CHAPTER 1

Unit I

Intelligence Searching Techniques

Syllabus Topics

Artificial Intelligence : Introduction, Typical Applications.

State Space Search : Depth Bounded DFS, Depth First Iterative Deepening.

Heuristic Search : Heuristic Functions, Best First Search, Hill Climbing, Variable Neighborhood Descent, Beam Search, Tabu Search.

Optimal Search : A* algorithm, Iterative Deepening A* , Recursive Best First Search, Pruning the CLOSED and OPEN Lists.

Syllabus Topic : Artificial Intelligence - Introduction

1.1 Artificial Intelligence

Q. 1.1.1 What is artificial intelligence ? Explain the Turing test approach to act humanly.
(Refer sections 1.1 and 1.1.1) **(4 Marks)**

- John McCarthy who has coined the word "Artificial Intelligence" in 1956, has defined AI as "the science and engineering of making intelligent machines", especially intelligent computer programs.
- **Artificial Intelligence (AI)** is relevant to any intellectual task where the machine needs to take some decision or choose the next action based on the current state of the system, in short act intelligently or rationally. As it has a very wide range of applications, it is truly a universal field.
- In simple words, Artificial Intelligent System works like a Human Brain, where a machine or software shows intelligence while performing given tasks; such systems are called **intelligent systems** or **expert systems**. You can say that these systems can "think" while generating output!!!
- AI is one of the newest fields in science and engineering and has a wide variety of application fields. AI applications range from the general fields like learning, perception and prediction to the specific field, such as writing stories, proving mathematical theorems, driving a bus on a crowded street, diagnosing diseases, and playing chess.
- AI is the study of how to make machines do thing which at the moment people do better. Following are the four approaches to define AI.

DEFINITION

*In general, **artificial intelligence** is the study of how to make machines do things which at the moment human do better. Following are the four approaches to define AI.*

- Historically, all four approaches have been followed by different group of people with different methods.

1.1.1 Acting Humanly : The Turing Test Approach

Q. 1.1.2 Explain Turing test designed for satisfactory operational definition of AI.
(Refer section 1.1.1) **(8 Marks)**

DEFINITION

***Definition 1** : "The art of creating machines that perform functions that requires intelligence when performed by people." (Kurzweil, 1990)*

DEFINITION

***Definition 2** : "The study of how to make computers do things at which, at the moment, people are better." (Rich and Knight, 1991)*

- To judge whether the system can act like a human, Sir Alan Turing had designed a test known as **Turing test**.
- As shown in Fig. 1.1.1, in Turing test, a computer needs to interact with a human interrogator by answering his questions in written format. Computer passes the test if a human interrogator, cannot identify whether the written responses are from a person or a computer. Turing test is valid even after 60 year of research.

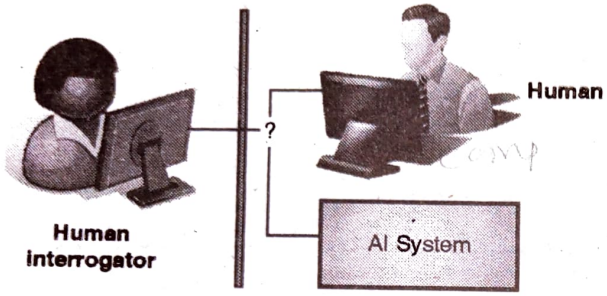


Fig. 1.1.1 : Turing Test Environment

- For this test, the computer would need to possess the following capabilities :
  1. **Natural Language Processing (NLP) :** This unit enables computer to interpret the English language and communicate successfully.
  2. **Knowledge Representation :** This unit is used to store knowledge gathered by the system through input devices.
  3. **Automated Reasoning :** This unit enables to analyze the knowledge stored in the system and makes new inferences to answer questions.
  4. **Machine Learning :** This unit learns new knowledge by taking current input from the environment and adapts to new circumstances, thereby enhancing the knowledge base of the system.
- To pass total Turing test, the computer will also need to have **computer vision,** which is required to perceive objects from the environment and **Robotics,** to manipulate those objects.
- Fig. 1.1.2 lists all the capabilities a computer needs to have in order to exhibit artificial intelligence. Mentioned above are the six disciplines which implement most of the artificial intelligence.

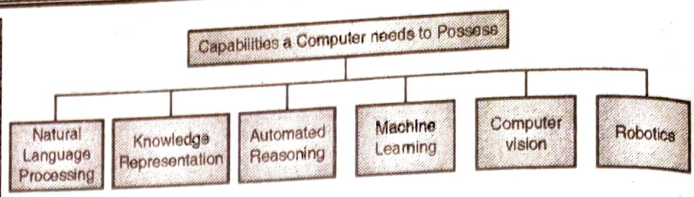


Fig. 1.1.2 : Capabilities a Computer needs to possess

1.1.2 Thinking Humanly : The Cognitive Modelling Approach

DEFINITION

Definition 1 : *"The exciting new effort to make computers think ... machines with minds, in the full and literal sense". (Haugeland, 1985)*

DEFINITION

Definition 2 : *"The automation of activities that we associate with human thinking, activities such as decision making, problem solving, learning ..." (Hellman, 1978)*

- **Cognitive science :** It is interdisciplinary field which combines computer models from Artificial Intelligence with the techniques from psychology in order to construct precise and testable theories for working of human mind.
- In order to make machines think like human, we need to first understand how human think. Research showed that there are three ways using which human's thinking pattern can be caught.
  1. **Introspection** through which human can catch their own thoughts as they go by.
  2. **Psychological experiments** can be carried out by observing a person in action.
  3. **Brain imaging** can be done by observing the brain in action.
- By catching the human thinking pattern, it can be implemented in computer system as a program and if the program's input output matches with that of human, then it can be claimed that the system can operate like humans.

1.1.3 Thinking Rationally : The "Laws of Thought" Approach

DEFINITION

Definition 1 : *"The study of mental faculties through the use of computational models". (Charniak and McDermott, 1985)*

DEFINITION

Definition 2 : *"The study of the computations that make it possible to perceive, reason, and act".*

- The laws of thought are supposed to implement the operation of the mind and their study initiated the field called logic. It provides precise notations to express facts of the real world.
- It also includes reasoning and "right thinking" that is irrefutable thinking process. Also computer programs based on those logic notations were developed to create intelligent systems.

☞ **Two problems in this approach**

1. This approach is not suitable to use when 100% knowledge is not available for any problem.
2. As vast number of computations was required even to implement a simple human reasoning process; practically, all problems were not solvable because even problems with just a few hundred facts can exhaust the computational resources of any computer.

1.1.4 Acting Rationally : The Rational Agent Approach

DEFINITION

Definition 1 : *"Computational Intelligence is the study of the design of intelligent agents". (Poole et at, 1998)*

DEFINITION

Definition 2 : *"AI ... is concerned with intelligent behaviour in artifacts". (Nilsson, 1998)*

☞ **Rational Agent**

- Agents perceive their environment through sensors over a prolonged time period and adapt to change to create and pursue goals and take actions through actuators to achieve those goals. A rational agent is the one that does "right" things and acts rationally so as to achieve the best outcome even when there is uncertainty in knowledge.
- **The rational-agent approach has two advantages over the other approaches**
  1. As compared to other approaches this is the more general approach as, rationality can be achieved by selecting the correct inference from the several available.
  2. Rationality has specific standards and is mathematically well defined and completely general and can be used to develop agent designs that achieve it. Human behavior, on the other hand, is very subjective and cannot be proved mathematically.
- The two approaches namely, thinking humanly and thinking rationally are based on the reasoning expected from intelligent systems while; the other two acting humanly and acting rationally are based on the intelligent behaviour expected from them.
- In our syllabus we are going to study acting rationally approach.

Syllabus Topic : Introduction

1.2 Introduction

As AI is a very broad concept, there are different types or forms of AI. The critical categories of AI can be based on the capacity of intelligent program or what the program is able to do. Under this consideration there are three main categories :

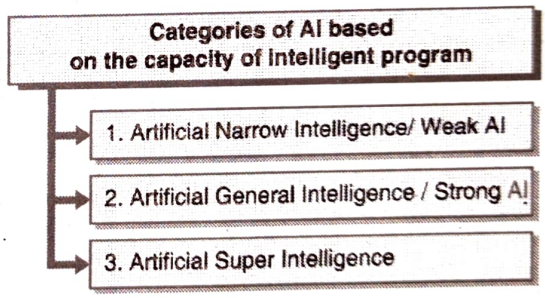


Fig. 1.2.1 : Categories of AI based on the capacity of Intelligent program

➜ **1. Artificial Narrow Intelligence/ Weak AI**

Weak AI is AI that specializes in one area. It is not a general purpose intelligence. An intelligent agent is built to solve a particular problem or to perform a specific task is termed as narrow intelligence or weak AI. For example, it took years of AI development to be able to beat the chess grandmaster, and since then we have not been able to beat the machines at chess. But that is all it can do, which is does extremely well.

➜ **2. Artificial General Intelligence / Strong AI**

Strong AI or general AI refers to intelligence demonstrated by machines in performing any intellectual task that human

can perform. Developing strong AI is much harder than developing weak AI. Using artificial general intelligence machines can demonstrate human abilities like reasoning, planning, problem solving, comprehending complex ideas, learning from self experiences, etc. Many companies, corporations' are working on developing a general intelligence but they are yet to complete it.

3. Artificial Super Intelligence

As defined by a leading AI thinker Nick Bostrom, "Super intelligence is an intellect that is much smarter than the best human brains in practically every field, including scientific creativity, general wisdom and social skills." Super intelligence ranges from a machine which is just a little smarter than a human to a machine that is trillion times smarter. Artificial super intelligence is the ultimate power of AI.

1.2.1 Components of AI

AI is a vast field for research and it has got applications in almost all possible domains. By keeping this in mind, components of AI can be identified as follows : (refer Fig. 1.2.2)

1. Perception
2. Knowledge representation
3. Learning
4. Reasoning
5. Problem Solving
6. Natural Language Processing (language-understanding)

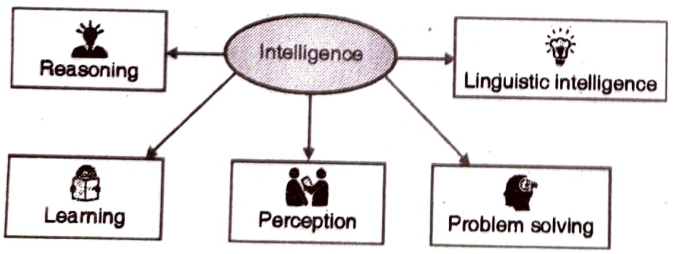


Fig. 1.2.2 : Components of AI

1. Perception

In order to work in the environment, intelligent agents need to scan the environment and the various objects in it. Agent scans the environment using various sense organs like camera, temperature sensor, etc. This is called as perception. After capturing various scenes, perceiver analyses the different objects in it and extracts their features and relationships among them.

2. Knowledge representation

The information obtained from environment through sensors may not be in the format required by the system. Hence, it need to be represented in standard formats for further processing like learning various patterns, deducing inference, comparing with past objects, etc. There are various knowledge representation techniques like Prepositional logic and first order logic.

3. Learning

Learning is a very essential part of AI and it happens in various forms. The simplest form of learning is by trial and error. In this form the program remembers the action that has given desired output and discards the other trial actions and learns by itself. It is also called as unsupervised learning. In case of rote learning, the program simply remembers the problem solution pairs or individual items. In other case, solution to few of the problems is given as input to the system, basis on which the system or program needs to generate solutions for new problems. This is known as supervised learning.

4. Reasoning

Reasoning is also called as logic or generating inferences form the given set of facts. Reasoning is carried out based on strict rule of validity to perform a specified task. Reasoning can be of two types, deductive or inductive. The deductive reasoning is in which the truth of the premises guarantees the truth of the conclusion while, in case of inductive reasoning, the truth of the premises supports the conclusion, but it cannot be fully dependent on the premises. In programming logic generally deductive inferences are used. Reasoning involves drawing inferences that are relevant to the given problem or situation.

5. Problem-solving

AI addresses huge variety of problems. For example, finding out winning moves on the board games, planning actions in order to achieve the defined task, identifying various objects from given images, etc. As per the types of problem, there is variety of problem solving strategies in AI. Problem solving methods are mainly divided into general purpose methods and special purpose methods. General purpose methods are applicable to wide range of problems while, special purpose methods are customized to solve particular type of problems.

6. Natural Language Processing

Natural Language Processing, involves machines or robots to understand and process the language that human speak, and infer knowledge from the speech input. It also involves the active participation from machine in the form of dialog i.e. NLP aims at the text or verbal output from the machine or robot. The input and output of an NLP system can be speech and written text respectively.

1.2.2 History of Artificial Intelligence

- The term **Artificial Intelligence (AI)** was introduced by John McCarthy, in 1955. He defined artificial intelligence as **"The science and engineering of making intelligent machines"**.
- Mathematician **Alan Turing** and others presented a study based on logic driven computational theories which showed that any computer program can work by simply shuffling "0" and "1" (i.e. electricity off and electricity *on*). Also, during that time period, research was going on in the areas like Automations, Neurology, Control theory, Information theory, etc.
- This inspired a group of researchers to think about the possibility of creating an electronic brain. In the year 1956 a conference was conducted at the campus of Dartmouth College where the field of artificial intelligence research was founded.
- This conference was attended by John McCarthy, Marvin Minsky, Allen Newell and Herbert Simon, etc., who are supposed to be the pioneers of artificial intelligence research for a very long time. During that time period, Artificial Intelligence systems were developed by these researchers and their students.
- **Let's see few examples of such artificial intelligent systems :**
  - **Game - Checkers :** Computer played as an opponent,
  - **Education - Algebra :** For solving word problems,
  - **Education - Math :** Proving logical theorems,
  - **Education - Language :** Speaking English, etc.
- During that time period these founders predicted that in few years machines can do any work that a man can do, but they failed to recognize the difficulties which can be faced.
- Meanwhile we will see the ideas, viewpoints and techniques which Artificial Intelligence has inherited from other disciplines. They can be given as follows:

1. **Philosophy :** Theories of reasoning and learning have emerged, along with the viewport that the mind is constituted by the operation of a physical system.
2. **Mathematical :** Formal theories of logic, probability, decision making and computation have emerged.
3. **Psychology :** Psychology has emerged tools to investigate the human mind and a scientific language which are used to express the resulting theories.
4. **Linguistic :** Theories of the structure and meaning of language have emerged.
5. **Computer science :** The tools which can make artificial intelligence a reality has emerged.

Syllabus Topic : Typical Applications

1.3 Typical Applications of Artificial Intelligence

Q. 1.3.1 Explain the Artificial Intelligence Applications. (Refer section 1.3) (8 Marks)

- You must have seen use of Artificial Intelligence in many SCI-FI movies. To name a few we have I Robot, Wall-E, The Matrix Trilogy, Star Wars, etc. movies. Many a times these movies show positive potential of using AI and sometimes also emphasize the dangers of using AI. Also there are games

based on such movies, which show us many probable applications of AI.

- Artificial Intelligence is commonly used for problem solving by analyzing or/and predicting output for a system. AI can provide solutions for constraint satisfaction problems. It is used in wide range of fields for example in diagnosing diseases, in business, in education, in controlling a robots, in entertainment field, etc.

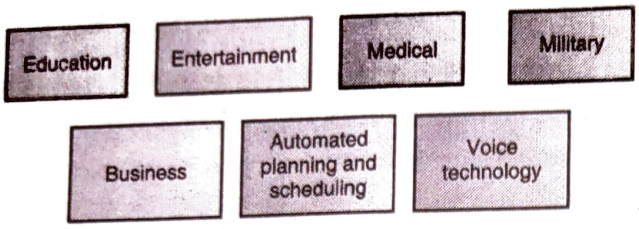


Fig. 1.3.1 : Fields of AI Application

- Fig. 1.3.1 shows few fields in which we have applications of artificial intelligence. There can be many fields in which Artificially Intelligent Systems can be used.

1. **Education**

Training simulators can be built using artificial intelligence techniques. Software for pre-school children are developed to enable learning with fun games. Automated grading, Interactive tutoring, instructional theory are the current areas of application.

2. **Entertainment**

Many movies, games, robots are designed to play as a character. In games they can play as an opponent when human player is not available or not desirable.

3. **Medical**

AI has applications in the field of cardiology (CRG), Neurology (MRI), Embryology (Sonography), complex operations of internal organs, etc. It can be also used in organizing bed schedules, managing staff rotations, store and retrieve information of patient. Many expert systems are enabled to predict the decease and can provide with medical prescriptions.

4. **Military**

Training simulators can be used in military applications. Also areas where human cannot reach or in life stacking conditions, robots can be very well used to do the required jobs. When decisions have to be made quickly taking into account an enormous amount of information, and when lives are at stake, artificial intelligence can provide crucial assistance. From developing intricate flight plans to implementing complex supply systems or creating training simulation exercises, AI is a natural partner in the modern military.

5. **Business and Manufacturing**

Latest generation of robots are equipped well with the performance advances, growing integration of vision and an enlarging capability to transform manufacturing.

6. **Automated planning and scheduling**

Intelligent planners are available with AI systems, which can process large datasets and can consider all the constraints to design plans satisfying all of them.

7. **Voice Technology**

Voice recognition is improved a lot with AI. Systems are designed to take voice inputs which are very much applicable in case of handicaps. Also scientists are developing an intelligent machine to emulate activities of a skillful musician. Composition, performance, sound processing, music theory are some of the major areas of research.

8. **Heavy Industry**

Huge machines involve risk in operating and maintaining them. Human robots are better placing human operators. These robots are safe and efficient. Robot are proven to be effective as compare to human in the jobs of repetitive nature, human may fail due to lack of continuous attention or laziness.

1.3.1 Current Trends in Artificial Intelligence

Artificial Intelligence has touched each and every aspect of our life. From washing machine, Air conditioners, to smart phones everywhere AI is serving to ease our life. In industry, AI is doing marvellous work as well. Robots are doing the sound work in factories. Driverless cars have become a reality. WiFi-enabled Barbie uses speech-recognition to talk and listen to children. Companies are using AI to improve their product and increase

sales. AI saw significant advances in machine learning. Following are the areas in which AI is showing significant advancements.

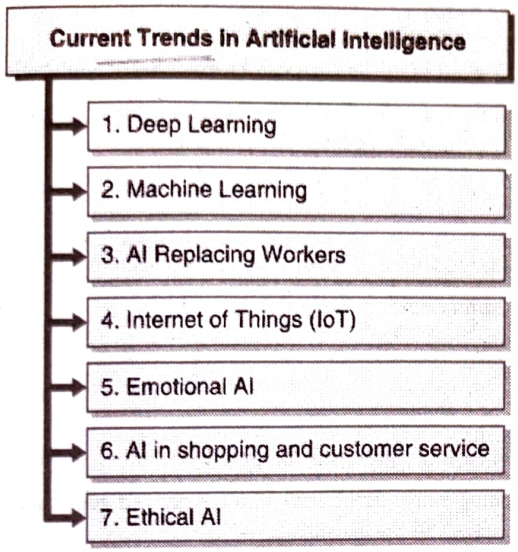


Fig. 1.3.2 : Current Trends in Artificial Intelligence

➜ **1. Deep Learning**

Convolutional Neural Networks enabling the concept of deep learning is the top most area of focus in Artificial intelligence in todays' era. Many problems and applications areas of AI like, natural language and text processing, speech recognition, computer vision, information retrieval, and multimodal information processing empowered by multi-task deep learning.

➜ **2. Machine Learning**

The goal of machine learning is to program computers to use example data or past experience to solve a given problem. Many successful applications of machine learning include systems that analyse past sales data to predict customer behaviour, optimize robot behaviour so that a task can be completed using minimum resources, and extract knowledge from bioinformatics data.

➜ **3. AI Replacing Workers**

In industry where there are safety hazards, robots are doing a good job. Human resources are getting replaced by robots rapidly. People are worries to see that the white color jobs of data processing are being done exceedingly well by intelligent programs. A study from The National Academy of Sciences brought together technologists and economists and social scientists to figure out what's going to happen.

➜ **4. Internet of Things (IoT)**

The concepts of smarter homes, smarter cars and smarter world is evolving rapidly with the invention of internet of things. The future is no far when each and every object will be wirelessly connected to something in order to perform some smart actions without any human instructions or interference. The worry is how the mined data can potentially be exploited.

➜ **5. Emotional AI**

Emotional AI, where AI can detect human emotions, is another upcoming and important area of research. Computers' ability to understand speech will lead to an almost seamless interaction between human and computer. With increasingly accurate cameras, voice and facial recognition, computers are better able to detect our emotional state. Researchers are exploring how this new knowledge can be used in education, to treat depression, to accurately predict medical diagnoses, and to improve customer service and shopping online.

➜ **6. AI in shopping and customer service**

Using AI, customers' buying patterns, behavioral patterns can be studied and systems that can predict the purchase or can help customer to figure out the perfect item. AI cab be used to find out what will make the customer happy or unhappy. For example, if a customer is shopping online, like a dress pattern but needs dark shades and thick material, computer understand the need and brings out new set of perfectly matching clothing for him.

➜ **7. Ethical AI**

With all the evolution happening in technology in every walk of life, ethics must be considered at the forefront of research. For example, in case of driverless car, while driving, if the decision has to be made between weather to dash a cat or a lady having both in an uncontrollable distance in front of the car, is an ethical decision. In such cases how the programming should decide who is more valuable, is a question. These are not the problems to be solved by computer engineers or research scientists but someone has to come up with an answer.

Syllabus Topic : State Space Search

1.4 State Space Search

- Given a goal to achieve; problem formulation is the process of deciding what states to be considered and what actions to be taken to achieve the goal. This is the first step to be taken by any problem solving agent.
- **State space :** The state space of a problem is the set of all states reachable from the initial state by executing any sequence of actions. State is representation of all possible outcomes.
- The state space specifies the relation among various problem states thereby, forming a directed network or graph in which the nodes are states and the links between nodes represent actions.
- **State Space Search :** Searching in a given space of states pertaining to a problem under consideration is called a state space search.
- **Path :** A path is a sequence of states connected by a sequence of actions, in a given state space.

1.4.1 Components of Problems Formulation

Q. 1.4.1 What are the components of problem formulation? (Refer section 1.4.1) **(8 Marks)**

Q. 1.4.2 Explain steps in problem formulation with example. (Refer section 1.4.1) **(8 Marks)**

Problem can be defined formally using five components as follows :

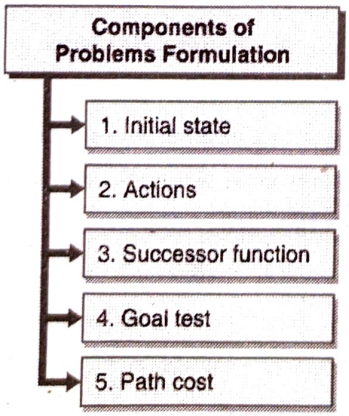


Fig. 1.4.1 : Components of Problems Formulation

➜ **1. Initial state**

The initial state is the one in which the agent starts in.

➜ **2. Actions**

It is the set of actions that can be executed or applicable in all possible states. A description of what each action does; the formal name for this is the transition model.

➜ **3. Successor function**

It is a function that returns a state on executing an action on the current state.

➜ **4. Goal test**

- It is a test to determine whether the current state is a goal state. In some problems the goal test can be carried out just by comparing current state with the defined goal state, called as **explicit goal test**. Whereas, in some of the problems, state cannot be defined explicitly but needs to be generated by carrying out some computations, it is called as **implicit goal test**.
- **For example :** In Tic-Tac-Toe game making diagonal or vertical or horizontal combination declares the winning state which can be compared explicitly; but in the case of chess game, the goal state cannot be predefined but it's a scenario called as "Checkmate", which has to be evaluated implicitly.

➜ **5. Path cost**

It is simply the cost associated with each step to be taken to reach to the goal state. To determine the cost to reach to each state, there is a cost function, which is chosen by the problem solving agent.

☞ **Problem solution**

- A well-defined problem with specification of initial state, goal test, successor function, and path cost. It can be represented as a data structure and used to implement a program which can search for the goal state.
- A solution to a problem is a sequence of actions chosen by the problem solving agent that leads from the initial state to a goal state. Solution quality is measured by the path cost function.

☞ **Optimal solution**

- An optimal solution is the solution with least path cost among all solutions.

- A general sequence followed by a simple problem solving agent is, first it formulates the problem with the goal to be achieved, then it searches for a sequence of actions that would solve the problem, and then executes the actions one at a time.

1.4.2 Example of 8-Puzzle Problem

Q. 1.4.3 Formulate 8-puzzle problem. (Refer section 1.4.2) **(8 Marks)**

- Fig. 1.4.2 depicts a typical scenario of 8-puzzle problem. It has a 3 × 3 board with tiles having 1 through 8 numbers on it. There is a blank tile which can be moved forward, backward, to left and to right. The aim is to arrange all the tiles in the goal state form by moving the blank tile minimum number of times.

1	2	3
4	8	–
7	6	5

Initial State

1	2	3
4	5	6
7	8	–

Goal State

Fig. 1.4.2 : A scenario of 8-Puzzle Problem

This problem can be formulated as follows :

States : States can represented by a 3 × 3 matrix data structure with blank denoted by 0.

1. **Initial state :** {{1, 2, 3},{4, 8, 0},{7, 6, 5}}
2. **Actions :** The blank space can move in Left, Right, Up and Down directions specifying the actions.
3. **Successor function :** If we apply "Down" operator to the start state in Fig. 1.4.2, the resulting state has the 5 and the blank switching their positions.
4. **Goal test :** {{1, 2, 3},{4, 5, 6},{7, 8, 0}}
5. **Path cost :** Number of steps to reach to the final state.

Solution :

{{1, 2, 3},{4, 8, 0},{7, 6, 5}}→ {{1, 2, 3},{4, 8, 5},

{7, 6, 0}} →{{1, 2, 3},

{4, 8, 5},{7, 0, 6}} →{{1, 2, 3},{4, 0, 5},

{7, 8, 6}}→{{1, 2, 3},{4, 5, 0},

{7, 8, 6}} →{{1, 2, 3},{4, 5, 6},{7, 8, 0}}

Path cost = 5 steps

1.4.3 Example of Missionaries and Cannibals Problem

- The problem statement as discussed in the previous section. Let's formulate the problem first.
- **States :** In this problem, state can be data structure having triplet (i, j, k) representing the number of missionaries, cannibals, and canoes on the left bank of the river respectively.

1. **Initial state :** It is (3, 3, 1), as all missionaries, cannibals and canoes are on the left bank of the river.
2. **Actions :** Take x number of missionaries and y number of cannibals.
3. **Successor function :** If we take one missionary, one cannibal the other side of the river will have two missionaries and two cannibals left.
4. **Goal test :** Reached state (0, 0, 0)
5. **Path cost :** Number of crossings to attain the goal state.

Solution :

The sequence of actions within the path :

(3,3,1) → (2,2,0) → (3,2,1) → (3,0,0) → (3,1,1) → (1,1,0) → (2,2,1) → (0,2,0) → (0,3,1) → (0,1,0) → (0,2,1) → (0,0,0)

Cost = 11 crossings

1.4.4 Vacuum-Cleaner Problem

States : In vacuum cleaner problem, state can be represented as [<block>, clean] or [<block>, dirty]. The agent can be in one of the two blocks which can be either clean or dirty. Hence there are total 8 states in the vacuum cleaner world.

1. **Initial State :** Any state can be considered as initial state. For example, [A, dirty]
2. **Actions :** The possible actions for the vacuum cleaner machine are left, right, absorb, idle.
3. **Successor function :** Fig. 1.4.3 indicating all possible states with actions and the next state.
4. **Goal Test :** The aim of the vacuum cleaner is to clean both the blocks. Hence the goal test if [A, Clean] and [B, Clean].
5. **Path Cost :** Assuming that each action/ step costs 1 unit cost. The path cost is number of actions/ steps taken.

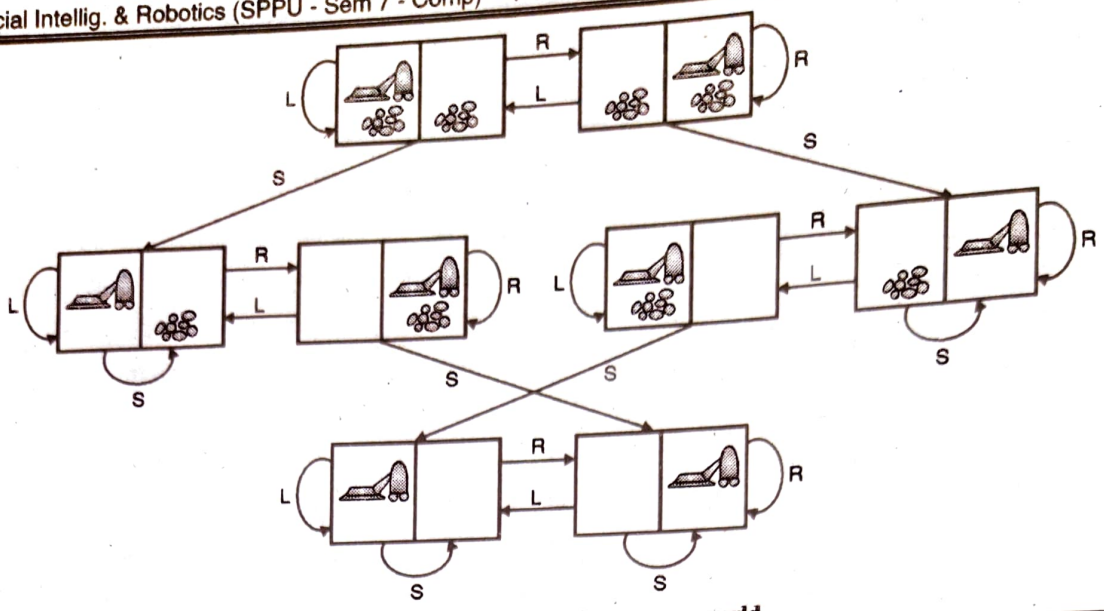


Fig. 1.4.3 : The state space for vacuum world

1.4.5 Example of Real Time Problems

- There are varieties of real time problems that can be formulated and solved by searching. Robot Navigation, Rout Finding Problem, Traveling Salesman Problem (TSP), VLSI design problem, Automatic Assembly Sequencing, etc. are few to name.
- There are number of applications for route finding algorithms. Web sites, car navigation systems that provide driving directions, routing video streams in computer networks, military operations planning, and airline travel-planning systems are few to name. All these systems involve detailed and complex specifications.
- For now, let us consider a problem to be solved by a travel planning web site; the airline travel problem.
- **State :** State is represented by airport location and current date and time. In order to calculate the path cost state may also record more information about previous segments of flights, their fare bases and their status as domestic or international.

1. **Initial state :** This is specified by the user's query, stating initial location, date and time.
2. **Actions :** Take any flight from the current location, select seat and class, leaving after the current time, leaving enough time for within airport transfer if needed.
3. **Successor function :** After taking the action i.e. selecting fight, location, date, time; what is the next location date and time reached is denoted by the successor function. The location reached is considered as the current location and the flight's arrival time as the current time.
4. **Goal test :** Is the current location the destination location?
5. **Path cost :** In this case path cost is a function of monetary cost, waiting time, flight time, customs and immigration procedures, seat quality, time of day, type of airplane, frequent-flyer mileage awards and so on.

1.4.6 Measuring Performance of Problem Solving Algorithm / Agent

There are variety of problem solving methods and algorithms available in AI. Before studying any of these algorithms in detail, let's consider the criteria to judge the efficiency of those algorithms. The performance of all these algorithms can be evaluated on the basis of following factors.

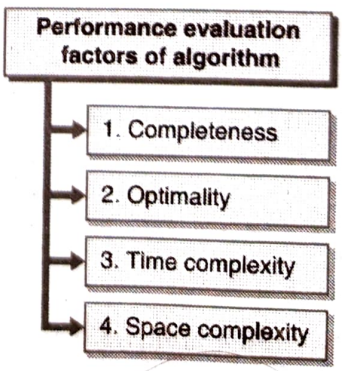


Fig. 1.4.4 : Performance evaluation factors of Algorithm

➜ **1. Completeness**

If the algorithm is able to produce the solution if one exists then it satisfies completeness criteria.

➜ **2. Optimality**

If the solution produced is the minimum cost solution, the algorithm is said to be optimal.

➜ **3. Time complexity**

It depends on the time taken to generate the solution. It is the number of nodes generated during the search.

➜ **4. Space complexity**

Memory required to store the generated nodes while performing the search.

Complexity of algorithms is expressed in terms of three quantities as follows :

1. **b :** Called as branching factor representing maximum number of successors a node can have in the search tree.
2. **d :** Stands for depth of the shallowest goal node.
3. **m :** It is the maximum depth of any path in the search tree.

Syllabus Topic : Depth Bounded DFS

1.5 Depth Bounded DFS

☞ **Concept**

- In order to avoid the infinite loop condition arising in DFS, in depth limited search technique, depth-first search is carried out with a predetermined depth limit.
- The nodes with the specified depth limit are treated as if they don't have any successors. The depth limit solves the infinite-path problem.
- But as the search is carried out only till certain depth in the search tree, it introduces problem of incompleteness.
- Depth-first search can be viewed as a special case of depth-limited search with depth limit equal to the depth of the tree. The process of DLS is depicted in Fig. 1.5.1.

1.5.1 Process

If depth limit is fixed to 2, DLS carries out depth first search till second level in the search tree.

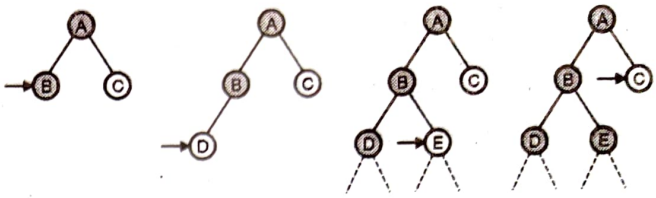


Fig. 1.5.1 : DLS working with depth limit

1.5.2 Implementation

- As in case of DFS in DLS we can use the same fringe implemented as queue.
- Additionally the level of each node needs to be calculated to check whether it is within the specified depth limit.
- Depth-limited search can terminate with two conditions :
  1. If the solution is found.
  2. If there is no solution within given depth limit.

1.5.3 Algorithm

- Determine the start node and the search depth.
- Check if the current node is the goal node
  - **If not :** Do nothing
  - **If yes :** return
- Check if the current node is within the specified search depth
  - **If not :** Do nothing
  - **If yes :** Expand the node and save all of its successors in a stack.
- Call DLS recursively for all nodes of the stack and go back to Step 2.

1.5.4 Pseudo Code

```
booleanDLS(Node node, int limit, intdepth)
{
    if (depth > limit) return failure;
    if (node is a goal node) return success;
    for each child of node
```

```
    {
        if (DLS(child, limit, depth + 1))
        return success;
    }
    return failure;
}
```

1.5.5 Performance Evaluation

- **Completeness :** Its incomplete if shallowest goal is beyond the depth limit.
- **Optimality :** Non optimal, as the depth chosen can be greater than d.
- **Time complexity :** Same as DFS, $O(b^l)$, where l is the specified depth limit.
- **Space complexity :** Same as DFS, $O(b^l)$, where l is the specified depth limit.

Syllabus Topic : Depth First Iterative Deepening

1.6 Depth First Iterative Deepening (DFID)

1.6.1 Concept

- Iterative deepening depth first search is a combination of BFS and DFS. In DFID search happens depth wise but, at a time the depth limit will be incremented by one. Hence iteratively it deepens down in the search tree.
- It eventually turns out to be the breadth-first search as it explores a complete layer of new nodes at each iteration before going on to the next layer.
- It does this by gradually increasing the depth limit-first 0, then 1, then 2, and so on-until a goal is found; and thus guarantees the optimal solution. Iterative deepening combines the benefits of depth-first and breadth-first search. The search process is depicted in Fig. 1.6.1.
- Fig. 1.6.1 shows four iterations of on a binary search tree, where the solution is found on the fourth iteration.

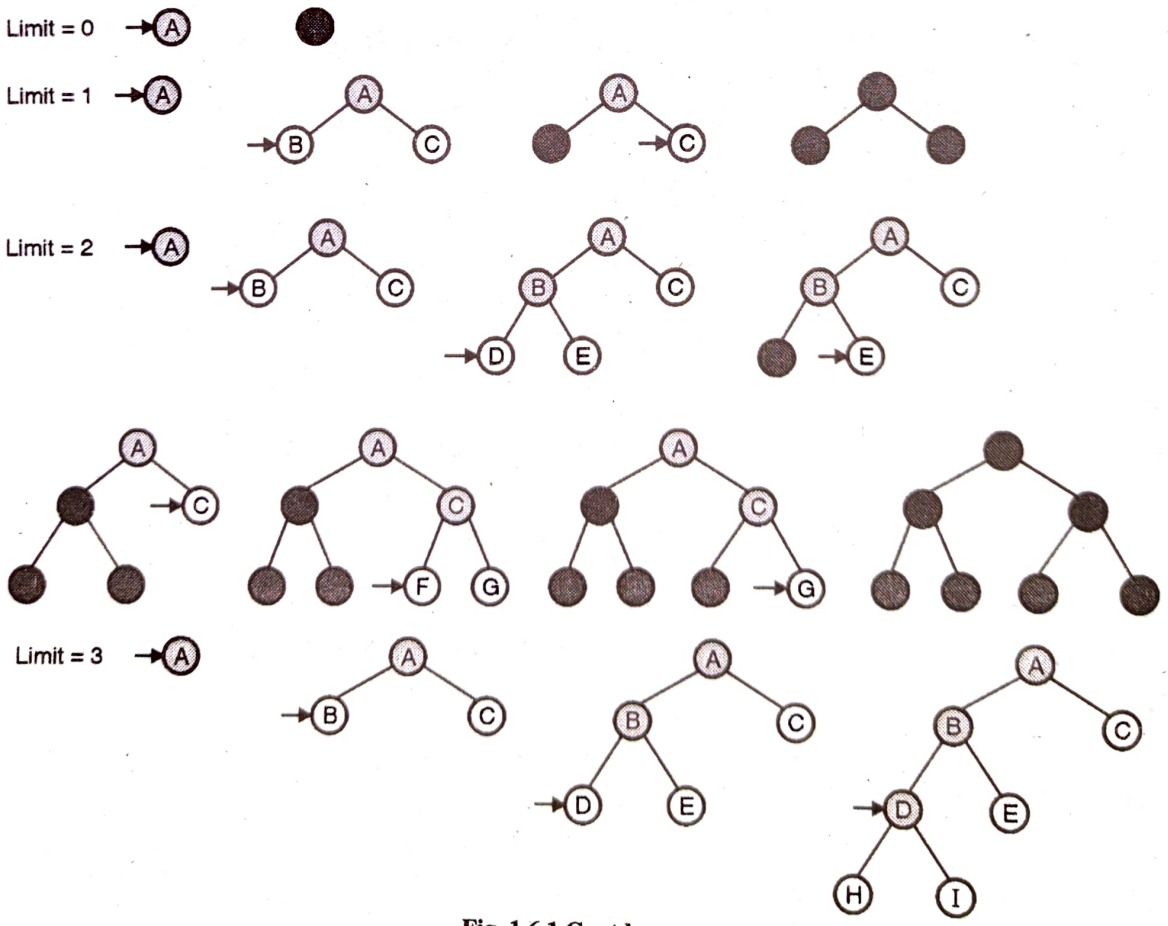


Fig. 1.6.1 Contd...

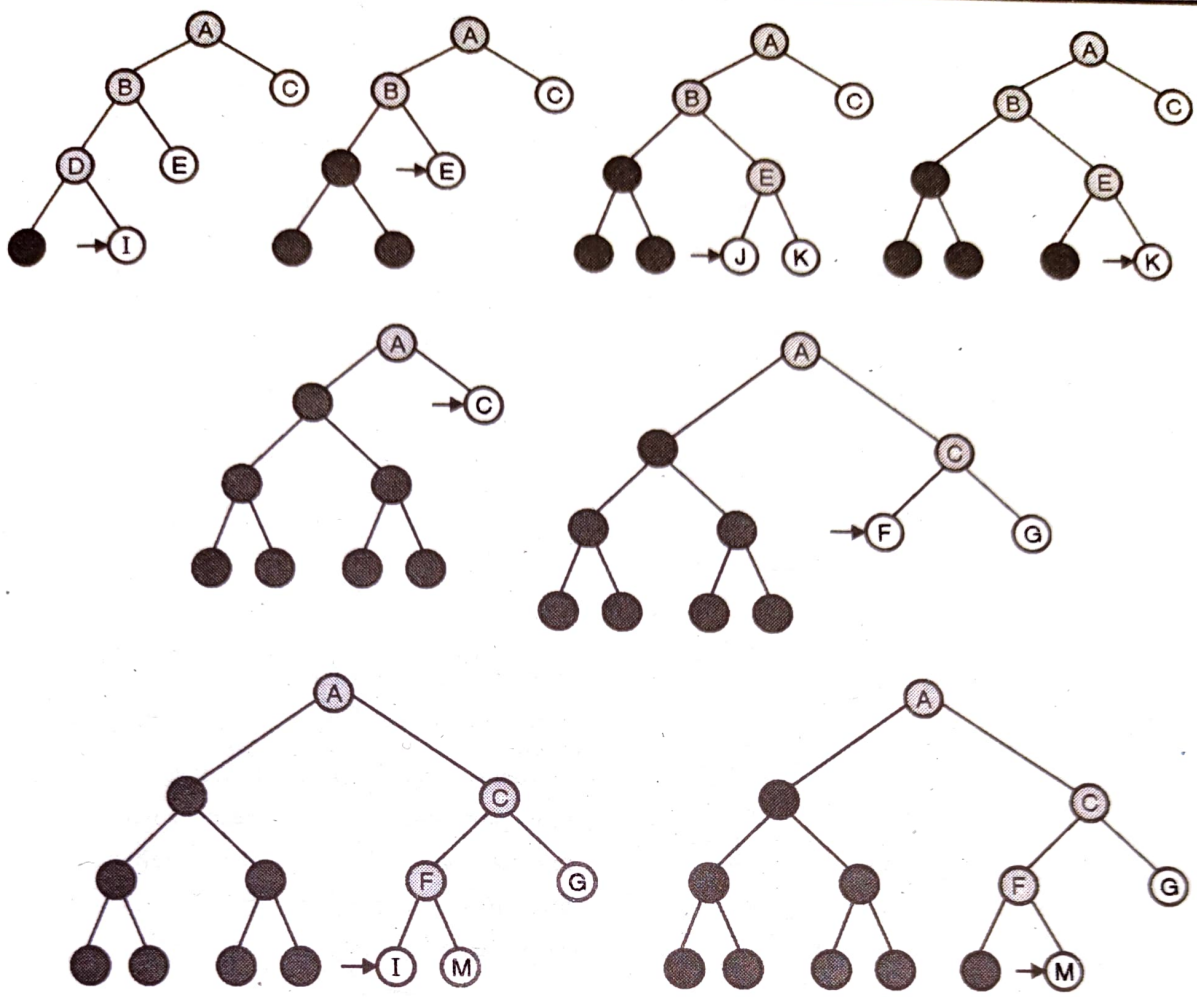


Fig. 1.6.1 : Search process in DFID

1.6.2 Process

- **Function 1** iterative : Depending search (problem) **returns** a solution, or failure

```
for depth = 0 to ∞ do
result ← Depth – Limited – Search (problem, depth)
if result ≠cutoff
then return result
```

- Fig. 1.6.1 the iterative depending search algorithm, which repeatedly applies depth limited search with increasing limits. It terminates when a solution is found or if the depth limited search returns failure, meaning that no solution exists.

1.6.3 Implementation

It has exactly the same implementation as that of DLS. Additionally, iterations are required to increment the depth limit by one in every recursive call of DLS.

1.6.4 Algorithm

- Initialize depth limit to zero.
- Repeat Until the goal node is found.
  - (a) Call Depth limited search with new depth limit.
  - (b) Increment depth limit to next level.

1.6.5 Pseudo Code

```
DFID()
{
    limit = 0;
    found = false;
    while (not found)
    {
        found = DLS(root, limit, 0);
        limit = limit + 1;
    }
}
```

1.6.6 Performance Evaluation

☞ **Completeness**

DFID is complete when the branching factor b is finite.

☞ **Optimality**

It is optimal when the path cost is a non-decreasing function of the depth of the node.

☞ **Time complexity**

- Do you think in DFID there is a lot of wastage of time and memory in regenerating the same set of nodes again and again?
- It may appear to be waste of memory and time, but it's not so. The reason is that, in a search tree with almost same branching factor at each level, most of the nodes are in the bottom level which are explores very few times as compared to those on upper level.
- The nodes on the bottom level that is level 'd' are generated only once, those on the next to bottom level are generated twice, and so on, up to the children of the root, which are generated d times. Hence the time complexity is $O(b^d)$.

☞ **Space complexity**

Memory requirements of DFID are modest, i.e. $O(b^d)$.

Note : As the performance evaluation is quite satisfactory on all the four parameters, DFID is the preferred uninformed search method when the search space is large and the depth of the solution is not known.

Syllabus Topic : Heuristic Search

1.7 Heuristic Search

- Informed searching techniques or Heuristic search is a further extension of basic un-informed search techniques. The main idea is to generate additional information about the search state space using the knowledge of problem domain, so that the search becomes more intelligent and efficient. The evaluation function is developed for each state, which quantities the desirability of expanding that state in order to reach the goal.
- All the strategies use this evaluation function in order to select the next state under consideration, hence the name "Informed Search". These techniques are very much efficient with respect to time and space requirements as compared to uninformed search techniques.

Syllabus Topic : Heuristic Functions

1.7.1 Heuristic Functions

- A heuristic function is an evaluation function, to which the search state is given as input and it generates the tangible representation of the state as output.
- It maps the problem state description to measures of desirability, usually represented as number weights. The value of a heuristic function at a given node in the search process gives a good estimate of that node being on the desired path to solution.
- It evaluates individual problem state and determines how much promising the state is. Heuristic functions are the most common way of imparting additional knowledge of the problem states to the search algorithm. Fig. 1.7.1 shows the general representation of heuristic function.

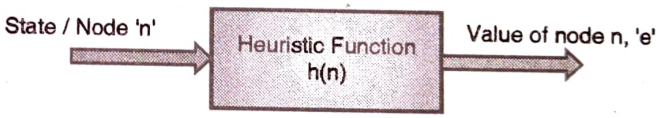


Fig. 1.7.1 : General representation of Heuristic function

- The representation may be the approximate cost of the path from the goal node or number of hopes required to reach to the goal node, etc.
- The heuristic function that we are considering in this syllabus, for a node n is, h(n) = estimated cost of the cheapest path from the state at node n to a goal state.

☞ **Example**

- For the Travelling Salesman Problem, the sum of the distances traveled so far can be a simple heuristic function.
- Heuristic function can be of two types depending on the problem domain. It can be a **Maximization Function** or **Minimization function** of the path cost.
- In maximization types of heuristic, greater the cost of the node, better is the node while; in case of minimization heuristic, lower is the cost, better is the node. There are

heuristics of every general applicability as well as domain specific. The search strategies are general purpose heuristics.

- It is believed that in general, a heuristic will always lead to faster and better solution, even though there is no guarantee that it will never lead in the wrong direction in the search tree.
- Design of heuristic plays a vital role in performance of search.
- As the purpose of a heuristic function is to guide the search process in the most profitable path among all that are available; a well designed heuristic functions can provides a fairly good estimate of whether a path is good or bad.
- However in many problems, the cost of computing the value of a heuristic function would be more than the effort saved in the search process. Hence generally there is a trade-off between the cost of evaluating a heuristic function and the savings in search that the function provides.
- So, are you ready to think of your own heuristic function definitions? Here is the word of caution. See how the function definition impact.
- Following are the examples demonstrate how design of heuristic function completely alters the scenario of searching process.

1.7.2 Example of 8-puzzle Problem

- Remember 8-puzzle problem? Can we estimate the number of steps required to solve an 8-puzzle from a given state?? What about designing a heuristic function for it?

7	5	4
5		6
8	3	1

Start state

	1	2
3	4	5
6	7	8

Goal State

Fig. 1.7.2 : A scenario of 8-puzzle problem

- Two simple heuristic functions are :
  - h_1 = the number of misplaced tiles. This is also known as the Hamming Distance. In the Fig. 1.7.2 example, the start state has $h_1 = 8$. Clearly, h_1 is an acceptable heuristic because any tile that is out of place will have to be moved at least once, quite logical. Isn't it?
  - h_2 = the sum of the distances of the tiles from their goal positions. Because tiles cannot be moved diagonally, the distance counted is the sum of horizontal and vertical distances. This is also known as the Manhattan Distance. In the Fig. 1.7.2, the start state has $h_2 = 3 + 1 + 2 + 2 + 2 + 3 + 3 + 2 = 18$. Clearly, h_2 is also an admissible heuristic because any move can, at best, move one tile one step closer to the goal.
- As expected, neither heuristic overestimates the true number of moves required to solve the puzzle, which is 26 ($h_1 + h_2$).
- Additionally, it is easy to see from the definitions of the heuristic functions that for any given state, h_2 will always be greater than or equal to h_1. Thus, we can say that h_2 dominates h_1.

1.7.3 Example of Block World Problem

Q. 1.7.1 Find the heuristics value for a particular state of the blocks world problem. (Refer section 1.7.3) **(8 Marks)**

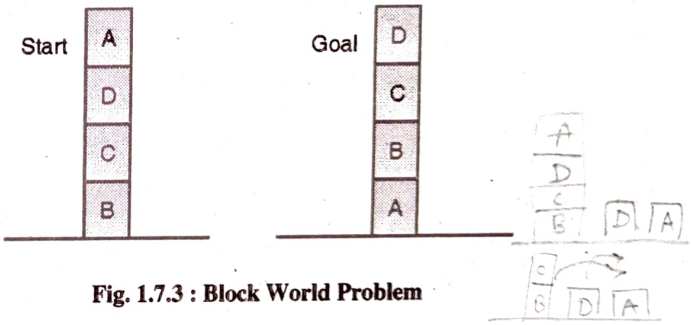


Fig. 1.7.3 : Block World Problem

- Fig. 1.7.3 depicts a block problem world, where the A, B, C, D letter bricks are piled up on one another and required to be arranged as shown in goal state, by moving one brick at a time. As shown, the goal state with the particular arrangement of blocks need to be attain from the given start state. Now it's time to scratch your head and define a heuristic function that will distinguish start state from goal state. Confused??
- Let's design a function which assigns + 1 for the brick at right position and – 1 for the one which is at wrong position. Consider Fig. 1.7.4.

Local heuristic

- + 1 for each block that is resting on the thing it is supposed to be resting on.

- – 1 for each block that is resting on a wrong thing.

Fig. 1.7.4 : Heuristic Function "h_1"

- Fig. 1.7.5 shows the heuristic values generated by heuristic function "h_1" for various different states in the state space. Please observe that, this heuristic is generating same value for different states.

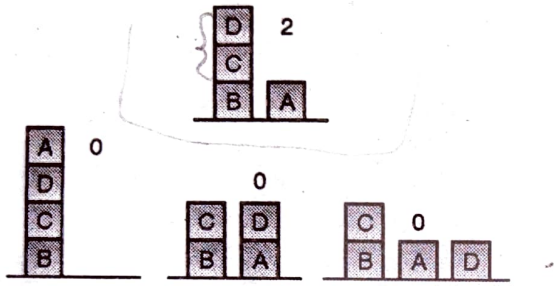


Fig. 1.7.5 : State evaluations using Heuristic function "h_1"

- Due to this kind of heuristic the search may end up in limitless iterations as the state showing most promising heuristic value may not hold true or search may end up in finding an undesirable goal state as the state evaluation may lead to wrong direction in the search tree.
- Let's have another heuristic design for the same problem. Fig. 1.7.6 is depicting a new heuristic function "h_2" definition, in which the correct support structure of each brick is given +1 for each brick in the support structure. And the one not having correct support structure, –1 for each brick in the wrong support structure.

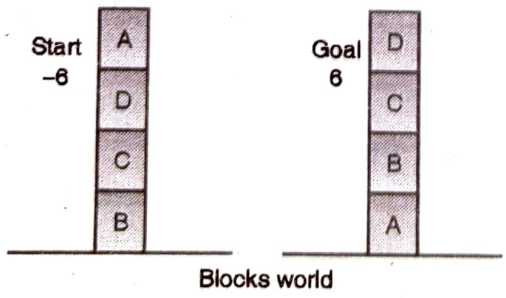


Fig. 1.7.6 : heuristic function "h_2"

- As we observe in Fig. 1.7.7, the same states are considered again as that of Fig. 1.7.5, but this time using h_2, each one of the state is assigned a unique value generate according to heuristic function h_2.
- Observing this example one can easily understand that, in the second part of the example, search will be carried out smoothly as each unique state is getting a unique value assigned to it.
- This example makes it clear that, the design of heuristic plays a vital role in search process, as the whole search is carried out by considering the heuristic values as basis for selecting the next state to be explored.
- The state having the most promising value to reach to the goal state will be the first prior candidate for exploration, this continues till we find the goal state.

☞ **Global heuristic**

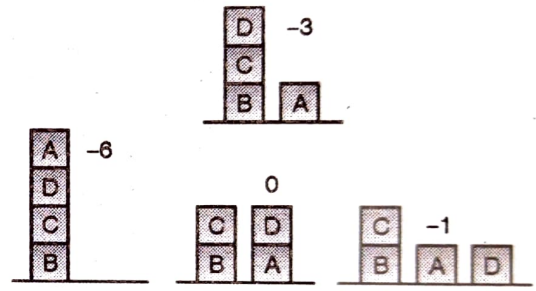


Fig. 1.7.7 : State evaluations using Heuristic function "h_2"

- For each block that has the correct support structure : + 1 to every block in the support structure.
- For each block that has the wrong support structure : – 1 to every block in the support structure.
- This leads to a discussion of a better heuristic function definition.
- Is there any particular way of defining a heuristic function that will guarantee a better performance in search process??

1.7.4 Properties of Good Heuristic Function

1. It should generate a unique value for each unique state in search space.
2. The values should be a logical indicator of the profitability of the state in order to reach the goal state.
3. It may not guarantee to find the best solution, but almost always should find a very good solution.

4. It should reduce the search time; specifically for hard problems like travelling salesman problem where the time required is exponential.

- The main objective of a heuristic is to produce a solution in a reasonable time frame that is good enough for solving the problem, as it's an extra task added to the basic search process.
- The solution produced by using heuristic may not be the best of all the actual solutions to this problem, or it may simply approximate the exact solution. But it is still valuable because finding the solution does not require a prohibitively long time. So we are investing some amount of time in generating heuristic values for each state in search space but reducing the total time involved in actual searching process.
- Do we require to design heuristic for every problem in real world? There is a trade-off criterion for deciding whether to use a heuristic for solving a given problem. It is as follows.
  - **Optimality :** Does the problem require to find the optimal solution, if there exist multiple solutions for the same?
  - **Completeness :** In case of multiple existing solution of a problem, is there a need to find all of them? As many heuristics are only meant to find one solution.
  - **Accuracy and precision :** Can the heuristic guarantee to find the solution within the precision limits? Is the error bar on the solution unreasonably large?
  - **Execution time :** Is it going to affect the time required to find the solution? Some heuristics converge faster than others. Whereas, some are only marginally quicker than classic methods.
- In many AI problems, it is often hard to measure precisely the goodness of a particular solution. But still it is important to keep performance question in mind while designing algorithm. For real world problems, it is often useful to introduce heuristics based on relatively unstructured knowledge. It is impossible to define this knowledge in such a way that mathematical analysis can be performed.

Syllabus Topic : Best First Search

1.8 Best First Search

Concept

Q. 1.8.1 Explain search strategy to overcome drawbacks of BFS and DFS. (Refer section 1.8) **(8 Marks)**

- In depth first search all competing branches are not getting expanded. And breadth first search never gets trapped on dead end paths. If we combine these properties of both DFS and BFS, it would be "follow a single path at a time, but switch paths whenever some competing path look more promising than the current one". This is what the Best First search is..!!
- Best-first search is a search algorithm which explores the search tree by expanding the most promising node chosen according to the heuristic value of nodes.
- **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".
- Efficient selection of the current best candidate for extension is typically implemented using a priority queue. Fig. 1.8.1 depicts the search process of Best first search on an example search tree. The values noted below the nodes are the estimated heuristic values of nodes.

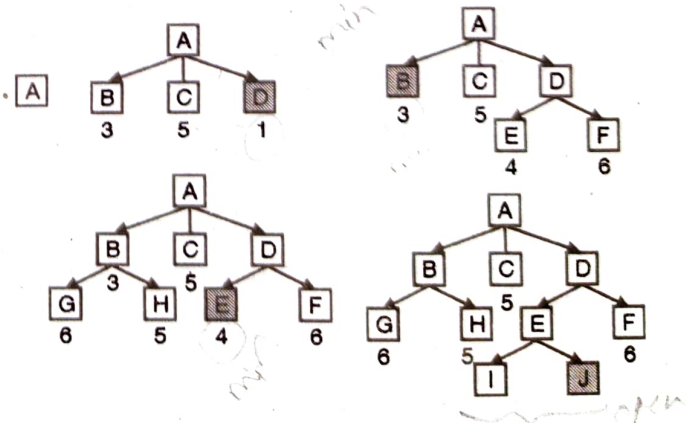


Fig. 1.8.1 : Best first search tree expansion scenario

1.8.1 Implementation

- Best first search uses two lists in order to record the path. These are namely OPEN list and CLOSED list for implementation purpose.
- OPEN list stores nodes that have been generated, but have not examined. This is organized as a priority queue, in which nodes are stored with the increasing order of their heuristic value, assuming we are implementing maximization heuristic. It provides efficient selection of the current best candidate for extension.
- CLOSED list stores nodes that have already been examined. This CLOSED list contains all nodes that have been evaluated and will not be looked at again. Whenever a new node is generated, check whether it has been generated before. If it is already visited before, check its recorded value and change the parent if this new value is better than previous one. This will avoid any node being evaluated twice, and will never get stuck into an infinite loops.

1.8.2 Algorithm : Best First Search (BFS)

OPEN = [initial state]

CLOSED = []

while OPEN is not empty

do

1. Remove the best node from OPEN, call it n, add it to CLOSED.
2. If n is the goal state, backtrack path to n through recorded parents and return path.
3. Create n's successors.
4. For each successor do:
   a. If it is not in CLOSED and it is not in OPEN: evaluate it, add it to OPEN, and record its parent.
   b. Otherwise, if it is already present in OPEN with different parent node and this new path is better than previous one, change its recorded parent.
      i. If it is not in OPEN add it to OPEN.
      ii. Otherwise, adjust its priority in OPEN using this new evaluation.

done

This algorithm of Best First Search algorithm just terminates when no path is found. An actual implementation would of course require special handling of this case.

1.8.3 Performance Measures for Best First Search

- **Completeness :** Not complete, may follow infinite path if heuristic rates each state on such a path as the best option. Most reasonable heuristics will not cause this problem however.
- **Optimality :** Not optimal; may not produce optimal solution always.
- **Time Complexity :** Worst case time complexity is still O(bm) where m is the maximum depth.
- **Space Complexity :** Since must maintain a queue of all unexpanded states, space-complexity is also O(bm).

1.8.4 Greedy Best First Search

Q. 1.8.2 What is the difference between best first and greedy best first search ? (Refer sections 1.8 and 1.8.4) **(8 Marks)**

- A greedy algorithm is an algorithm that follows the heuristic of making the locally optimal choice at each stage with the hope of finding a global optimum.
- When Best First Search uses a heuristic that leads to goal node, so that nodes which seems to be more promising are expanded first. This particular type of search is called greedy best-first search.
- In greedy best first search algorithm, first successor of the parent is expanded.
- For the successor node, check the following :
  1. If the successor node's heuristic is better than its parent, the successor is set at the front of the queue, with the parent reinserted directly behind it, and the loop restarts.
  2. Else, the successor is inserted into the queue, in a location determined by its heuristic value. The procedure will evaluate the remaining successors, if any of the parent.
- In many cases, greedy best first search may not always produce an optimal solution, but the solution will be locally

optimal, as it will be generated in comparatively less amount of time. In mathematical optimization, greedy algorithms solve combinatorial problems.

Example

- Consider the traveling salesman problem, which is of a high computational complexity, works well with greedy strategy as follows. Refer to Fig. 1.8.2. The values written on the links are the straight line distances from the nodes. Aim is to visit all the cities A through F with the shortest distance travelled.
- Let us apply a greedy strategy for this problem with a heuristic as, "At each stage visit an unvisited city nearest to the current city". Simple logic... isn't it? This heuristic need not find a best solution, but terminates in a reasonable number of steps by finding an optimal solution which typically requires unreasonably many steps. Let's verify.

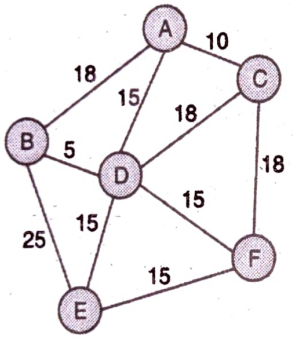


Fig. 1.8.2 : Travelling Salesmen Problem

- As greedy algorithm, it will always make a local optimal choice. Hence it will select node C first as it found to be the one with less distance from the next non-visited node from node A, and then the path generated will be A → C → D → B → E → F with the total cost = 10 + 18 + 5 + 25 + 15 = 73. While by observing the graph one can find the optimal path and optimal distance the salesman needs to travel. It turns out to be, A→B→D→E→F→C where the cost comes out to be 18 + 5 + 15 + 15 + 18 = 68.

1.8.5 Properties of Greedy Best-First Search

1. **Completeness :** It's not complete as, it can get stuck in loops, also is susceptible to wrong start and quality of heuristic function.
2. **Optimality :** It's not optimal; as it goes on selecting a single path and never checks for other possibilities.
3. **Time Complexity :** $O(b^m)$, but a good heuristic can give dramatic improvement.
4. **Space Complexity :** $O(b^m)$, it needs to keep all nodes in memory.

Syllabus Topic : Hill Climbing

1.9 Hill Climbing

Q. 1.9.1 Explain Hill climbing algorithm. Explain plateau, ridge and local maxima. (Refer sections 1.9 and 1.9.3) **(8 Marks)**

- Hill climbing is simply a combination of depth first with generate and test where a feedback is used here to decide on the direction of motion in the search space.
- Hill climbing technique is used widely in artificial intelligence, to solve solving computationally hard problems, which has multiple possible solutions.
- In the depth-first search, the test function will merely accept or reject a solution. But in hill climbing the test function is provided with a heuristic function which provides an estimate of how close a given state is to goal state.
- In Hill climbing, each state is provided with the additional information needed to find the solution, i.e. the heuristic value. The algorithm is memory efficient since it does not maintain the complete search tree. Rather, it looks only at the current state and immediate level states.
- For example, if you want to find a mall from your current location. There are n possible paths with different directions to reach to the mall. The heuristic function will just give you the distance of each path which is reaching to the mall, so that it becomes very simple and time efficient for you to reach to the mall.
- Hill climbing attempts to iteratively improve the current state by means of an evaluation function. "Consider all the possible states laid out on the surface of a landscape. The height of any point on the landscape corresponds to the evaluation function of the state at that point" (Russell & Norvig, 2003). Fig. 1.9.1 depicts the typical hill climbing scenario, where multiple paths are available to reach to the hill top from ground level.

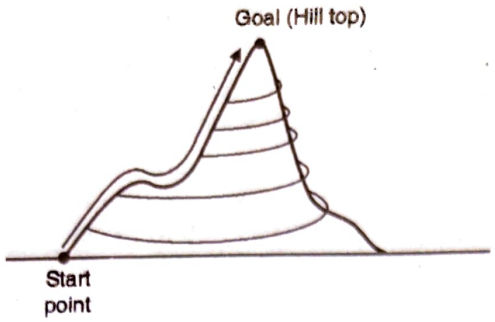


Fig. 1.9.1 : Hill Climbing Scenario

- Hill climbing always attempts to make changes that improve the current state. In other words, hill climbing can only advance if there is a higher point in the adjacent landscape.
- Hill climbing is a type of local search technique. It is relatively simple to implement. In many cases where state space is of moderate size, hill climbing works even better than many advanced techniques.
- For example, hill climbing when applied to travelling salesman problem; initially it produces random combinations of solutions having all the cities visited. Then it selects the better rout by switching the order, which visits all the cities in minimum cost.
- There are two variations of hill climbing as discussed follows.

1.9.1 Simple Hill Climbing

- It is the simplest way to implement hill climbing. Following is the algorithm for simple hill climbing technique. Overall the procedure looks similar to that of generate and test but, the main difference between the two is use of heuristic function for state evaluation which is used in hill climbing.
- The goodness of any state is decided by the heuristic value of that state. It can be either incremental heuristic or detrimental one.

Algorithm

1. Evaluate the initial state. If it is a goal state, then return and quit; otherwise make it a current state and go to Step 2.
2. Loop until a solution is found or there are no new operators left to be applied (i.e. no new children nodes left to be explored).
   a. Select and apply a new operator (i.e. generate new child node)
   b. Evaluate the new state
      (i) If it is a goal state, then return and quit.
      (ii) If it is better than current state then make it a new current state.
      (iii) If it is not better than the current state then continue the loop, go to Step 2.

As we studied the algorithm, we observed that in every pass the first node/state that is better than the current state is considered for further exploration. This strategy may not guarantee that most optimal solution to the problem, but may save upon the execution time.

1.9.2 Steepest Ascent Hill Climbing

Q. 1.9.2 Write algorithm of steepest ascent hill climbing and compare it with simple hill climbing. (Refer sections 1.9.1 and 1.9.2) **(8 Marks)**

As the name suggests, steepest hill climbing always finds the steepest path to hill top. It does so by selecting the best node among all children of the current node / state. All the states are evaluated using heuristic function. Obviously, the time requirement of this strategy is more as compared to the previous one. The algorithm for steepest ascent hill climbing is as follows.

Algorithm

1. Evaluate the initial state, if it is a goal state, return and quit; otherwise make it as a current state.
2. Loop until a solution is found or a complete iteration produces no change to current state :
   a. SUCC = a state such that any possible successor of the current state will be better than SUCC.
   b. For each operator that applies to the current state, evaluate the new state :
      (i) If it is goal; then return and quit.
      (ii) If it is better than SUCC then set SUCC to this state.
   c. SUCC is better than the current state → set the current state to SUCC.

- As we compare simple hill climbing with steepest ascent, we find that there is a tradeoff for the time requirement and the accuracy or optimality of the solution.

- In case of simple hill climbing technique as we go for first better successor, the time is saved as all the successors are not evaluated but it may lead to more number of nodes and branches getting explored, in turn the solution found may not be the optimal one.
- While in case of steepest ascent hill climbing technique, as every time the best among all the successors is selected for further expansion, it involves more time in evaluating all the successors at earlier stages, but the solution found will be always the optimal solution, as only the states leading to hill top are explored.
- This also makes it clear that the evaluation function i.e. the heuristic function definition plays a vital role in deciding the performance of the algorithm.

1.9.3 Limitations of Hill Climbing

Q. 1.9.3 Explain limitations of Hill Climbing in brief. (Refer section 1.9.3) **(8 Marks)**

Q. 1.9.4 What are the problems / frustrations that occur in hill climbing technique? Illustrate with an example. (Refer section 1.9.3) **(8 Marks)**

Q. 1.9.5 Write short note on Limitations of Hill-climbing algorithm. (Refer section 1.9.3) **(6 Marks)**

- Now let's see what can be the impact of incorrect design of heuristic function on the hill climbing techniques.
- Following are the problems that may arise in hill climbing strategy.
- Sometimes the algorithms may lead to a position, which is not a solution, but from which there is no move possible which will lead to a better place on hill i.e. no further state that is going closer to the solution. This will happen if we have reached one of the following three states.

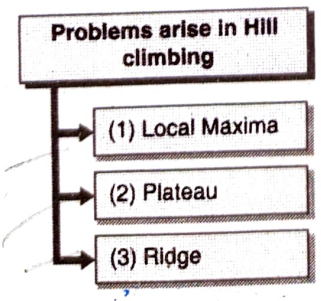


Fig. 1.9.2 : Problems arise in Hill climbing

(1) Local Maxima

- A "local maxima" is a location in hill which is at height from other parts of the hill but is not the actual hill top. In the search tree, it is a state better than all its neighbors, but there is not next better state which can be chosen for further expansion. Local maxima sometimes occur within sight of a solution. In such cases they are called "Foothills".

Fig. 1.9.3 : Local Maxima

- In the search tree local maxima can be seen as follows :

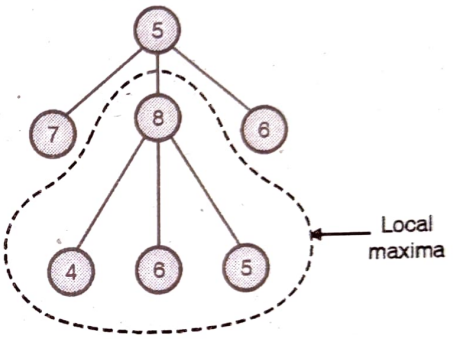


Fig. 1.9.4 : Local maxima in Search Tree

(2) Plateau

- A "plateau" is a flat area at some height in hilly region. There is a large area of same height in plateau. In the search space, plateau situation occurs when all the neighboring states have the same value. On a plateau, it is not possible to determine the best direction in which to move by making local comparisons.

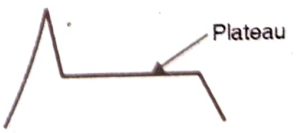


Fig. 1.9.5 : Plateau in hill climbing

- In the search tree plateau can be identified as follows :

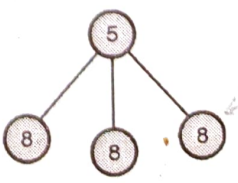


Fig. 1.9.6

Ridge : A "ridge" is an area in the hill such that, it is higher than the surrounding areas, but there is no further uphill path from ridge. In the search tree it is the situation, where all successors are either of same value or lesser, it's a ridge condition. The suitable successor cannot be searched in a simple move.

Fig. 1.9.7 : Ridge in Hill Climbing

- In the search tree ridge can be identified as follows :

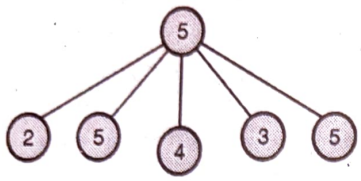


Fig. 1.9.8

- Fig. 1.9.9 depicts all the different situations together in hill climbing.

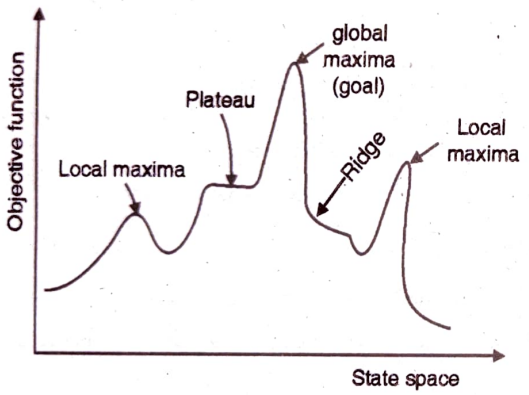


Fig. 1.9.9 : Hill climbing problems scenario

1.9.4 Solutions on Problems in Hill Climbing

Q. 1.9.6 Explain solution in brief for Problems in Hill Climbing. (Refer section 1.9.4) **(8 Marks)**

- In order to overcome these problems we can try following techniques.
- At times combination of two techniques will provide a better solution.

1. A good way to deal with local maximum, we can back track to some earlier nodes and try a different direction.
2. In case of plateau and ridges, make a big jump in some direction to a new area in the search. This can be done by applying two more rules of the same rule several times, before testing. This is a good strategy is dealing with plateau and ridges.

- Hill climbing is a local method. It decides what to do next by looking only at the "immediate" consequences of its choices. Global information might be encoded in heuristic functions. Hill climbing becomes inefficient in large problem spaces, and when combinatorial explosion occurs. But it uses very little memory, usually a constant amount, as it doesn't retain the path.
- It is a useful when combined with other methods. The success of hill climbing depends very much on the shape of the state space. If there are few local maxima and plateau, random-restart hill climbing will find a good solution very quickly.

Syllabus Topic : Variable Neighbourhood Descent Search

1.10 Variable Neighbourhood Descent Search

- It is a metaheuristic searching technique, used to solve global optimization problems. Variable neighbourhood search systematically changes the neighbourhood in two phases :
  - **Phase 1 :** descent to find a local optimum.
  - **Phase 2 :** a perturbation phase to get out of the corresponding valley.
- There are many applications in various fields like, location theory, cluster analysis, scheduling, vehicle routing, network design, lot-sizing, artificial intelligence, engineering, pooling problems, biology, phylogeny, reliability, geometry, telecommunication design, etc.
- This searching technique systematically performs the procedure of neighborhood change, both in descent to local minima and in escape from the valleys which contain them.
- Variable neighborhood search is built upon the following perceptions :

1. A local minimum with respect to one neighborhood structure is not necessarily a local minimum for another neighborhood structure.
2. A global minimum is a local minimum with respect to all possible neighborhood structures.

3. For many problems, local minima with respect to one or several neighborhoods are relatively close to each other.

- The variable neighborhood algorithm is simple and requires very few or sometimes no parameters. Therefore, in addition to providing very good solutions, often in simpler ways than other methods, this searching technique, gives insight into the reasons for such a performance, which, in turn, can lead to more efficient and sophisticated implementations.
- Following is the basic algorithm for variable neighborhood search. It combines deterministic and stochastic changes of neighborhood. Observe that point x' is generated at random in Step 4 in order to avoid cycling, which might occur if a deterministic rule were applied. In Step 5, the first improvement local search is applied and then it goes for neighborhood change depending upon the best neighborhood obtained by first improvement function.

☞ **Algorithm**

```
Function VNS (x, kmax, tmax );
1: repeat
2:   k ← 1;
3:   repeat
4:     x' ←Shake(x, k) /* Shaking */;
5:     x'' ← FirstImprovement(x') /* Local search */;
6:     x ←NeighbourhoodChange(x, x'', k)
         /* Change neighbourhood */;
7:   until k = k_max ;
8:   t ←CpuTime()
9: until t >t_max ;
```

Function FirstImprovement

```
Function FirstImprovement(x)
1: repeat
2:   x' ← x; i←0
3:   repeat
4:     i←i+1
5:     x←argmin{ f (x), f (x^i)}, x^i∈ N(x)
6:   until ( f (x) < f (x^i) or i = |N(x)|)
7: until ( f (x) ≥ f (x'))
8: return x
```

Function : Neighborhood change

```
Function NeighborhoodChange (x, x', k)
1: if f (x') < f(x) then
2:   x ← x' // Make a move
3:   k ← 1 // Initial neighborhood
4: else
5:   k ← k+1 // Next neighborhood
```

- When Variable neighborhood search does not render good solution, there are several steps which could be helped in process, such as comparing first and best improvement strategies in local search, reducing neighborhood, intensifying shaking, adopting variable neighborhood descent, and experimenting with parameter settings.

☞ **Variable Neighborhood Descent**

- The Variable Neighborhood Descent (VND) method is obtained if a change of neighborhoods is performed in a deterministic way. In the descriptions of its algorithms, we assume that an initial solution x is given. Most local search heuristics in their descent phase use very few neighborhoods.
- The final solution should be a local minimum with respect to all neighborhoods; hence the chances to reach a global one are larger when using variable neighborhood descent than with a single neighborhood structure.

Syllabus Topic : Beam Search

1.11 Beam Search

- In all the variations of hill climbing till now, we have considered only one node getting selected at a time for further search process. These algorithms are memory efficient in that sense. But when an unfruitful branch gets explored even for some amount of time it is a complete waste of time and memory. Also the solution produce may not be the optimal one.

- The local beam search algorithm keeps track of k best states by performing parallel k searches. At each step it generates successor nodes and selects k best nodes for next level of search. Thus rather than focusing on only one branch it concentrates on k paths which seems to be promising. If any of the successors found to be the goal, search process stops.
- In parallel local beam search, the parallel threads communicate to each other, hence useful information is passed among the parallel search threads.
- In turn, the states that generate the best successors say to the others, "Come over here, the grass is greener!" The algorithm quickly terminates unfruitful branches exploration and moves its resources to where the path seems most promising. In stochastic beam search the maintained successor states are chosen with a probability based on their goodness.

Algorithm : Local Beam search

```
Step 1 :  Found = false;
Step 2 :  NODE = Root_node;
Step 3 :  If NODE is the goal node, then Found = true
          else find SUCCs of NODE, if any with its
          estimated cost and store in OPEN list;
Step 4 :  While (Found = false and not able to proceed
          further)
              {
              Sort OPEN list;
              Select top W elements from OPEN list and
              put it in W_OPEN list and empty OPEN list;
          While (W_OPEN ≠ϕ and Found = false)
          {
              Get NODE from W_OPEN;
              if NODE = Goal state then Found = true
              else
              {
                  Find SUCCs of NODE, if any with its
                  estimated cost store in OPEN list;
              }
          } // end inner while
      } // end outer while
ep 5 :  If Found = true then return Yes otherwise return
        No and Stop
```

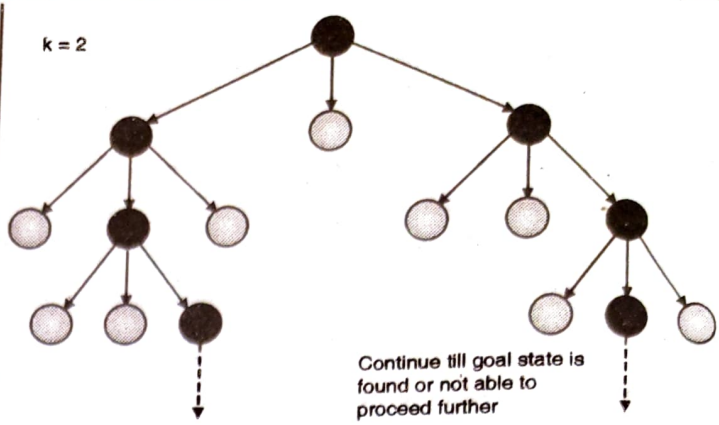


Fig. 1.11.1 : Process of local Beam Search

- As shown in Fig. 1.11.1, here k = 2, hence two better successors are selected for expansion at first level of search and at each next level, two better successors will be selected by both searches. They do exchange their information with each other throughout the search process. The search will continue till goal state is found or no further search is possible.
- It may seem to be that local beam search is same as running k searches in parallel. But it is not so. In case of parallel searches, all search run independent of each other. While in case of local beam search, the parallel running threads continuously coordinate with one another to decide the fruitful region of the search tree.
- Local beam search can suffer from a lack of diversity among the k states by quickly concentrating to small region of the state space.
- While it is possible to get excellent fits to training data, the application of back propagation is fraught with difficulties and pitfalls for the prediction of the performance on independent test data. Unlike most other learning systems that have been previously discussed, there are far more choices to be made in applying the gradient descent method.
- The key variations of these choices are : The learning rate and local minima - the selection of a learning rate is of critical importance in finding the true global minimum of the error distance.
- Back propagation training with too small a learning rate will make agonizingly slow progress. Too large a learning rate will proceed much faster, but may simply produce oscillations between relatively poor solutions.

- Both of these conditions are generally detectable through experimentation and sampling of results after a fixed number of training epochs.
- Typical values for the learning rate parameter are numbers between 0 and 1:0.05 < h < 0.75.
- One would like to use the largest learning rate that still converges to the minimum solution momentum - empirical evidence shows that the use of a term called momentum in the back propagation algorithm can be helpful in speeding the convergence and avoiding local minima.
- The idea about using a momentum is to stabilize the weight change by making non-radical revisions using a combination of the gradient decreasing term with a fraction of the previous weight change :

$$\Delta w(t) = -\partial Ee/\partial w(t) + \alpha \Delta w(t-1)$$

where a is taken 0£ a £ 0.9, and t is the index of the current weight change.

- This gives the system a certain amount of inertia since the weight vector will tend to continue moving in the same direction unless opposed by the gradient term.
- The momentum has the following effects :
  - It smooths the weight changes and suppresses cross-stitching, that is cancels side-to-side oscillations accross the error valey ;
  - When all weight changes are all in the same direction the momentum amplifies the learning rate causing a faster convergence;
  - Enables to escape from small local minima on the error surface.
- The hope is that the momentum will allow a larger learning rate and that this will speed convergence and avoid local minima. On the other hand, a learning rate of 1 with no momentum will be much faster when no problem with local minima or non-convergence is encountered.

Sequential or random presentation

- The epoch is the fundamental unit for training, and the length of training often is measured in terms of epochs. During a training epoch with revision after a particular example, the examples can be presented in the same sequential order, or the examples could be presented in a different random order for each epoch. The random representation usually yields better results.

Advantages and disadvantages of randomness

- **Advantages :** It gives the algorithm some stochastic search properties. The weight state tends to jitter around its equilibrium, and may visit occasionally nearby points. Thus it may escape trapping in suboptimal weight configurations. The on-line learning may have a better chance of finding a global minimum than the true gradient descent technique.
- **Disadvantages :** The weight vector never settles to a stable configuration. Having found a good minimum it may then continue to wander around it.

Random initial state - unlike many other learning systems, the neural network begin in a random state. The network weights are initialized to some choice of random numbers with a range typically between – 0.5 and 0.5 (The inputs are usually normalized to numbers between 0 and 1). Even with identical learning conditions, the random initial weights can lead to results that differ from one training session to another.

The training sessions may be repeated till getting the best results.

Syllabus Topic : Tabu Search

1.12 Tabu Search

- Tabu means socially or culturally proscribed : forbidden to be used, mentioned, or approached because of social or cultural rather than legal prohibitions.
- Tabu search is based on introducing flexible memory structures in conjunction with strategic restrictions and aspiration levels as a means for exploiting search spaces. Meta-heuristic that guides a local heuristic search procedure to explore the solution space beyond local optimum by use of a Tabu List.
- Tabu search is used to solve combinatorial optimization problems. It is a dynamic neighbourhood search method. Use of a flexible memory to restrict the next solution choice to some subset of neighbourhood of current solution.

- There are 3 main strategies of Tabu search.

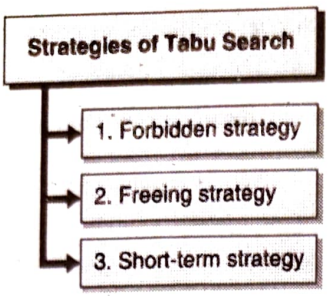


Fig. 1.12.1 : Strategies of Tabu search

➜ **1. Forbidding strategy**

Control what enters the tabu list.

➜ **2. Freeing strategy**

Control what exits the tabu list and when.

➜ **3. Short-term strategy**

- Manage interplay between the forbidding strategy and freeing strategy to select trial solutions.

☞ **Parameters used in Tabu search**

- Local search procedure
- Neighborhood structure
- Aspiration conditions
- Form of tabu moves
- Addition of a tabu move
- Maximum size of tabu list
- Stopping rule

☞ **Concept**

- A chief way to exploit memory in tabu search is to classify a subset of the moves in a neighborhood as forbidden (or tabu).
- A neighborhood is constructed to identify adjacent solutions that can be reached from current solution.
- The classification depends on the history of the search, and particularly on the recency or frequency that certain move or solution components, called attributes, have participated in generating past solutions. A tabu list records forbidden moves, which are referred to as tabu moves.
- Tabu restrictions are subject to an important exception. When a tabu move has a sufficiently attractive evaluation where it would result in a solution better than any visited so far, then its tabu classification may be overridden.
- A condition that allows such an override to occur is called an aspiration criterion.

1.12.1 Basic Tabu Search Algorithm

Step 1 : Choose an initial solution i in S. Set i* = i and k = 0.

Step 2 : Set k = k + 1 and generate a subset V of solution in N(i,k) such that either one of the Tabu conditions is violated or at least one of the aspiration conditions holds.

Step 3 : Choose a best j in V and set i = j.

Step 4 : If f(i) < f(i*) then set i* = i.

Step 5 : Update Tabu and aspiration conditions.

Step 6 : If a stopping condition is met then stop.

Else go to Step 2.

- Following are the Stopping Conditions for Tabu search.

Some immediate stopping conditions could be the following :

1. N(i, K+1) = 0. (no feasible solution in the neighbourhood of solution i)
2. K is larger than the maximum number of iterations allowed.
3. The number of iterations since the last improvement of i* is larger than a specified number.
4. Evidence can be given than an optimum solution has been obtained.

- For example, stopping criterion can use a fixed number of iterations, a fixed amount of CPU time, or a fixed number of consecutive iterations without an improvement in the best objective function value. Also stop at any iteration where there are no feasible moves into the local neighbourhood of the current trial solution.

☞ **Example**

- Consider following example of minimum spanning tree problem with constraints.
- **Objective :** Connects all nodes with minimum costs.

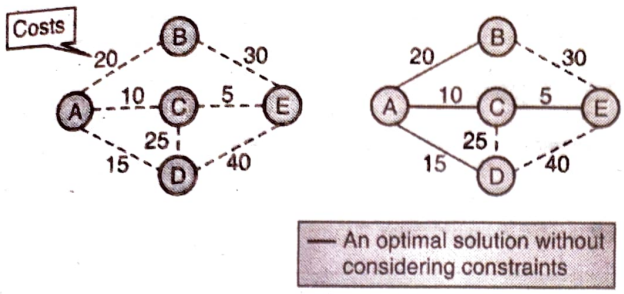


Fig. 1.12.2

- **Constraints 1 :** Link AD can be included only if link DE also is included. (Penalty : 100)
- **Constraints 2 :** At most one of the three links – AD, CD and AB – can be included. (Penalty of 100 if selected two of the three, 200 if all three are selected.)

Iteration 1

Cost = 50 + 200 (Constraint penalties)

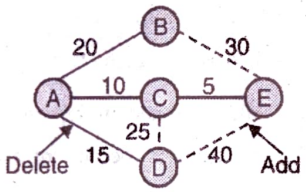


Fig. 1.12.3

Add	Delete	Cost
BE	CE	75 + 200 = 275
BE	AC	70 + 200 = 270
BE	AB	60 + 100 = 160
CD	AD	60 + 100 = 160
CD	AC	65 + 300 = 365
DE	CE	85 + 100 = 185
DE	AC	80 + 100 = 180
DE	AD	75 + 0 = 75

New cost = 75 (Iteration 2)

(local optimum)

- **Constraints 1 :** Link AD can be included only if link DE also is included. (Penalty : 100)
- **Constraints 2 :** At most one of the three links – AD, CD and AB – can be included. (Penalty of 100 if selected two of the three, 200 if all three are selected.)

Tabu list : DE

Iteration 2

Cost = 75

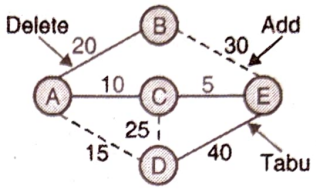


Fig. 1.12.4

Add	Delete	Cost
AD	DE*	Tabu move
AD	CE	85 + 100 = 185
AD	AC	80 + 100 = 180
BE	CE	100 + 0 = 100
BE	AC	95 + 0 = 95
BE	AB	85 + 0 = 85
CD	DE*	Tabu move
CD	CE	95 + 100 = 195

A* tabu move will be considered only if it would result in a better solution than the best trial solution found previously (Aspiration condition) Iteration3 new cost = 85 Escape local optimum.

- **Constraints 1 :** Link AD can be included only if link DE also is included. (Penalty : 100)
- **Constraints 2 :** At most one of the three links – AD, CD and AB – can be included. (Penalty of 100 if selected two of the three, 200 if all three are selected.)

Tabu list : DE & BE

Iteration 3

Cost = 85

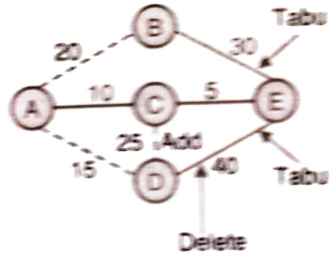


Fig. 1.12.5

Add	Delete	Cost
AB	BE*	Tabu move
AB	CE	100 + 0 = 100
AB	AC	95 + 0 = 95
AD	DE*	60 + 100 = 160
AD	AC	95 + 0 = 95
AD	AB	90 + 0 = 90
CD	DE*	Tabu move
CD	CE	105 + 0 = 105

A* tabu move will be considered only if it would result in a better solution than the best trial solution found previously (Aspiration condition) Iteration3 new cost = 70 Override tabu status

- **Constraints 1 :** Link AD can be included only if link DE also is included. (Penalty : 100)
- **Constraints 2 :** At most one of the three links – AD, CD and AB – can be included. (Penalty of 100 if selected two of the three, 200 if all three are selected.)

☞ **Optimal solution**

Cost = 70

Additional iterations only find inferior solutions

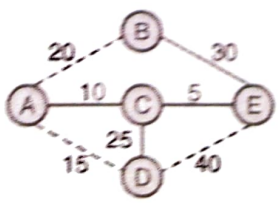


Fig. 1.12.6

Syllabus Topic : Optimal Search

1.13 Optimal Search

Syllabus Topic : A* Algorithm

1.13.1 A* Algorithm

Q. 1.13.1 Explain A* Algorithm in detail with example. (Refer section 1.13.1) (8 Marks)

☞ **Concept**

- A* pronounced as "Aystar" (Hart, 1972) search method is a combination of branch and bound and best first search, combined with the dynamic programming principle.
- It's a variation of Best First search where the evaluation of a state or a node not only depends on the heuristic value of the node but also considers its distance from the start state. It's the most widely known form of best-first search. A* algorithm is also called as OR graph / tree search algorithm.
- In A* search, the value of a node n, represented as f(n) is a combination of g(n), which is the cost of cheapest path to reach to the node from the root node, and h(n), which is the cost of cheapest path to reach from the node to the goal node. Hence

$$f(n) = g(n) + h(n).$$

- As the heuristic can provide only the estimated cost from the node to the goal we can represent h(n) as h*(n); similarly g*(n) can represent approximation of g(n) which is the distance from the root node observed by A* and the algorithm A* will have,

$$f^*(n) = g^*(n) + h^*(n)$$

- As we observe the difference between the A* and Best first search is that; in Best first search only the heuristic estimation of h(n) is considered while A* counts for both, the distance travelled till a particular node and the estimation of distance need to travel more to reach to the goal node, it always finds the cheapest solution.
- A reasonable thing to try first is the node with the lowest value of g*(n) + h*(n). It turns out that this strategy is more than just reasonable, provided that the heuristic function

h*(n) satisfies certain conditions which are discussed further in the chapter. A* search is both complete and optimal.

☞ **Implementation**

A* does also use both OPEN and CLOSED list.

1.13.1(A) Algorithm (A*)

1. Initialization OPEN list with initial node; CLOSED= ϕ; g = 0, f = h, Found = false;
2. While (OPEN ≠ ϕ and Found = false)

{1

i. Remove the node with the lowest value of f from OPEN to CLOSED and call it as a Best_Node.

ii If Best_Node = Goal state then Found = true

iii. else

{2

a. Generate the Succof Best_Node

b. For each Succ do

{3

i. Compute g(Succ) = g(Best_Node) + cost of getting from Best_Node to Succ.

ii. If Succ∈ OPEN then /* already being generated but not processed */

{4

a. Call the matched node as OLD and add it in the list of Best_Node successors.

b. Ignore the Succ node and hange the parent of OLD, if required.

– If g(Succ) < g(OLD) then make parent of LD to be Best_Node and change the values of g and f for OLD

– If g(Succ) >= g(OLD) then ignore

}4

a. If Succ∈ CLOSED then /* already processed */

{5

i. Call the matched node as OLD and add it in the list of Best_Node successors.

ii. Ignore the Succ node and change the parent of OLD, if required

- If g(Succ) < g(OLD) then make parent of OLD to be Best_Node and change the values of g and f for OLD.
- Propogate the change to OLD's children using depth first search
- If g(Succ) >= g(OLD) then do nothing

}5

a. If Succ∉ OPEN or CLOSED

{6

i. Add it to the list of Best_Node's successors

ii. Compute f(Succ) = g(Suce) + h(Succ)

iii. Put Succ on OPEN list with its f value

}6

}3 /* for loop*/

}2 /* else if */

}1 /* End while */.

3. If Found = true then report the best path else report failure
4. Stop

1.13.1(B) Behaviour of A* Algorithm

Q. 1.13.2 Write short note on behavior of A* in case of underestimating and overestimating Heuristic. (Refer section 1.13.1(B)) **(8 Marks)**

As stated already the success of A* totally depends upon the design of heuristic function and how well it is able to evaluate each node by estimating its distance from the goal node. Let us understand the effect of heuristic function on the execution of the algorithm and how the optimality gets affected by it.

A. Underestimation

- If we can guarantee that heuristic function 'h' never over estimates actual value from current to goal that is, the value generated by h is the always lesser than the actual cost or actual number of hopes required to reach to the goal state. In this case, A* algorithm is guaranteed to find an optimal path to a goal, if one exists.

Example

f = g + h, Here h is underestimated.

- If we consider cost of all arcs to be 1. A is expanded to B, C and D. 'f' values for each node is computed. B is chosen to be expanded to E. We notice that f(E) = f(C) = 5. Suppose we resolve in favor of E, the path currently we are expanding. E is expanded to F. Expansion of a node F is stopped as f(F) = 6 so we will now expand node C.
- Hence by underestimating h(B), we have wasted some effort but eventually discovered that B was farther away than we thought. Then we go back and try another path, and will find optimal path.

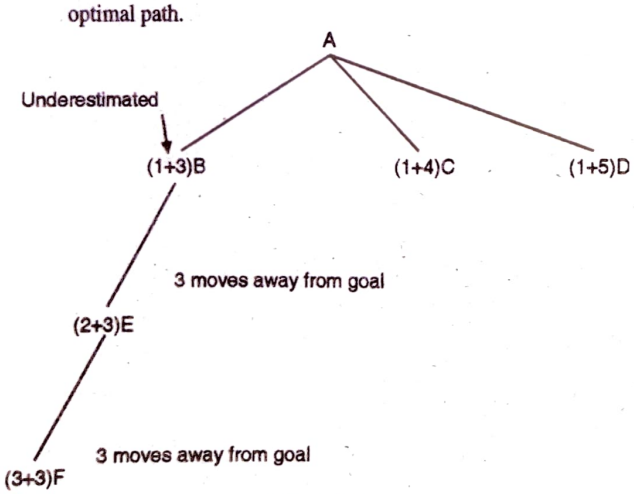


Fig. 1.13.1

B. Overestimation

- Here h is overestimated that is, the value generated for each node is greater than the actual number of steps required to reach to the goal node.

Example

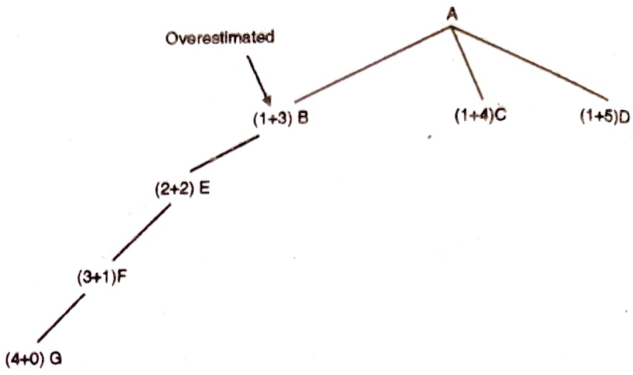


Fig. 1.13.2

- As shown in the example, A is expanded to B, C and D.
- Now B is expanded to E, E to F and F to G for a solution path of length 4. Consider a scenario when there a direct path from D to G with a solution giving a path of length 2. This path will never be found because of overestimating h(D).
- Thus, some other worse solution might be found without ever expanding D. So by overestimating h, one cannot guarantee to find the cheaper path solution.

1.13.1(C) Admissibility of A*

Q. 1.13.3 Discuss admissibility of A* in case of optimality. (Refer section 1.13.1(C)) **(8 Marks)**

Q. 1.13.4 What do you mean by admissible heuristic function? Explain with example. (Refer section 1.13.1(C)) **(8 Marks)**

Q. 1.13.5 Write short note on admissibility of A*. (Refer section 1.13.1(C)) **(8 Marks)**

- A search algorithm is admissible, if for any graph, it always terminates in an optimal path from initial state to goal state, if path exists. A heuristic is admissible if it never over estimates the actual cost from current state to goal state.
- Alternatively, we can say that A* always terminates with the optimal path in case h(n) is an admissible heuristic function.
- A heuristic h(n) is admissible if for every node n, if $h(n) \leq h^*(n)$, where $h^*(n)$ is the true cost to reach the goal state from n. An admissible heuristic never overestimates the cost to reach the goal.

- Admissible heuristics are by nature optimistic because they think the cost of solving the problem is less than it actually is.
- An obvious example of an admissible heuristic is the straight line distance. Straight line distance is admissible because the shortest path between any two points is a straight line, so the straight line cannot over estimate the actual road distance.
  - **Theorem :** If h(n) is admissible, tree search using A* is optimal.
  - **Proof :** Optimality of A*with admissible heuristic.
- Suppose some suboptimal goal G_2 has been generated and is in the fringe. Let n be an unexpanded node in the fringe such that n is on a shortest path to an optimal goal G.

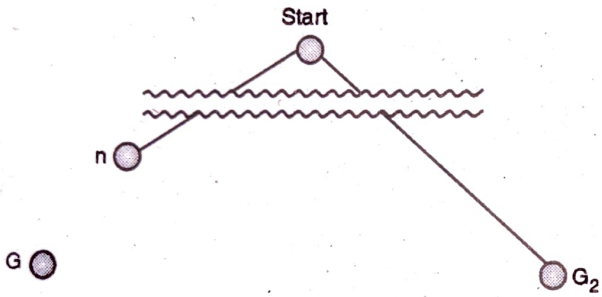


Fig. 1.13.3 : Optimality of A*

$$f(G_2) = g(G_2) \quad \text{since } h(G_2) = 0$$

$$g(G_2) > g(G) \quad \text{since } G_2 \text{ is suboptimal}$$

$$f(G) = g(G) \quad \text{since } h(G) = 0$$

$$f(G_2) > f(G) \quad \text{from above}$$

$$h(n) \leq h^*(n) \quad \text{since h is admissible}$$

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

$$f(n) \leq f(G)$$

Hence $f(G_2) > f(n)$, and A* will never select G_2 for expansion.

1.13.1(D) Monotonicity

Q. 1.13.6 Prove that A* is admissible if it uses Monotone heuristic. (Refer section 1.13.1(D)) **(8 Marks)**

- A heuristic function h is monotone or consistent if,

  ∀states X_i and X_j such that X_j is successor of Xi,

  $$h(X_i) - h(X_j) \leq \text{cost}(X_i, X_j)$$

  where, cost (X_i, X_j) actual cost of going from X_i to X_j and h (goal) = 0
- In this case, heuristic is locally admissible i.e., consistently finds the minimal path to each state they encounter in the search. The monotone property in other words is that search space which is everywhere locally consistent with heuristic function employed i.e., reaching each state along the shortest path from its ancestors.
- With monotonic heuristic, if a state is rediscovered, it is not necessary to check whether the new path is shorter.
- Each monotonic heuristic is admissible.
- A cost function f(n) is monotone if $f(n) \leq f(\text{succ}(n))$, ∀n.
- For any admissible cost function f, we can construct a monotone admissible function.
- Alternatively, the monotone property that search space which is everywhere locally consistent with heuristic function employed i.e., reaching each state along the shortest path from its ancestors.
- With monotonic heuristic, if a state is rediscovered, it is not necessary to check whether the new path is shorter.

1.13.1(E) Properties of A*

1. **Completeness :** It is complete, as it will always find solution if one exist.
2. **Optimality :** Yes, it is Optimal.
3. **Time Complexity :** $O(b^m)$, as the number of nodes grows exponentially with solution cost.
4. **Space Complexity :** $O(b^m)$, as it keeps all nodes in memory.

1.13.1(F) Example : 8 Puzzle Problem using A* Algorithm

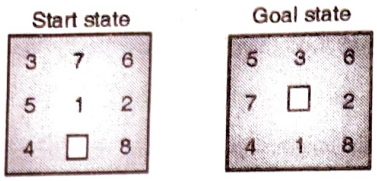
- The choice of evaluation function critically determines search results. Consider Evaluation function

  $f(X) = g(X) + h(X)$

  $h(X)$ = the number of tiles not in their goal position in a given state X

g(X) = depth of node X in the search tree

For Initial node f(initial_node) = 4



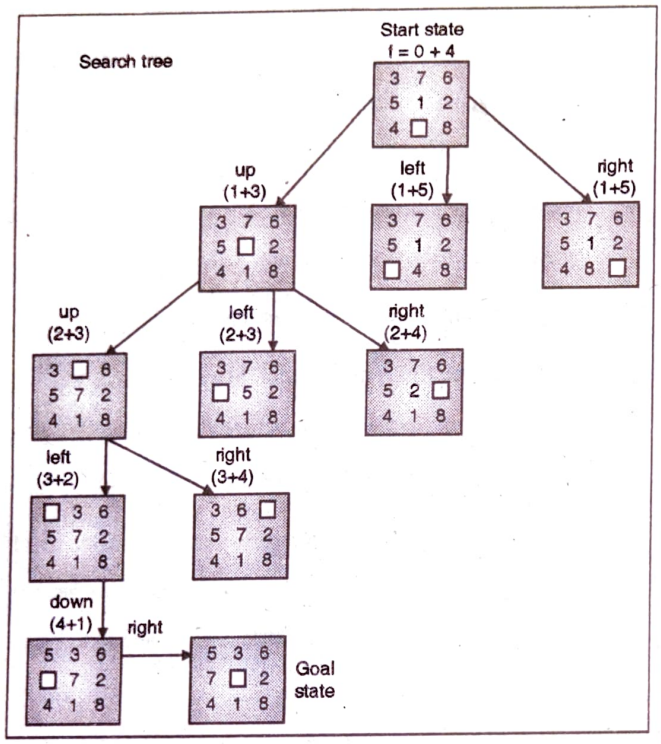


Fig. 1.13.4 : Solution of 8-puzzle using A*

Ex. 1.13.1 : Consider the graph given in Fig. P. 1.13.1. Assume that the initial state is S and the goal state is 7. Find a path from the initial state to the goal state using A* Search. Also report the solution cost. The straight line distance heuristic estimates for the nodes are as follows : h(1) = 14, h(2) = 10. h(3) = 8, h(4) = 12, h(5) = 10, h(6) = 10, h(S) = 15.

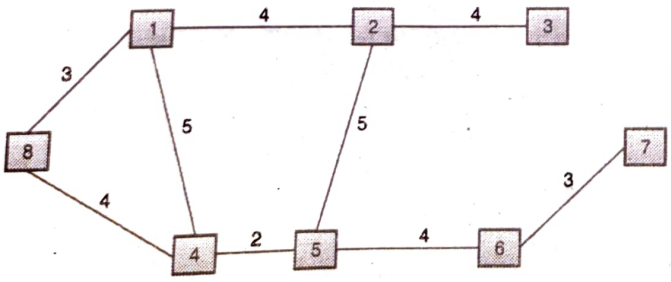
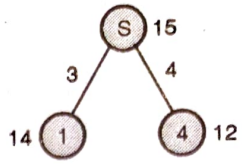


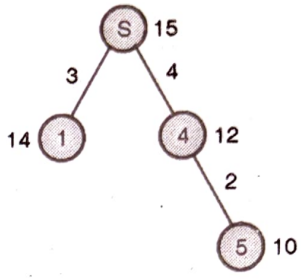
Fig. P. 1.13.1

Soln. :



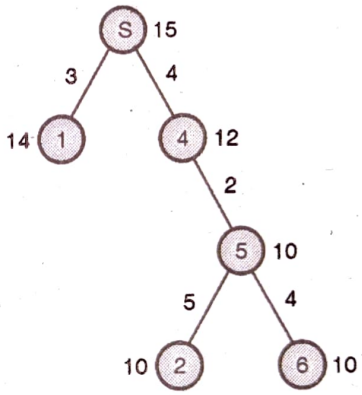
Open : 4(12 + 4), 1(14 + 3)

Closed : S(15)



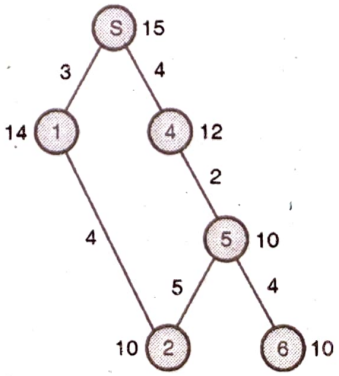
Open : 5(10 + 6), 1(14 + 3)

Closed : S(15), 4 (12 + 4)



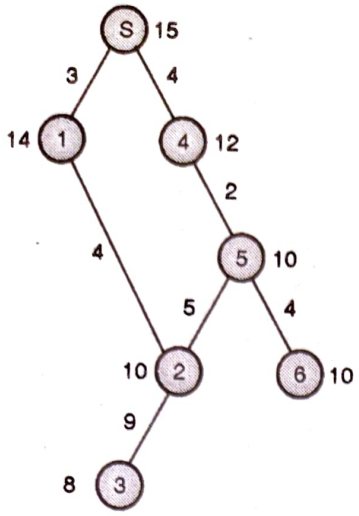
Open : 1(14 + 3) 6(10 + 10) 2(10 + 11)

Closed : S(15) 4(12 + 4) 5(10 + 6)



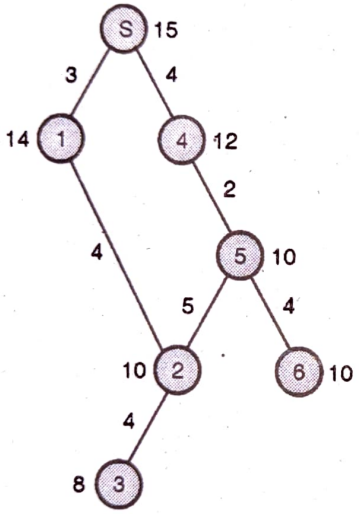
Open : 2(10 + 7) 6(10+10)

Closed : S(5) 4(12 + 4) 5(10 + 6) 1(14 +3)



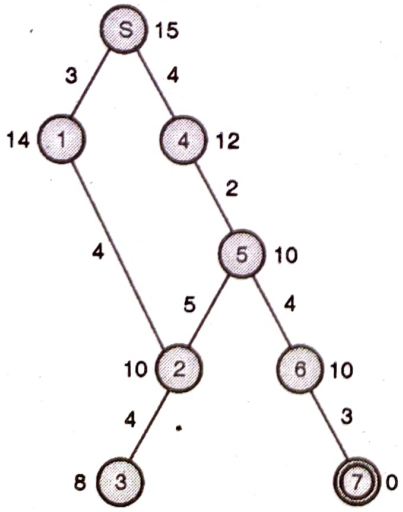
Open : 3(8 +11), 6(10 + 10)

Closed : S(15),4(12+4),5(10+6), 1(14+ 3),2(10+7)



Open : 6(10 + 10)

Closed : s(15), 4(12 + 4), 5(10 + 6), 1(14 + 3) 2(10 + 7) 3(8 + 11)



Open : 7(0 + 13)

Closed : S(15), 4(12 + 4), 5(10 + 6), 1(14 + 3),

2(10 + 7), 5(8 + 4), 6(10 + 10)

Syllabus Topic : Iterative Deepening A*

1.13.2 Iterative Deepening A*

☞ **Concept**

- As we have learned that A* requires to keep all the nodes in memory, it becomes very difficult to manage in case of large and complex state space. In spite of being complete and optimal, A* cannot be used for large state problems because of this limitation.
- Breadth First search faces same problem of huge memory requirement. But Iterative Deepening (IDDFS) solved it by limiting the search depth.
- Similarly, Iterative Deepening A* (IDA*) eliminates the memory constraints of A* by keeping optimality of the solution intact.
- In IDA* each iteration is a depth-first search with a threshold assigned to it. It keeps track of the cost, f(n) = g(n) + h(n), of each node generated.
- As soon as the cost of newly generated node exceeds a threshold for that iteration, the node will not be explored further and the search backtracks.
- The threshold is initialized to the heuristic estimate of the initial state. It is increased in each successive iteration to the total cost of the lowest cost node that was pruned during the previous iteration. The algorithm terminates when a goal state is reached whose total cost does not exceed the current threshold.

1.13.2(A) Algorithm

Node current node	
g	the cost to reach current node
f	estimated cost of the cheapest path (root..node..goal)
h(node)	estimated cost of the cheapest path (node..goal)
cost(node, succ)	step cost function
is_goal(node)	goal test

```
successors(node)  node expanding function

procedureida_star(root)
bound := h(root)
loop
t := search(root, 0, bound)
if t = FOUND then return bound
if t = ∞then return NOT_FOUND
bound := t
end loop
end procedure

functionsearch(node, g, bound)
f := g + h(node)
if f > bound then return f
ifis_goal(node) then return FOUND
min := ∞
forsuccinsuccessors(node) do
t := search(succ, g + cost(node, succ), bound)
if t = FOUND then return FOUND
if t < min then min := t
end for
return min
end function
```

1.13.2(B) Performance Evaluation

1. **Optimality :** IDA* finds optimal solution, if the heuristic function is admissible.
2. **Completeness :** IDA* is complete. It always finds solution if it exists within the threshold limit.
3. **Time Complexity :** IDA* expands the same number of nodes, as A*. So it has exponential time complexity.
4. **Space Complexity :** IDA* performs a series of depth-first searches. Hence, its memory requirement is linear with respect to the maximum search depth.

From the above discussion it is clear that IDA* is optimal in time and space as compared to all other heuristic search algorithms that find optimal solutions on a tree. As in case of A*, in IDA*, we need not manage OPEN and CLOSED list hence, it often runs faster than A* and the implementation is much simpler as compared to A*.

Syllabus Topic : Recursive Best First Search

1.14 Recursive Best First Search

Concept

- Recursive Best-First Search (RBFS) is an alternative algorithm to IDA*. Recursive best-first search is a best-first search that runs in space that is linear with respect to the maximum search depth, regardless of the cost function used. Even with an admissible cost function, Recursive Best-First Search generates fewer nodes than IDA*, and is generally superior to IDA*, except for a small increase in the cost per node generation.
- It works by maintaining on the recursion stack the complete path to the current node being expanded as well as all immediate siblings of nodes on that path, along with the cost of the best node in the sub-tree explored below each sibling. Whenever the cost of the current node exceeds that of some other node in the previously expanded portion of the tree, the algorithm backs up to their deepest common ancestor, and continues the search down the new path. In effect, the algorithm maintains a separate threshold for each sub-tree diverging from the current search path.

1.14.1 Algorithm

```
RBFS (node: N, value: F(N), bound: B)
IF f(N)>B, RETURN f(N)
IF N is a goal, EXIT algorithm
IF N has no children, RETURN infinity
FOR each child Ni of N,
  IF f(N)<F(N), F[i] := MAX(F(N),f(Ni))
  ELSE F[i] := f(Ni)
sort Ni and F[i] in increasing order of F[i]
IF only one child, F[2] := infinity
```

```
WHILE (F[1] <= B and F[1] < infinity)
F[1] := RBFS(N1, F[1], MIN(B, F[2]))
insert Ni and F[1] in sorted order
RETURN F[1]
```

Example

- Let us understand the workings of this algorithm with a binary tree where f(N) = the depth of node N. What follows is a brief, structured description of how the algorithm works in this case. It is best to try to work the algorithm on your own on paper and use this as a reference to check your work.
- Nodes in the binary tree are named A, B, C, ... from left-to-right, top-to-bottom. Assume the tree is infinite and has no goal. Note that the stored value F(N) is different from f(N). When sorted children are listed (e.g. "B(1) C(1)"), the number inside the parentheses is the stored value. Recursive calls are indented; the first line is the initial call on the root.

```
RBFS(A, 0, 4)
f(A)=F(A), so F(B)=f(B)=1, F(C)=f(C)=1
Sorted children: B(1) C(1)
F(B)< 4, so
  RBFS(B, 1, 1)
  f(B)=F(B), so F(D)=f(D)=2, F(E)=f(E)=2
  Sorted children: D(2) E(2)
  F(D)>1, so return 2
F(B)=2
Sorted children: C(1) B(2)
F(C)< 4, so
  RBFS(C, 1, 2)
  f(C)=F(C), so F(F)=f(F)=2, F(G)=f(G)=2
  Sorted children: F(2) G(2)
  F(F)<=2, so
    RBFS(F, 2, 2)
    ... search F's children to 2 returning min cost beyond ...
    return 3
  F(F)=3
  Sorted children: G(2) F(3)
  F(G)<=2, so
    RBFS(G, 2, 2)
    ... search G's children to 2 returning min cost beyond ...
    return 3
  F(G)=3
  Sorted children: F(3) G(3)
  F(F)>2, so return 3
F(C)=3
Sorted children: B(2) C(3)
F(B)< 4, so
  RBFS(B, 2, 3)
  f(B)<F(B), so F(D)=MAX(F(B),f(D))=2,
  F(E)=MAX(F(B),f(E))=2
  Sorted children: D(2) E(2)
  F(D)<=3, so
RBFS(D, 2, 2)
... search D's children to 2 returning min cost beyond ...
return 3
  F(D)=3
```

Sorted children: E(2) D(3)

F(E) <=3, so

RBFS(E, 2, 3)

... search E's children to 3 returning min cost beyond ...

return 4

F(E)=4

Sorted children: D(3) E(4)

F(D) <=3, so

RBFS(D, 3, 3)

... search D's children to 3 returning min cost beyond ...

return 4

F(D)=4

Sorted children: E(4) D(4)

F(E)>3, so return 4

F(B)=4

Sorted children: C(3) B(4)

F(C)< 4, so

RBFS(C, 3, 4)

... search C's children to 4 returning min cost beyond ...

return 5

F(C)=5

Sorted children: B(4) C(5)

F(B)< 4, so

RBFS(B, 4, 5)

... search B's children to 5 returning min cost beyond ...

return 6

F(B)=6

Sorted children: C(5) B(6)

... and so on ...

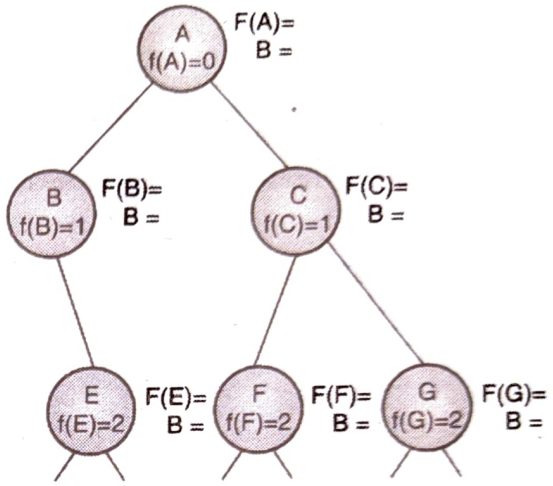


Fig. 1.14.1

Syllabus Topic : Pruning the CLOSED and OPEN Lists

1.15 Pruning the CLOSED and OPEN Lists

1.15.1 Pruning the CLOSED Lists

☞ **Divide and Conquer Frontier search**

- The objective of having CLOSED list is to maintain all visited nodes so that they are not explored again and again. i.e. stopping the search from leaking back.
- In the algorithm designed by Korf and Zang, the parent information is stored with the child node which is placed in the OPEN list. That forma tabu list, i.e. the children which are already in this list will not be expanded again.
- This enables the search to proceed in the forward direction. If we get a new node having the same parent, the parent list of the node is updated. And we have every edge traversed only once while searching. So that the child parent relationship is maintained. And the child can never become parent of the same node.

- By putting this information in this list we are avoiding the nodes from generation again and again and hence called as pruning of closed list.
- Another Zhou and Hansen, devised another algorithm, in which the CLOSED list is categorized into two parts. One is called as Kernel and the other is called as Boundary.
- Kernel has no children on OPEN list and Boundary is the one having at least one child in OPEN list. They are formed basically to distinguish between closed and open nodes.
- The nodes on the frontier are open nodes and boundary nodes are the one connected to these frontier open nodes. And all the CLOSED nodes are shown in the Kernel.
- When the children of open are generated we only look ahead to open list and no node in CLOSED list will be expanded again. i.e. if any of the newly generated child already exists in Boundary or Kernel, it will not be expanded again.

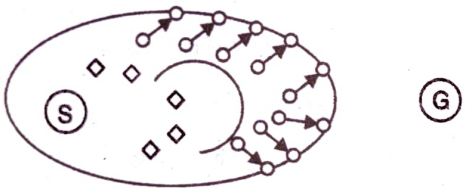


Fig. 1.15.1

- The second function of CLOSED list is to reconstruct the path from start to goal node. This can be generated from Boundary nodes. All the nodes in kernel need not be seen to generate the path to destination.

1.15.2 Pruning OPEN List

- For the problems in which the problem space grows exponentially, the OPEN list is the one that grows up very fast and accounts for most of the memory.
- Lets understand the techniques to prun this OPEN list in this section.

☞ Breadth First Heuristic Search

- In case of breadth first search the main limitation was the higher memory requirement.
- If we can make the search as informed search and put an upper bound U on the cost, then we can limit the search for the nodes having cost higher than U.
- This upper bound can be obtained by applying a heuristic function or by using local beam search. The algorithm works same as breadth first search but explores only those nodes having cost less that U. It can be called as Breadth First Heuristic Search.
- Pruning the OPEN list nodes will always give admissible solution because, the pruning will happen till the half way mark to the solution and after that as the search moves towards the destination, the cost will keep on reducing and all the nodes will be explored to find the optimal solution.
- It is observed that the number of nodes required to be expanded in moderate problems are even less than that of A* search and breadth first heuristic can produce optimal solution most of the times.

❑❑❑