapek in his play R.U.R. (Rossum's Universal Robots), which premiered in 1921.

According to the Oxford English Dictionary, the word robotics was first used in print by Isaac Asimov, in his science fiction short story "Liar!", published in May 1941 in Astounding Science Fiction. Asimov was unaware that he was coining the term; since the science and technology of electrical devices is electronics, he assumed robotics already referred to the science and technology of robots. In some of Asimov's other works, he states that the first use of the word robotics was in his short story Runaround (Astounding Science Fiction, March 1942). However, the word robotics appears in "Liar!"

3.1.1 Types of Robots

Figure 1:   TOPIO, a humanoid robot, played ping pong at Tokyo International Robot Exhibition (IREX)

Figure 2:   The Shadow robot hand system

Figure 3:   A Pick and Place robot in a factory

3.1.2 History of Robots

Stories of artificial helpers and companions and attempts to create them have a long history.

Figure 4:   A scene from Karel Čapek's 1920 play R.U.R. (Rossum's Universal Robots), showing three robots.

The word robot was introduced to the public by the Czech writer Karel Čapek in his play R.U.R. (Rossum's Universal Robots), published in 1920. The play begins in a factory that makes artificial people called robots creatures who can be mistaken for humans – though they are closer to the modern ideas of androids. Karel Čapek himself did not coin the word. He wrote a short letter in reference to an etymology in the Oxford English Dictionary in which he named his brother Josef Čapek as its actual originator.

In 1927 the Maschinenmensch ("machine-human") gynoid humanoid robot (also called "Parody", "Futura", "Robotrix", or the "Maria

impersonator") was the first and perhaps the most memorable depiction of a robot ever to appear on film was played by German actress Brigitte Helm) in Fritz Lang's film Metropolis.

In 1942 the science fiction writer Isaac Asimov formulated his Three Laws of Robotics and, in the process of doing so, coined the word "robotics" (see details in "Etymology" section below).

In 1948 Norbert Wiener formulated the principles of cybernetics, the basis of practical robotics.

Fully autonomous robots only appeared in the second half of the 20th century. The first digitally operated and programmable robot, the Unimate, was installed in 1961 to lift hot pieces of metal from a die casting machine and stack them. Commercial and industrial robots are widespread today and used to perform jobs more cheaply, or more accurately and reliably, than humans. They are also employed in jobs which are too dirty, dangerous, or dull to be suitable for humans. Robots are widely used in manufacturing, assembly, packing and packaging, transport, earth and space exploration, surgery, weaponry, laboratory research, safety, and the mass production of consumer and industrial goods.

Date	Significance	Robot Name	Inventor
Third century B.C. and earlier	One of the earliest descriptions of automata appears in the Lie Zi text, on a much earlier encounter between King Mu of Zhou (1023-957 BC) and a mechanical engineer known as Yan Shi, an 'artificer'. The latter allegedly presented the king with a life-size, human-shaped figure of his mechanical handiwork.		Yan Shi
First century A.D. and earlier	Descriptions of more than 100 machines and automata, including a fire engine, a wind organ, a coin-operated machine, and a steam-powered engine, in Pneumatica and Automata by Heron of Alexandria		Ctesibius, Philo of Byzantium, Heron of Alexandria, and others
1206	Created early humanoid automata, programmable automaton band	Robot band, hand-	Al-Jazari

		washing automaton,[11] automated moving peacocks[12]	
1495	Designs for a humanoid robot	Mechanical knight	Leonardo da Vinci
1738	Mechanical duck that was able to eat, flap its wings, and excrete	Digesting Duck	Jacques de Vaucanson
1898	Nikola Tesla demonstrates first radio-controlled vessel.	Teleautomaton	Nikola Tesla
1921	First fictional automatons called "robots" appear in the play R.U.R.	Rossum's Universal Robots	Karel Čapek
1930s	Humanoid robot exhibited at the 1939 and 1940 World's Fairs	Elektro	Westinghouse Electric Corporation
1948	Simple robots exhibiting biological behaviors	Elsie and Elmer	William Grey Walter
1956	First commercial robot, from the Unimation company founded by George Devol and Joseph Engelberger, based on Devol's patents	Unimate	George Devol
1961	First installed industrial robot.	Unimate	George Devol
1963	First palletizing robot http://www.ask.com/wiki/Robotics - cite_note-14	Palletizer	Fuji Yusoki Kogyo
1973	First industrial robot with six electromechanically driven axes	Famulus	KUKA Robot Group
1975	Programmable universal manipulation arm, a Unimation product	PUMA	Victor Scheinman

3.2   Components

3.2.1   Power source

At present; mostly (lead-acid) <u>batteries</u> are used, but potential power sources could be:

- pneumatic (compressed gases)
- hydraulics (compressed liquids)
- <u>flywheel energy storage</u>
- organic garbage (through <u>anaerobic digestion</u>)
- faeces (human, animal); may be interesting in a military context as faeces of small combat groups may be reused for the energy requirements of the robot assistant (see DEKA's project Slingshot Stirling engine on how the system would operate)
- still unproven energy sources: for example <u>Nuclear fusion</u>, as yet not used in nuclear reactors whereas <u>Nuclear fission</u> is proven (although there are not many robots using it as a power source apart from the Chinese rover tests).
- radioactive source (such as with the proposed Ford car of the '50s); to those proposed in movies such as *Red Planet*
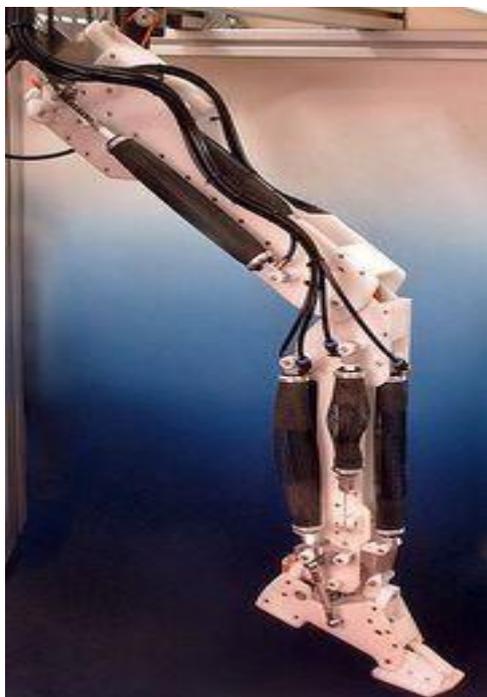
3.2.2   Actuation

Figure 5:   A robotic leg powered by Air Muscles

Actuators are like the "muscles" of a robot, the parts which convert stored energy into movement. By far the most popular actuators are electric motors that spin a wheel or gear, and linear actuators that control industrial robots in factories. But there are some recent advances in alternative types of actuators, powered by electricity, chemicals, or compressed air:

3.2.2.1 Electric motors

The vast majority of robots use electric motors, often brushed and brushless DC motors in portable robots or AC motors in industrial robots and CNC machines.

3.2.2.2 Linear Actuators

Various types of linear actuators move in and out instead of by spinning, particularly when very large forces are needed such as with industrial robotics. They are typically powered by compressed air (pneumatic actuator) or an oil (hydraulic actuator).

3.2.2.3 Series Elastic Actuators

A spring can be designed as part of the motor actuator, to allow improved force control. It has been used in various robots, particularly walking humanoid robots.

3.2.2.4 Air Muscles

Pneumatic artificial muscles, also known as air muscles, are special tubes that contract (typically up to 40%) when air is forced inside it. They have been used for some robot applications.

3.2.2.5 Muscle Wire

Muscle wire, also known as Shape Memory Alloy, Nitinol or Flexinol Wire, is a material that contracts slightly (typically under 5%) when electricity runs through it. They have been used for some small robot applications.

3.2.2.6 Electroactive Polymers

EAPs or EPAMs are a new plastic material that can contract substantially (up to 400%) from electricity, and have been used in facial muscles and arms of humanoid robots, and to allow new robots to float, fly, swim or walk.

3.2.2.7 Piezo Motors

A recent alternative to DC motors are piezo motors or ultrasonic motors. These work on a fundamentally different principle, whereby tiny piezoceramic elements, vibrating many thousands of times per second, cause linear or rotary motion. There are different mechanisms of operation; one type uses the vibration of the piezo elements to walk the motor in a circle or a straight line. Another type uses the piezo elements to cause a nut to vibrate and drive a screw. The advantages of these motors are nanometer resolution, speed, and available force for their size. These motors are already available commercially, and being used on some robots.

3.2.2.8 Elastic Nanotubes

Elastic Nanotubes are a promising artificial muscle technology in early-stage experimental development. The absence of defects in carbon nanotubes enables these filaments to deform elastically by several percent, with energy storage levels of perhaps 13 J/cm3 for metal nanotubes. Human biceps could be replaced with an 8 mm diameter wire of this material. Such compact "muscle" might allow future robots to outrun and out jump humans.

3.3   Sensing

3.3.1 Touch

Current robotic and prosthetic hands receive far less tactile information than the human hand. Recent research has developed a tactile sensor array that mimics the mechanical properties and touch receptors of human fingertips. The sensor array is constructed as a rigid core surrounded by conductive fluid contained by an elastomeric skin. Electrodes are mounted on the surface of the rigid core and are connected to an impedance-measuring device within the core. When the artificial skin touches an object the fluid path around the electrodes is deformed, producing impedance changes that map the forces received from the object. The researchers expect that an important function of such artificial fingertips will be adjusting robotic grip on held objects.
Scientists from several European countries and Israel developed a prosthetic hand in 2009, called SmartHand, which functions like a real one—allowing patients to write with it, type on a keyboard, play piano and perform other fine movements. The prosthesis has sensors which enable the patient to sense real feeling in its fingertips.

3.3.2 Vision

Computer vision is the science and technology of machines that see. As a scientific discipline, computer vision is concerned with the theory behind artificial systems that extract information from images. The image data can take many forms, such as video sequences and views from cameras.

In most practical computer vision applications, the computers are pre-programmed to solve a particular task, but methods based on learning are now becoming increasingly common.

Computer vision systems rely on image sensors which detect electromagnetic radiation which is typically in the form of either visible light or infra-red light. The sensors are designed using solid-state physics. The process by which light propagates and reflects off surfaces is explained using optics. Sophisticated image sensors even require quantum mechanics to provide a complete understanding of the image formation process.

There is a subfield within computer vision where artificial systems are designed to mimic the processing and behavior of biological systems, at different levels of complexity. Also, some of the learning-based methods developed within computer vision have their background in biology.

3.4   Manipulation

Robots which must work in the real world require some way to manipulate objects; pick up, modify, destroy, or otherwise have an effect. Thus the "hands" of a robot are often referred to as end effectors, while the "arm" is referred to as a manipulator. Most robot arms have replaceable effectors, each allowing them to perform some small range of tasks. Some have a fixed manipulator which cannot be replaced, while a few have one very general purpose manipulator, for example a humanoid hand.

For the definitive guide to all forms of robot end-effectors, their design, and usage consult the book "Robot Grippers".

3.4.1 Mechanical Grippers

One of the most common effectors is the gripper. In its simplest manifestation it consists of just two fingers which can open and close to

pick up and let go of a range of small objects. Fingers can for example be made of a chain with a metal wire run through it. See Shadow Hand.

3.4.2 Vacuum Grippers

Vacuum grippers are very simple astrictive devices, but can hold very large loads provided the prehension surface is smooth enough to ensure suction.

Pick and place robots for electronic components and for large objects like car windscreens, often use very simple vacuum grippers.

3.4.3 General Purpose Effectors

Some advanced robots are beginning to use fully humanoid hands, like the Shadow Hand, MANUS, and the Schunk hand. These highly dexterous manipulators with as many as 20 degrees of freedom and hundreds of tactile sensors.

3.5   Locomotion

3.5.1 Rolling Robots

Figure 6:   Segway in the Robot museum in Nagoya.

For simplicity most mobile robots have four wheels or a number of continuous tracks. Some researchers have tried to create more complex wheeled robots with only one or two wheels. These can have certain advantages such as greater efficiency and reduced parts, as well as allowing a robot to navigate in confined places that a four wheeled robot would not be able to.

3.5.1.1 Two-Wheeled Balancing Robots

Balancing robots generally use a gyroscope to detect how much a robot is falling and then drive the wheels proportionally in the opposite direction, to counter-balance the fall at hundreds of times per second, based on the dynamics of an inverted pendulum. Many different balancing robots have been designed. While the Segway is not commonly thought of as a robot, it can be thought of as a component of a robot, such as NASA's Robonaut that has been mounted on a Segway.

3.5.1.2   One-Wheeled Balancing Robots

A one-wheeled balancing robot is an extension of a two-wheeled balancing robot so that it can move in any 2D direction using a round ball as its only wheel. Several one-wheeled balancing robots have been designed recently, such as Carnegie Mellon University's "Ballbot" that is the approximate height and width of a person, and Tohoku Gakuin University's "BallIP". Because of the long, thin shape and ability to maneuver in tight spaces, they have the potential to function better than other robots in environments with people.

3.5.1.3 Spherical Orb Robots

Several attempts have been made in robots that are completely inside a spherical ball, either by spinning a weight inside the ball, or by rotating the outer shells of the sphere. These have also been referred to as an orb bot or a ball bot.

3.5.1.4 Six-Wheeled Robots

Using six wheels instead of four wheels can give better traction or grip in outdoor terrain such as on rocky dirt or grass.

3.5.1.5 Tracked Robots

Tank tracks provide even more traction than a six-wheeled robot. Tracked wheels behave as if they were made of hundreds of wheels, therefore are very common for outdoor and military robots, where the robot must drive on very rough terrain. However, they are difficult to use indoors such as on carpets and smooth floors. Examples include NASA's Urban Robot "Urbie".

3.5.2 Walking Applied to Robots

Figure 6:   iCub robot, designed by the RobotCub Consortium

Walking is a difficult and dynamic problem to solve. Several robots have been made which can walk reliably on two legs; however none have yet been made which are as robust as a human. Many other robots have been built that walk on more than two legs, due to these robots being significantly easier to construct. Hybrids too have been proposed in movies such as I, Robot, where they walk on 2 legs and switch to 4 (arms+legs) when going to a sprint. Typically, robots on 2 legs can walk well on flat floors and can occasionally walk up stairs. None can walk over rocky, uneven terrain. Some of the methods which have been tried are:

3.5.2.1 ZMP Technique

The Zero Moment Point (ZMP) is the algorithm used by robots such as Honda's ASIMO. The robot's onboard computer tries to keep the total inertial forces (the combination of earth's gravity and the acceleration and deceleration of walking), exactly opposed by the floor reaction force (the force of the floor pushing back on the robot's foot). In this way, the two forces cancel out, leaving no moment (force causing the robot to rotate and fall over). However, this is not exactly how a human walks, and the difference is obvious to human observers, some of whom have pointed out that ASIMO walks as if it needs the lavatory. ASIMO's walking algorithm is not static, and some dynamic balancing is used (see below). However, it still requires a smooth surface to walk on.

3.5.2.2 Hopping

Several robots, built in the 1980s by Marc Raibert at the MIT Leg Laboratory, successfully demonstrated very dynamic walking. Initially, a robot with only one leg, and a very small foot, could stay upright simply by hopping. The movement is the same as that of a person on a pogo stick. As the robot falls to one side, it would jump slightly in that direction, in order to catch itself. Soon, the algorithm was generalised to two and four legs. A bipedal robot was demonstrated running and even performing somersaults. A quadruped was also demonstrated which could trot, run, pace, and bound. For a full list of these robots, see the MIT Leg Lab Robots page.

3.5.2 .3 Dynamic Balancing (Controlled Falling)

A more advanced way for a robot to walk is by using a dynamic balancing algorithm, which is potentially more robust than the Zero Moment Point technique, as it constantly monitors the robot's motion, and places the feet in order to maintain stability. This technique was recently demonstrated by Anybots' Dexter Robot,http://www.ask.com/ wiki/Robotics - cite_note-64 which is so stable, it can even jump. Another example is the TU Delft Flame.

3.5.2.4 Passive Dynamics

Perhaps the most promising approach utilizes passive dynamics where the momentum of swinging limbs is used for greater efficiency. It has been shown that totally unpowered humanoid mechanisms can walk down a gentle slope, using only gravity to propel them. Using this technique, a robot need only supply a small amount of motor power to walk along a flat surface or a little more to walk up a hill. This technique promises to make walking robots at least ten times more efficient than ZMP walkers, like ASIMO.

3.5.3 Other methods of locomotion

Figure 7:   RQ-4 Global Hawk unmanned aerial vehicle

3.5.3.1 Flying

A modern passenger airliner is essentially a flying robot, with two humans to manage it. The autopilot can control the plane for each stage of the journey, including takeoff, normal flight, and even landing. Other flying robots are uninhabited, and are known as unmanned aerial vehicles (UAVs). They can be smaller and lighter without a human pilot onboard, and fly into dangerous territory for military surveillance missions. Some can even fire on targets under command. UAVs are also being developed which can fire on targets automatically, without the need for a command from a human. Other flying robots include cruise missiles, the Entomopter, and the Epson micro helicopter robot. Robots such as the Air Penguin, Air Ray, and Air Jelly have lighter-than-air bodies, propelled by paddles, and guided by sonar.

Figure 8:   Two robot snakes. Left **one** has 64 motors (with 2 degrees of freedom per segment), the right one 10.

3.5.3.2 Snaking

Several snake robots have been successfully developed. Mimicking the way real snakes move, these robots can navigate very confined spaces, meaning they may one day be used to search for people trapped in collapsed buildings. The Japanese ACM-R5 snake robot can even navigate both on land and in water.

3.5.3.3 Skating

A small number of skating robots have been developed, one of which is a multi-mode walking and skating device. It has four legs, with unpowered wheels, which can either step or roll. Another robot, Plen, can use a miniature skateboard or rollerskates, and skate across a desktop.

3.5.3.4 Climbing

Several different approaches have been used to develop robots that have the ability to climb vertical surfaces. One approach mimicks the movements of a human climber on a wall with protrusions; adjusting the center of mass and moving each limb in turn to gain leverage. An example of this is Capuchin, built by Stanford University, California. Another approach uses the specialised toe pad method of wall-climbing geckoes, which can run on smooth surfaces such as vertical glass. Examples of this approach include Wallbot and Stickybot. China's "Technology Daily" November 15, 2008 reported New Concept Aircraft (ZHUHAI) Co. Ltd. Dr. Li Hiu Yeung and his research group have recently successfully developed the bionic gecko robot "Speedy Freelander". According to Dr. Li introduction, this gecko robot can rapidly climbing up and down in a variety of building walls, ground and vertical wall fissure or walking upside down on the ceiling, it is able to adapt on smooth glass, rough or sticky dust walls as well as the various surface of metallic materials and also can automatically identify obstacles, circumvent the bypass and flexible and realistic movements. Its flexibility and speed are comparable to the natural gecko. A third approach is to mimick the motion of a snake climbing a pole.

3.5.3.5 Swimming (like a Fish)

It is calculated that when swimming some fish can achieve a propulsive efficiency greater than 90%. Furthermore, they can accelerate and maneuver far better than any man-made boat or submarine, and produce less noise and water disturbance. Therefore, many researchers studying underwater robots would like to copy this type of locomotion. Notable examples are the Essex University Computer Science Robotic Fish, and the Robot Tuna built by the Institute of Field Robotics, to analyze and mathematically model thunniform motion. The Aqua Penguin, designed and built by Festo of Germany, copies the streamlined shape and propulsion by front "flippers" of penguins. Festo have also built the Aqua Ray and Aqua Jelly, which emulate the locomotion of manta ray, and jellyfish, respectively.

3.6 Environmental interaction and navigation

Figure 9: RADAR, GPS, LIDAR, ... are all combined to provide proper navigation and obstacle avoidance

Though a significant percentage of robots in commission today are either human controlled, or operate in a static environment, there is an increasing interest in robots that can operate autonomously in a dynamic environment. These robots require some combination of navigation hardware and software in order to traverse their environment. In particular unforeseen events (e.g. people and other obstacles that are not stationary) can cause problems or collisions. Some highly advanced robots as ASIMO, EveR-1, Meinü robot have particularly good robot navigation hardware and software. Also, self-controlled cars, Ernst Dickmanns' driverless car, and the entries in the DARPA Grand Challenge, are capable of sensing the environment well and subsequently making navigational decisions based on this information. Most of these robots employ a GPS navigation device with waypoints, along with radar, sometimes combined with other sensory data such as LIDAR, video cameras, and inertial guidance systems for better navigation between waypoints.
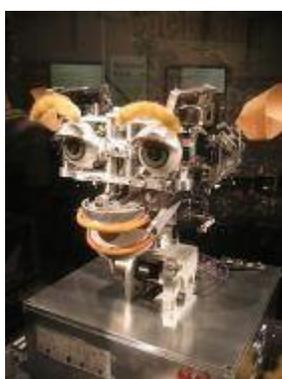
3.7 Human-Robot Interaction

Figure 10: Kismet can produce a range of facial expressions.

If robots are to work effectively in homes and other non-industrial environments, the way they are instructed to perform their jobs, and especially how they will be told to stop will be of critical importance. The people who interact with them may have little or no training in robotics, and so any interface will need to be extremely intuitive. Science fiction authors also typically assume that robots will eventually be capable of communicating with humans through speech, gestures, and facial expressions, rather than a command-line interface. Although speech would be the most natural way for the human to communicate, it is unnatural for the robot. It will probably be a long time before robots interact as naturally as the fictional C-3PO.

3.7.1 Speech Recognition

Interpreting the continuous flow of sounds coming from a human, in real time, is a difficult task for a computer, mostly because of the great variability of speech. The same word, spoken by the same person may sound different depending on local acoustics, volume, the previous word, whether or not the speaker has a cold, etc.. It becomes even harder when the speaker has a different accent. Nevertheless, great strides have been made in the field since Davis, Biddulph, and Balashek designed the first "voice input system" which recognized "ten digits spoken by a single user with 100% accuracy" in 1952. Currently, the best systems can recognize continuous, natural speech, up to 160 words per minute, with an accuracy of 95%.

3.7.2 Robotic Voice

Other hurdles exist when allowing the robot to use voice for interacting with humans. For social reasons, synthetic voice proves suboptimal as a communication medium, making it necessary to develop the emotional component of robotic voice through various techniques.

3.7.3 Gestures

One can imagine, in the future, explaining to a robot chef how to make a pastry, or asking directions from a robot police officer. In both of these cases, making hand gestures would aid the verbal descriptions. In the first case, the robot would be recognizing gestures made by the human, and perhaps repeating them for confirmation. In the second case, the robot police officer would gesture to indicate "down the road, then turn right". It is likely that gestures will make up a part of the interaction between humans and robots. A great many systems have been developed to recognize human hand gestures.

3.7.4 Facial Expression

Facial expressions can provide rapid feedback on the progress of a dialog between two humans, and soon it may be able to do the same for humans and robots. Robotic faces have been constructed by Hanson Robotics using their elastic polymer called Frubber, allowing a great amount of facial expressions due to the elasticity of the rubber facial coating and imbedded subsurface motors (servos) to produce the facial expressions. The coating and servos are built on a metal skull. A robot should know how to approach a human, judging by their facial expression and body language. Whether the person is happy, frightened, or crazy-looking affects the type of interaction expected of the robot. Likewise, robots like Kismet and the more recent addition, Nexi can produce a range of facial expressions, allowing it to have meaningful social exchanges with humans.

3.7.5 Artificial Emotions

Artificial emotions can also be imbedded and are composed of a sequence of facial expressions and/or gestures. As can be seen from the movie Final Fantasy: The Spirits Within, the programming of these artificial emotions is complex and requires a great amount of human observation. To simplify this programming in the movie, presets were created together with a special software program. This decreased the amount of time needed to make the film. These presets could possibly be transferred for use in real-life robots.

3.7.6 Personality

Many of the robots of science fiction have a personality, something which may or may not be desirable in the commercial robots of the future. Nevertheless, researchers are trying to create robots which appear to have a personality: i.e. they use sounds, facial expressions, and body language to try to convey an internal state, which may be joy, sadness, or fear. One commercial example is Pleo, a toy robot dinosaur, which can exhibit several apparent emotions.

3.8   Control

Figure 11:   A robot-manipulated marionette, with complex control systems

The mechanical structure of a robot must be controlled to perform tasks. The control of a robot involves three distinct phases - perception, processing, and action (robotic paradigms). Sensors give information about the environment or the robot itself (e.g. the position of its joints or its end effector). This information is then processed to calculate the appropriate signals to the actuators (motors) which move the mechanical.

The processing phase can range in complexity. At a reactive level, it may translate raw sensor information directly into actuator commands. Sensor fusion may first be used to estimate parameters of interest (e.g. the position of the robot's gripper) from noisy sensor data. An immediate task (such as moving the gripper in a certain direction) is inferred from these estimates. Techniques from control theory convert the task into commands that drive the actuators.

At longer time scales or with more sophisticated tasks, the robot may need to build and reason with a "cognitive" model. Cognitive models try to represent the robot, the world, and how they interact. Pattern recognition and computer vision can be used to track objects. Mapping techniques can be used to build maps of the world. Finally, motion planning and other artificial intelligence techniques may be used to figure out how to act. For example, a planner may figure out how to achieve a task without hitting obstacles, falling over, etc.

3.8.1 Autonomy Levels

Control systems may also have varying levels of autonomy.

Direct interaction is used for haptic or tele-operated devices, and the human has nearly complete control over the robot's motion.

Operator-assist modes have the operator commanding medium-to-high-level tasks, with the robot automatically figuring out how to achieve them.

An autonomous robot may go for extended periods of time without human interaction. Higher levels of autonomy do not necessarily require more complex cognitive capabilities. For example, robots in assembly plants are completely autonomous, but operate in a fixed pattern.

Another classification takes into account the interaction between human control and the machine motions.

Teleoperation. A human controls each movement, each machine actuator change is specified by the operator.

Supervisory. A human specifies general moves or position changes and the machine decides specific movements of its actuators.

Task-level autonomy. The operator specifies only the task and the robot manages itself to complete it.

Full autonomy. The machine will create and complete all its tasks without human interaction.

3.9   Robotics Research

Much of the research in robotics focuses not on specific industrial tasks, but on investigations into new types of robots, alternative ways to think about or design robots, and new ways to manufacture them but other investigations, such as MIT's cyberflora project, are almost wholly academic.

A first particular new innovation in robot design is the opensourcing of robot-projects. To describe the level of advancement of a robot, the term "Generation Robots" can be used. This term is coined by Professor Hans Moravec, Principal Research Scientist at the Carnegie Mellon University Robotics Institute in describing the near future evolution of robot technology. First generation robots, Moravec predicted in 1997,

should have an intellectual capacity comparable to perhaps a lizard and should become available by 2010. Because the first generation robot would be incapable of learning, however, Moravec predicts that the second generation robot would be an improvement over the first and become available by 2020, with intelligence maybe comparable to that of a mouse. The third generation robot should have intelligence comparable to that of a monkey. Though fourth generation robots, robots with human intelligence, professor Moravec predicts, would become possible, he does not predict this happening before around 2040 or 2050.

The second is Evolutionary Robots. This is a methodology that uses evolutionary computation to help design robots, especially the body form, or motion and behavior controllers. In a similar way to natural evolution, a large population of robots is allowed to compete in some way, or their ability to perform a task is measured using a fitness function. Those that perform worst are removed from the population, and replaced by a new set, which have new behaviors based on those of the winners. Over time the population improves, and eventually a satisfactory robot may appear. This happens without any direct programming of the robots by the researchers. Researchers use this method both to create better robots, and to explore the nature of evolution. Because the process often requires many generations of robots to be simulated, this technique may be run entirely or mostly in simulation, then tested on real robots once the evolved algorithms are good enough. Currently, there are about 1 million industrial robots toiling around the world, and Japan is the top country having high density of utilizing robots in its manufacturing industry.

3.9.1 Dynamics and Kinematics

The study of motion can be divided into kinematics and dynamics. Direct kinematics refers to the calculation of end effector position, orientation, velocity, and acceleration when the corresponding joint values are known. Inverse kinematics refers to the opposite case in which required joint values are calculated for given end effector values, as done in path planning. Some special aspects of kinematics include handling of redundancy (different possibilities of performing the same movement), collision avoidance, and singularity avoidance. Once all relevant positions, velocities, and accelerations have been calculated using kinematics, methods from the field of dynamics are used to study the effect of forces upon these movements. Direct dynamics refers to the calculation of accelerations in the robot once the applied forces are known. Direct dynamics is used in computer simulations of the robot. Inverse dynamics refers to the calculation of the actuator forces

necessary to create a prescribed end effector acceleration. This information can be used to improve the control algorithms of a robot.

In each area mentioned above, researchers strive to develop new concepts and strategies, improve existing ones, and improve the interaction between these areas. To do this, criteria for "optimal" performance and ways to optimize design, structure, and control of robots must be developed and implemented.

3.10   Education and Training

Figure 12:   The SCORBOT-ER 4u - educational robot.

Robots recently became a popular tool in raising interests in computing for middle and high school students. First year computer science courses at several universities were developed which involves the programming of a robot instead of the traditional software engineering based coursework.

3.10.1 Career training

Universities offer Bachelors, Masters and Doctoral degrees in the field of robotics. Select Private Career Colleges and vocational schools offer robotics training to train individuals towards being job ready and employable in the emerging robotics industry.

3.10.2 Certification

The Robotics Certification Standards Alliance (RCSA) is an international robotics certification authority who confers various industry and educational related robotics certifications.

3.11   Employment

Figure 13:   A robot technician builds small all-terrain robots. (Courtesy: MobileRobots Inc)

Robotics is an essential component in any modern manufacturing environment. As factories increase their use of robots, the number of robotics related jobs grow and have been observed to be on a steady rise.

3.11.1 Effects on Unemployment

Some analysts, such as Martin Ford, argue that robots and other forms of automation will ultimately result in significant unemployment as machines begin to match and exceed the capability of workers to perform most jobs. At present the negative impact is only on menial and repetitive jobs, and there is actually a positive impact on the number of jobs for highly skilled technicians, engineers, and specialists. However, these highly skilled jobs are not sufficient in number to offset the greater decrease in employment among the general population, causing structural unemployment in which overall (net) unemployment rises.

As robotics and artificial intelligence develop further, some worry even many skilled jobs may be threatened. In conventional economic theory this should merely cause an increase in the productivity of the involved industries, resulting in higher demand for other goods, and hence higher labour demand in these sectors, off-setting whatever negatives are caused. Conventional theory describes the past well but may not

describe the future due to shifts in the parameter values that shape the context.

4.0   CONCLUSION

Robotics is an essential component in any modern manufacturing environment. As factories increase their use of robots, the number of robotics related jobs grow and have been observed to be on a steady rise. As robotics and artificial intelligence develop further, some worry even many skilled jobs may be threatened.

5.0   SUMMARY

In this unit, you learnt:

- Robots and Types of Robots
- History of Robots
- Components of Robots
- Robotics research
- Education and training
- Robots and Employment.

6.0   TUTOR-MARKED ASSIGNMENT

1.   Explain the word Robotics.
2.   List five (5) areas where Robots can be used.
3.   List three (3) areas where Robots can be used for now.

7.0   REFERENCES/FURTHER READING

Asimov, I. (1996). [1995]. *"The Robot Chronicles".* Gold. London: Voyager. pp. 224–225. ISBN 0-00-648202-3.

Crane, C. D. & Duffy, J. (1998-03). Kinematic Analysis of Robot Manipulators. Cambridge: University Press. ISBN 0521570638. http://www.cambridge.org/us/catalogue/catalogue.asp?isbn=0521 570638.

Nishibori; *et al*. (2003). Robot Hand with Fingers Using Vibration-Type Ultrasonic Motors (Driving Characteristics). Journal of Robotics and Mechatronics. http://www.fujipress.jp/finder/xslt.php?mode =present&inputfile=ROBOT001500060002.xml.

Park; *et al.* (2005). Synthetic Personality in Robots and its Effect on Human-Robot Relationship.

Rosheim, M. E. (1994). Robot Evolution: The Development of Anthrobotics. Wiley-IEEE. pp. 9–10. ISBN 0471026220.