2 CHAPTER

AI APPROACHES

2.0 INTRODUCTION

A problem solving agent firstly formulates a goal and a problem to solve. Then the agent calls a search procedure to solve it. It then uses the solution to guide its actions. It will do whatever the solution recommends as the next thing to do and then remove that step from the sequence. Once the solution has been executed, the agent will formulate a new goal. **Please note that in intelligent agents case, a KB corresponds to the environment, operators correspond to sensors and the search techniques are the actuators.**

The KB describes the current task domain and the goal (or state). Operators manipulate this KB. And the control strategy decides what operators to apply and where to apply them. **Also note that the aim of any search technique is the application of an appropriate sequence of operators to an initial state to achieve the goal.** A problem solving agent (system) uses either forward or backward reasoning. Each of its operator works to produce a new state in the KB.

Forward reasoning or bottom-up or data-driven reasoning means applying operators to those structure in the KB that best describes the task domain in order to produce a modified state. On the other hand, **backward reasoning** will break down the goal (problem) statement into subgoals (problems) which are easier to solve and these sub-solutions are sufficient to solve the original problem as a whole.

2.1 PROBLEM SOLVING

2.1.1 Problem Specification

Problem domain formulation is a very important task. This problem definition must be unambiguous and clear. So, we define an abstract problem in terms of real workable states which are clearly understood.

Please understand that these states are operated upon by a set of operators. The overall control strategy decides which operator is to be used. To formulate any problem, **four components** are defined in a very formal way. These four components are —

(1) The initial state (or **starting state**).

(2) **State space** i.e., a description of all possible states reachable from the initial state. **Please note that this state space forms a graph in which the nodes are states and the arcs represent actions.**

(3) ***Goal test*** that determines whether a given state is a goal state.

(4) ***Path cost*** is a function that assigns a numeric cost to each path.

Please note here that solution to a problem is a path from the initial state to a goal state. Also note that the quality of the solution is measured by the path cost function and obviously, an optimal solution has the lowest path cost among all solution.

So, these four components will do problem formulation. To achieve our objective, we use **operationalization. The process of creating a formal description of a problem using the knowledge about the given problem, so as to create a program for solving a problem is called as operationalization.** Thus, in general, any problem can be solved using the following steps—

Step-1: **Define a state space** which contains all possible configurations of the relevant objects. You can even include some impossible states.

Step-2: **Identify initial states** i.e., one or more such states from our search space that would describe possible situations from where problem solving may start.

Step-3: **Specify goal states** i.e., specify one or more states which would be acceptable as solutions to our problem.

Step-4: **Specify set of rules** i.e., give a set of rules which describe the actions (operators) available and a control strategy to decide the order of application of these rules.

At the **core heart** of many intelligent processes lies the process of **search.**

Search can be characterized as finding a path through a graph or tree structure. This requires moving from node to node after successively expanding and generating connected nodes. Node generation is accomplished by computing the identification or the representation code of children nodes from a parent node. Once this is done, a child is said to be *generated* and the parent is said to be *explored*. **The process of generating all of the children of a parent is also known as expanding the node. A search procedure is a strategy for selecting the order in which nodes are generated and a given path selected.**

Search problems may be classified as:

1. Blind or uninformed search: In this search technique, no preference is given to the order of successor node generation and selection. The path selected is blindly or mechanically followed. No information is used to determine the preference of one child over another. For e.g. DFS, DFS, bidirectional searchs.

2. Informed or directed search: In this search technique, some information about the problem space is used to compute a preference among the children for exploration and expansion. For e.g. Heuristic search.

2.1.2 State Space Search With Examples

We define the state space as a set of all possible sates of a given problem. A state space representati allows for the formal definition of a problem which makes the movement from initial state to the goal state quite easy. In a way, we can also say that various problems like planning, learning, discoveries in science, theorem proving etc. are all essentially search problems only.

Advantages and Disadvantages of state-space representations

Advantages: This representation is very useful in AI because it provides a set of all possible states, operations and goals. If the entire state-space representation for a problem is given then it is possible to trace the path from the initial to goal state and identify the sequence of operators required for doing it.

Disadvantage(s): It is not possible to visualize all states for a given problem. Also, the resources of the computer system are limited to handle huge (combinational) state-space representation.

We are now in a position to describe some search space problems.

EXAMPLE 1: The eight tile puzzle problem formulation.

Problem formulation

The eight tile puzzle consists of a 3-by-3 (3*3) square frame board which holds eight (8) movable tiles numbered as 1 to 8. One square is empty, allowing the adjacent tiles to be shifted. The objective of the puzzle is to find a sequence of tile movements that leads from a starting configuration to a goal configuration, as shown in Fig. 2.1 below.

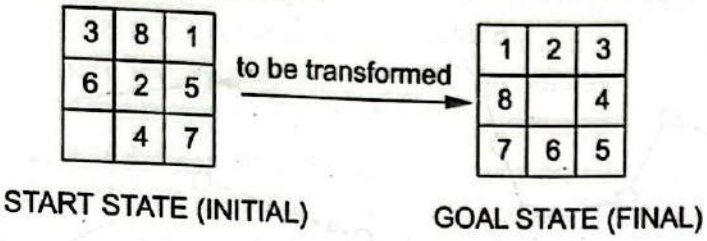


Fig. 2.1 Initial and final states of 8-puzzle problem.

The states of the 8-tile puzzle are the different permutations of the tiles within the frame.

Please note that an optimal solution is the one that maps an initial arrangement of tiles to the goal state with the smallest (minimum) number of moves. Also note that the root node can be any random starting state.

Let us do a *standard formulation* of this problem now.

States: It specifies the location of each of the 8-tiles and the blank in one of the nine squares.

***Initial state*:** Any state can be designated as the initial state.

Goal: Many goal configurations are possible one such is shown in Fig. 2.2.

***Legal moves (or states)*:** They generate legal states that result from trying the four actions—

(*a*) blank moves left.

(*b*) blank moves right.

(*c*) blank moves up.

(*d*) blank moves down.

Path cost: Each step costs 1, so the path cost is the number of steps in the path.

The tree diagram showing the search space is shown in Fig. 2.2.

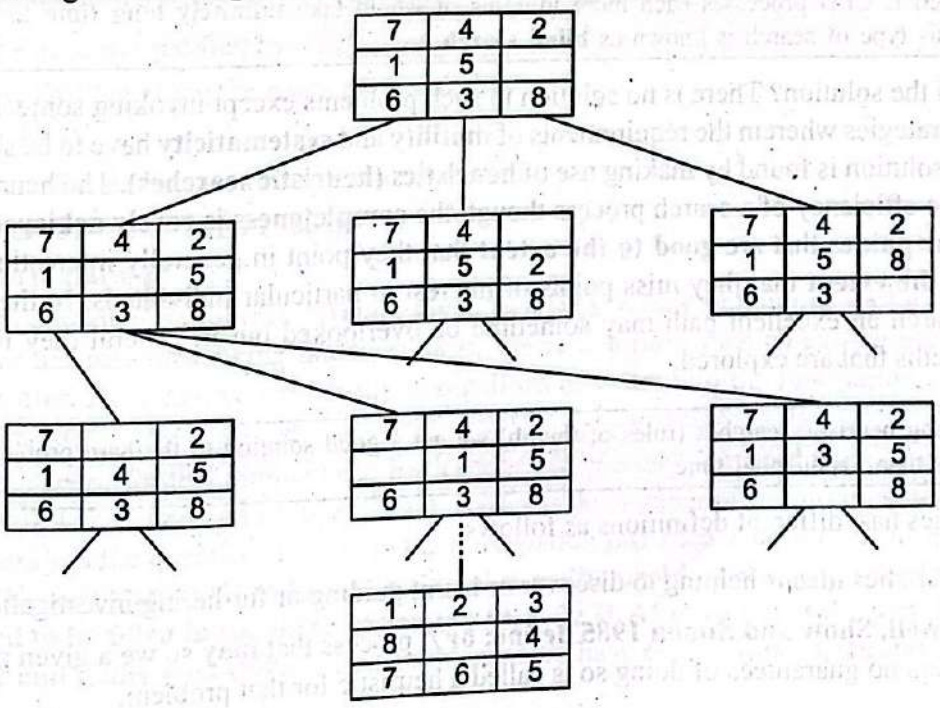


Fig. 2.2 Tree diagram (search space) of 8-puzzle problem.

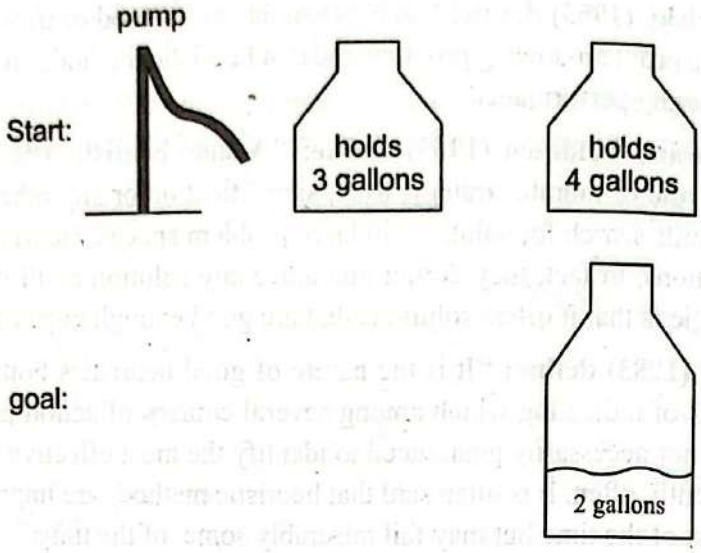


Fig. 2.4 A water jug problem.

The **operators** to be used to solve the problem is shown in table-1 below—

Table 1: Production rules (or operators) for the water jug problem

1. $(x, y) \rightarrow (4, y)$ if $x < 4$	Fill the 4-gallon jug
2. $(x, y) \rightarrow (x, 3)$ if $y < 3$	Fill the 3-gallon jug
3. $(x, y) \rightarrow (x - d, y)$ if $x > 0$	Pour some water out of the 4-gallon jug
4. $(x, y) \rightarrow (x, y - d)$ if $y > 0$	Pour some water out of 3-gallon jug
5. $(x, y) \rightarrow (0, y)$ if $x > 0$	Empty the 4-gallon jug on the ground
6. $(x, y) \rightarrow (x, 0)$ if $y > 0$	Empty the 3-gallon jug on the ground
7. $(x, y) \rightarrow (4, y - (4 - x))$ if $x + y \geq 4$ and $y > 0$	Pour water from the 3-gallon jug into the 4-gallon jug until the 4-gallon jug is full

Problem formulation

States: Amount of water in both jugs.

Actions: Empty large, empty small, pour from small to (empty) large, pour from large to (empty) small.

Goal: Specified amount of water in both jugs.

Path cost: Total number of actions applied.

Solution to Water Jug Problem

We also need a **control structure** which loops through a simple cycle in which some rule whose left side matches the current state is chosen, the appropriate change to the state is made as described in the corresponding right side and the resulting state is checked to see if it corresponds to a goal state. The loop continues as long as it does not lead to the goal. **Please note that the speed with which the problem is solved depends upon the mechanism.control structure, which is used to select the next operation. Also note that there are several sequences of operators which will solve the problem.** Two such possible solutions are shown in Fig. 2.5 (*a*) and Fig. 2.5 (*b*).

Water in four-gallon jug (x)	*Water in three-gallon jug (y)*	*Rule applied (control strategy)*
0	0	
0	3	2
3	0	9
3	3	2
4	2	7
0	2	5 or 12
2	0	9 or 11

Fig. 2.5 (a) One solution to water jug problem.

The 2nd solution can be —

Water in four-gallon jug (x)	*Water in three-gallon jug (y)*	*Rule applied (control strategy)*
0	0	
4	0	1
1	3	8
1	0	6
0	1	10
4	1	1
2	3	8

Fig. 2.5 (*b*) Another solution to water jug problem.

Thus, we find that how an informal problem state has been converted to a formal problem state. But the rules should be stated explicitly and not written because they are allowable. **For example,** rule-1 in table-1 states that — "Fill the four-gallon jug", but it should have been— "if the four gallon jug is not already filled completely". However, the rule as stated in its first-form is not wrong as there is no condition that the already filled jug cannot be filled. So, in order to increase the efficiency of problem solving program, it is imperative to encode some constraints in the left side of the rules so that the rules should lead to a solution. Thus, rules should be made more general.

Water jug problem is a simple example of solving a problem through state space search.

Even these approaches may not be exhaustive. So, we go with another method of problem-reduction techniques.

2.1.3. Problem Reduction

The process of decomposing (or breaking) up of a complex problem into a set of primitive sub-problems, finding the sub-solutions for these sub-problems and then integrating all these sub-solutions to get the solution of the given complex problem (as a whole) is known as problem reduction.

Practically, our space search is said to be good if a solution to a problem is naturally expressed in terms of either a final state or a path from an initial state to the final state. **Please note that we should be able to** define rules for transforming one state to another based on the available actions in the domain. Problem reduction is better if it is easy to decompose a problem into independent **sub-problems. Also note that we have to define rules to do this.** Here, searching will be less. We will discuss about it in detail a bit later.

2.1.4 Production Systems

These systems were proposed by Emil Post in 1943. They are also known a **inferential systems, Rule-based systems or simply productions.** For describing and performing the search operation in AI programs it is useful to structure them. The process of solving the problem can usefully be modeled as a production system. If one adopts a system with **production rules** and a '*rule-interpreter*', then the system is known as a **production system.** In these systems, the working memory of the system models human *short-term memory* while the productions are part of **long-term memory.** On each cycle of operation, productions are matched against the working memory of facts. And a production whose conditions are satisfied can add or delete facts in the working memory. **Please note that this is opposite to the situation in databases.** These production systems often have many rules and relatively fewer facts.

Rules of production systems.

Rule-1: Production systems represent both knowledge as well as action.

Rule-2: Production system provide a language in which the representation of expert knowledge is very natural.

Rule-3: Production systems provide a heuristics model for human behaviour.

Rule-4: Production systems model strong data-driven nature of intelligent action. Whenever new inputs enter the database, the behaviour of the system changes.

Rule-5: Production systems allow us to add new rules easily without disturbing the rest of the system.

Let us now see the architecture of a production system. It is shown in Fig. 2.5(c). It comprises of three main components—

1. Rule base.
2. Global database
3. Control structure.

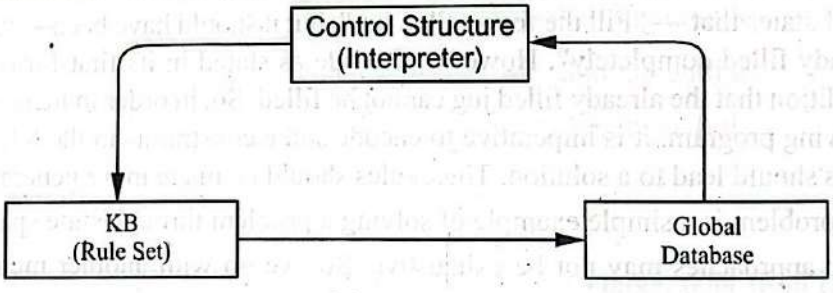


Fig. 2.5(c) Parts of a production system.

The process of transforming a problem statement into these components of the production system is called as problem representation. Selection of a good representation is an art in AI to solve practical problems and that there are many ways to represent a problem. **Please note that a good problem solution requires an efficient control strategy, good representations for problem states, moves and goal conditions.** The effort that is required to solve a problem in AI depends on the problem representation scheme. **Also note that we select those representations which need small state spaces.** If the state space is not less then we can even reduce or combine certain rules also. If these simple transformations do not work then we can even reformulate our problem but our ultimate goal is to be achieved i.e., of small state space.

Let us now see these various components of a production system.

I Rule Base

Production rules are extremely popular knowledge representation (KR) structure today. They are conditional IF-THEN branches only. These rules apply on the global database. Each rule may have a **Precondition.** If the precondition of the rule is satisfied then only the corresponding rule is applied to a state. It is only after the application of rule that the problem's state changes. The KR in production systems is decoded in a declarative (not procedural) form which comprises of a set of rules which are of this form—

Solution → action

Such problem situation-action rules are often called as production-rules or IF-THEN rules. And a system which uses this form of knowledge is called as a production system. Each of the IF condition is called as clause (s). A set of clauses joined by logical ANDs is called as a **Horn clause.** Each production rule constitutes a chunk of knowledge Practically speaking, to achieve modularity, the entire knowledge is broken down into fragments. **The ACTION part of one production rule becomes the CONDITION part for another production rule and this is known as the production-rule-chain or networking of rules.**

For example, the University's grading system might be reasoned as follows—

IF the marks obtained are less than 75

THEN declare the student as in First Class.

Associated with these production rules are **certainty factors/probability factors.**

For example,

IF Car doesn't work

THEN battery is down OR battery connection are poor.

In this scenario, we can use an certainty factor also and say that there is a strong evidence (0.9) that the battery is down. This factor of probability can also be used here. The range of this factor is from 0 to 1 as $0 \leq$ probability ≤ 1. This type of reasoning is also called as **inexact reasoning.** This example shows that human reasoning consists of IF-THEN rules with certainty factor attached with it. The syntax of these rules is—

IF <antecedent–1>
<antecedent–2>
⋮
<antecedent–n>

THEN
<consequence 1>|(with certainty C_1)
<consequence 2>|(with certainty C_2)
⋮
<consequence n>|(with certainty C_n)

The antecedent part is also called as a condition part or left hand side (LHS). The consequence part of a production-rule is also called as the action part or right hand side (RHS). When LHS matches then the designated condition in the production rule (action) will be executed. **Please note here that the application of rule changes the database.**

Many systems like our earlier water jug problem, OPS5 (a production system language), expert systems, SOAR (general problem solving architecture) etc. provide an overall architecture of a production system described above. It allows the programmer to write rules which define particular problems to be solved.

II Global database

It is a **central data structure** used by the production system. This database may be as simple as a small matrix of numbers or as complex as a large relational indexed file structure. **Please note, however, that this database is not the same as database of DBMS.** The global database can be accessed by all the rules. **Also remember that no part of the database is local to any of the rules in particular.** Each rule has a precondition which is either satisfied or not by the global database. If the precondition is satisfied, the rule can be applied. **Since the application of rules changes the database, so it is also called as a dynamic structure** (continuously changing due to the operation of production rules) or working memory **or short-term memory.**

III Control Structure/Strategy

It means an interpreter program to control the order in which the production rules are fixed. Also it resolves the conflicts if more than one rule becomes applicable simultaneously. **Please note that this structure applies the rules to the database, again and again, until a description of the goal state is produced. Also note that this process of identifying the rules which are tried until some sequence of them is found which produced a database satisfying the termination condition (goal) is called as a searching process.**

Efficient control strategies means enough knowledge about the problem to solve so that the rule or the sequence of rules, so selected has a good chance of being the most appropriate. The control strategies are of two types—

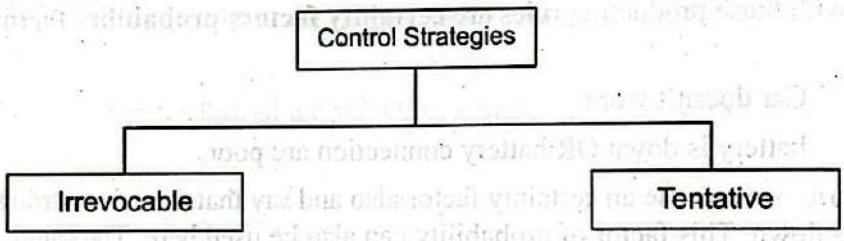


Fig. 2.6 Types of control strategies.

Let us define and explain them now, one by one.

Irrevocable Control

Here in, an acceptable rule is selected and applied irrevocably without provision for reconsideration later.

Tentative control

Herein, an applicable rule is selected arbitrarily or based on some good reason, then it (rule) is applied but with an option to return to this point in the computation to apply some other rule.

We can use two methods here. They are as follows—

(*a*) Back-tracking.

(*b*) Graph search control.

Back-tracking is a process in which the control strategy can be tentative. In this method, a rule is selected and if it does not lead to a solution, then the intervening steps are forgotten and instead another rule is selected. This back tracking technique can be used regardless of how much or how little knowledge is available for choice of rules. **Please note that if no knowledge is available, rules can be selected according to some arbitrary scheme.** Finally, the control will back track to select the appropriate rule. To reduce the frequency of back tracking to select suitable rules can be reduced and made more efficient if good rule-selection knowledge can be used.

Graph Search Control makes a provision for keeping track of the effects of several sequences of rules simultaneously. As we know that graphs (or trees) are very useful data structures to keep track of the effects of several sequences of rules. We can keep track of the various rules applied and

the database produced by a structure called as a *search* trees. **At the root of the tree is a description of the initial configuration. The links or arcs (directed) show various rules that can be applied.** It represents the states which can be reached by just one move from the initial state. This strategy keeps on generating a tree till a database is produced which satisfies the termination condition. **Please note that this strategy is inefficient as the resulting tree grows too rapidly. This happens because we show all applicable rules being applied to every state description. Also note that an intelligent control strategy would grow a tree much narrower using its special knowledge to focus the growth of the tree towards the goal.**

However, the control strategy has certain characteristics. These characteristics are as follows—

1. **It should cause motion from the initial state that must lead to a solution.**

   For example, consider the following problem.

   Problem: Water-jug problem.

   Control strategy: Start each time at the top of the list of applicable rules and apply it.

   What happens with this control strategy?

   We will continue filling indefinitely the four-litre jug with water in the first operator.

   Will this give us a solution?

   No, it doesn't give any solution as this does not cause motion from the initial state.

2. **It should be systematic**

   If in our water-jug problem, we apply the appropriate rules at random in our solution, instead of starting every time from the first rule, again and again then with the application of this **control strategy,** a tree is generated. This **control strategy** is better than the first one and leads to solution eventually. The tree gets expanded at more and more levels and needs many steps for expansion. This happened because this control strategy has been selected taking into view the local motion only i.e., considering one step at a time. It does not consider global motion i.e., taking into account a collective view of many steps. The tree is expanded continuously until the goal is reached. Now, we call this control strategy as breadth-first search (BFS). In BFS, the new states so generated at each level are put at the end of the queue which generated them. A bfs works like this—

Please note here that these strategy examples are the nodes for goal at the nth level before going to $(n + 1)^{th}$ level. The algorithm for bfs is as follows—

Step-1: Create a variable called START-LIST and set it to the initial state.

Step-2: Until a goal state is found or START-LIST is empty do:

2.1 Remove the first node (element) from the START-LIST and call it a. If START-LIST is empty, quit.

2.2 For each way that each rule can match the state described in a do:

2.2.1 Apply the rule to generate a new state i.e., if node a has successors then generate all of them.

2.2.2 If the new state is a **goal state,** quit and return this state i.e., terminate search.

2.2.3 Otherwise add the new list to the end of START-LIST.

Step-3: Goto step-2.

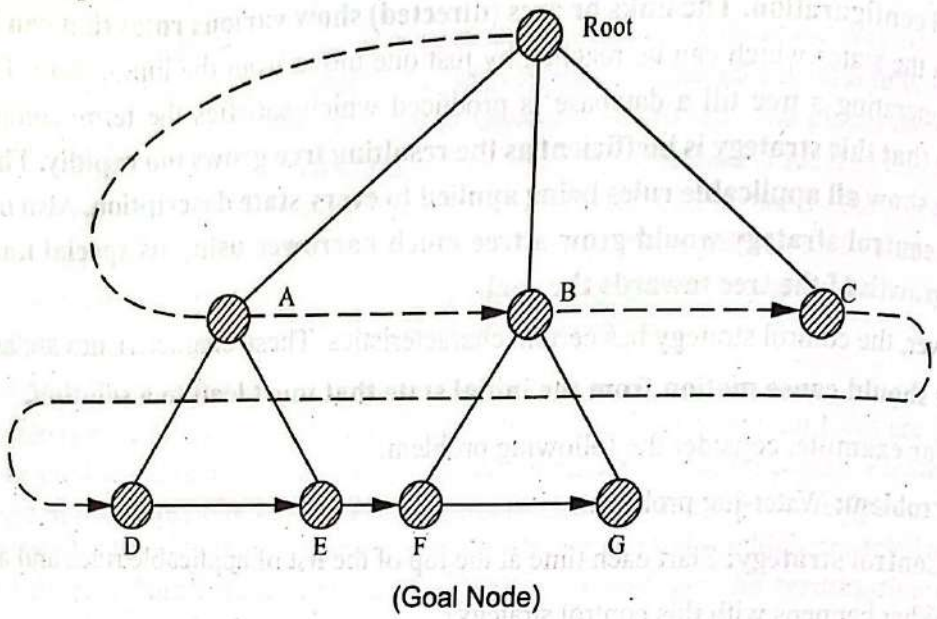


Fig. 2.7 Search tree after two levels of search.

Control strategy for putting new states in front of queue: As we already know that a queue works on a LIFO principle i.e., last-in, fist-out. Herein, a single branch of the tree is traversed until it finds a solution or until a decision to terminate the path is made. **Please note that this means that if two states A1 and A2 are produced by applying operators to a state A, then every state reachable from A1 will be examined before any state reachable from A2. Also note that if there are an infinite number of states reachable from A1, then A2 will never be examined.** So, what should be done?

In such a case, we need to impose some kind of **depth cut-off. It is defined as the maximum length of operator application-sequence.** Hence, the search tree or path is terminated when it reaches dead-end, produces previous state or it becomes larger then futility limit. **Please understand that back-tracking occurs, to the most recent created state from which alternate moves are available. A chronological backtracking** is one in which the order in which the steps are undone will depend only on the temporal sequence in which the steps were originally created. Another variant of backtracking is **dependency directed backtracking** which involves truth maintenance which uses non-monotonic reasoning and will be discussed later on. This search procedure is called as Depth-First Search (DFS). A DFS, in general, will look like this—

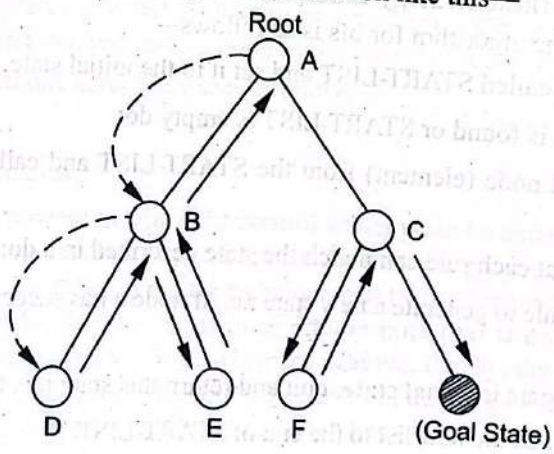


Fig. 2.8 Search tree using DFS.

We can generate such a tree for our water-jug problem also, as shown below in Fig. 2.9.

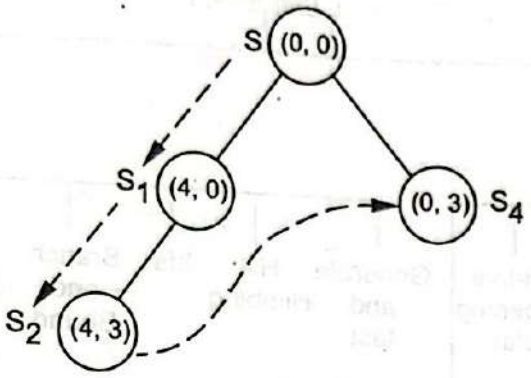


Fig. 2.9

The algorithm of dfs, in general is as follows:

Step-1: Put the initial node on a list, START-LIST.

Step-2: If (START-LIST is empty) or (START-LIST = GOAL), terminate search.

Step-3: Remove the first node from START. Call this node a.

Step-4: If (a = GOAL) terminate search with success.

Step-5: Else if node a has successors, generate all of them and add them at the beginning.

Step-6: Goto step-2.

Chronological back tracking, used in DFS involves lot of time and is not efficient also. In real problems, this can even make a problem-solving system impractical.

Non-chronological or dependency directed back tracking, those choice are withdrawn on which the dead-end depends. Herein, time does not determine which choices are to be withdrawn as it happens in chronological back tracking.

2.2 SEARCHING TECHNIQUES

As we have already discussed that searching our state space forms the core heart of AI problem solving. Once a problem is formulated, we then need to solve it. We can use this technique in many areas like theorem proving, game playing, expert systems, natural language processing etc. This involves the tasks like deduction, inferencing, planning, common sense reasoning, fuzzy logic etc.

2.2.1 Types of Searching

As we known that in a search method or technique or strategy, we firstly select one option and leave other options if this option is our final goal, (solution) else we continue selecting, testing and expanding until either a solution is found or there are no more states to be expanded. This will be determined by the search method. So, we need many different types of search algorithms. Basically, there are 2-types of searches—

1. Uniformed or blind search or unguided.
2. Informed or heuristic search or guided.

But, please note that all search techniques are distinguished by the order in which nodes are expanded. Various search strategies are shown in Fig. 2.10.

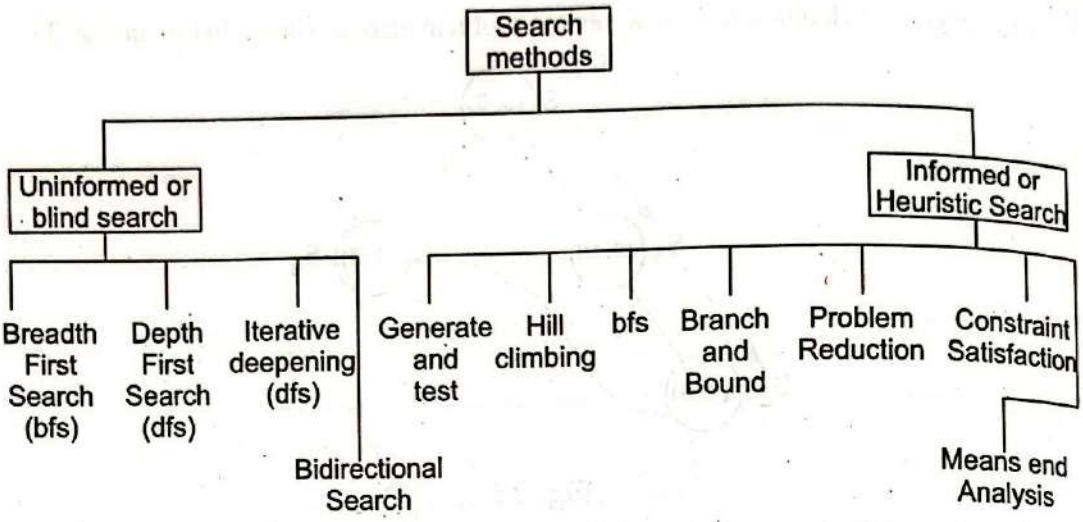


Fig. 2.10 Types of searching.

Let us now discuss these strategies one by one in detail.

2.2.2 Uniformed/Blind/Brute Force/Exhaustive Search

As we know that the state-space is a directed graph in which each node is a state and each arc shows the application of an operator transforming a state to a successor state. **But please note that a node is different from a state. A node is a data structure used to represent the search tree. However, a state corresponds to a configuration of the world.** So, nodes are on some particular paths always. They are defined by the parent node pointer. On the other hand, states are not on some particular paths. **Also note here that two different nodes can contain the same world state if that state is generated via two different search paths.** And a solution is a path from the initial or start state to a goal state. These goal states may be defined explicitly or as a set of states satisfying a given predicate. **In blind or uniformed search,** these is no order in which the solution paths are considered i.e., it uses no domain specific information to search a solution in the search space. **Again, the state space is combinational** , so this type of search is actually **impracticable for the nontrivial problems.** It is called as brute-force search as it examines every node in the tree until a goal is found. These algorithms make some assumptions:—

1. A **procedure** must be there to find all successors of a given node i.e., all states are **reachable** from the current state and that too by a single operator application.
2. The state-space graph is a **tree.** It means that there is only one start state (i.e., root of tree) and that the path from the start node to any other node is unique.
3. Whenever a node is expanded, creating a node for each of its successors, the successor nodes contain pointers back to the parent node. Finally, when a goal node is generated, the path from the root to the goal can be easily found.

So, blind search algorithms uses only the initial state, search operators and a test for a solution. It proceeds systematically by exploring nodes either randomly or in some predetermined order.

2.2.2.1 Breadth-First Search (BFS)

It is the most simple form of blind search. In this technique the root node is expanded first, then all of its successors are expanded and then their successors and so on. **In general, in bfs, all nodes are expanded at a given depth in the search tree before any nodes at the next level are expanded.**

It means that all immediate children of nodes are explored before any of the children's children are considered. The search tree generated by bfs is shown below in Fig. 2.11.

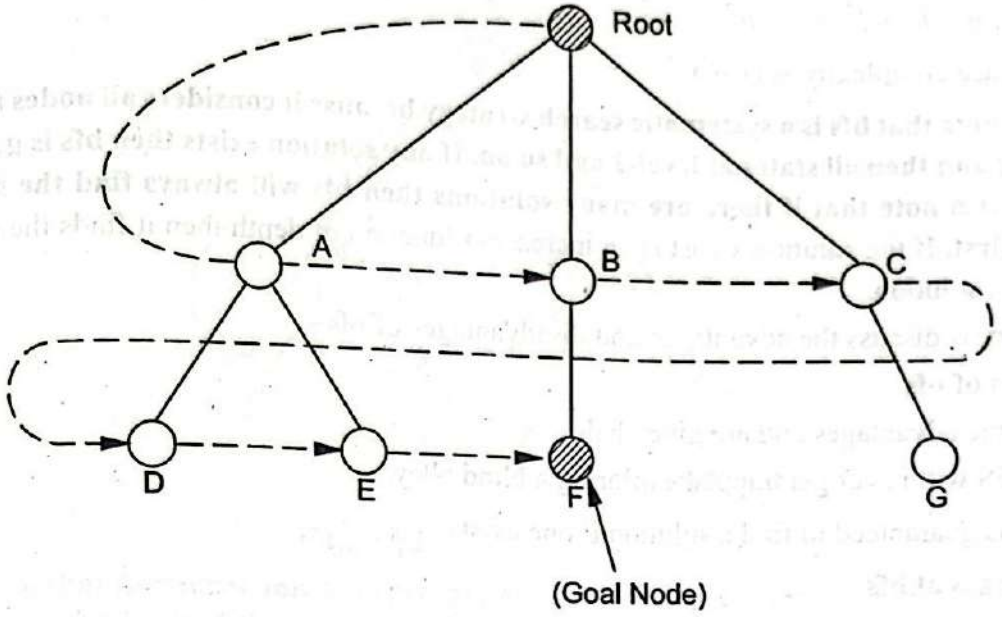


Fig. 2.11 Search tree for bfs.

Please note here that we are using the convention that the alternatives are tried in the left-to-right order. Also note that bfs is a brute-search so it generates all the nodes for identifying the goal. The breadth-first method expands nodes in order of their proximity to the start node measured by the number of arcs between them.

In general, it considers every possible sequences of length n before any sequence of length $n + 1$. Although this search may be an extremely long yet it is guaranteed eventually to find the minimum or shortest path length (possible solution sequence) solution, if any solution exists. A bfs algorithm uses a data structure-queue that works on FIFO principle. This queue will hold all generated but still unexplored nodes. **Please remember that the order in which nodes are placed on the queue for removal and exploration determines the type of search.** The bfs algorithm is as follows—

Step-1: Put the start (initial) node on a list, called OPEN of unexplored nodes. If the start node is a goal node, a solution has been found.

Step-2: If (OPEN is empty) or (OPEN = GOAL), no solution exists, so terminate search.

Step-3: Remove the first node, a, from OPEN and place it in a list called CLOSED, of expanded nodes.

Step-4: Expand node, a. If it has no successor then goto step-2.

Step-5: Place all successors of node, a, at the end of OPEN.

Step-6: If any of the successors of node, a, is a goal state then solution is found.

Time and Space complexity of bfs algorithm

The amount of time taken for generating these nodes is proportional to the depth, d and branching factor, b and is given by —

$$1 + b + b^2 + b^3 + \ldots + b^d$$

$\therefore$ Time complexity is $O(b^d)$.

Now, bfs has to remember each and every node it has generated. So, the space-complexity will also depend upon the function of depth, d and the branching factor, b.

So, space complexity is given by

$1 + b + b^2 + b^3 + \ldots + b^d$

or, Space complexity is $O(b^d)$.

Please note that bfs is a systematic search strategy because it considers all nodes (states) at level-1 first and then all states at level-2 and so on. If any solution exists then bfs is guaranteed to find it. Also note that if there are many solutions then bfs will always find the shallowest goal state first. If the solution's cost is an increasing function of depth then it finds the one which is a cheapest solution.

Let us now discuss the advantages and disadvantages of bfs—

Advantages of bfs

BFS has some advantages and are given below—

1. BFS will never get trapped exploring a blind alley.
2. It is guaranteed to find a solution if one exists.

Disadvantages of bfs

BFS has certain disadvantages also. They are given below—

1. Time complexity and space complexity are both $O(b^d)$ i.e. exponential type. This is a very big hurdle.
2. All nodes are to be generated in bfs. So, even unwanted nodes are to be remembered (stored in queue) which is of no practical use of the search.

2.2.2.2 *Depth-first Search (DFS)*

DFS begins by expanding the initial node i.e. by using an operator, generate all successors of the initial node and test them. It (dfs) is characterized by the expansion of the most recently generated or deepest node first. DFS needs to store the path from the root to the leaf node as well as the unexpanded nodes. It is neither complete nor optimal. If dfs goes down an infinite branch (depth cut-off point) will not end till a goal state is found. Even if it is found, there may be a better solution at a higher level. This **depth cut-off (d)** introduces many problems also. If d is set too shallow, goal may be missed. If d is set too deep then extra computation may be performed. The dfs algorithm is as follows—

Step-1: Put the initial node on a list, START-LIST.

Step-2: If (START-LIST is empty or (START = GOAL) then terminate (end) search.

Step-3: Remove the first node from START-LIST. Call this node, a.

Step-4: If (a = GOAL) then terminate search with success.

Step-5: Else if node a has successors, generate all of these and add them at the beginning of START-LIST.

Step-6: Goto step-2.

So, the search tree for dfs is as shown in Fig. 2.12.

Time and space complexities for dfs

The amount of time taken for generating these nodes is proportional to the depth, d and branching factor, b and is given by

$1 + b + b^2 + b^3 + \ldots b^d$

$\therefore$ Time complexity = O (b^d).

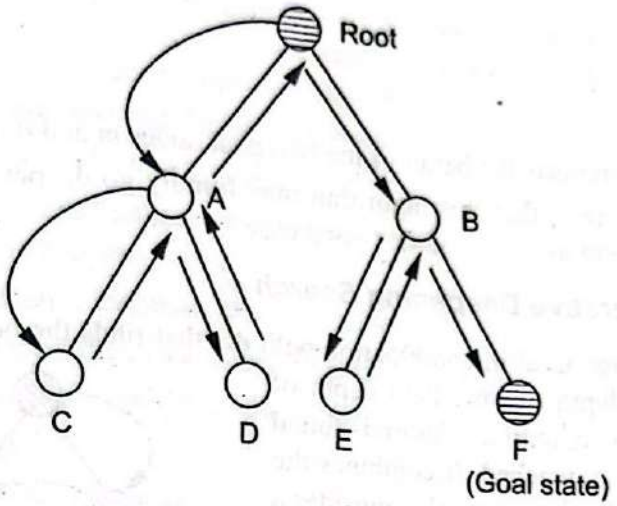


Fig. 2.12 Search tree for dfs.

Also note that dfs stores only the current path it is pursing. So, the space complexity will be a linear function of depth, d. So,

Space complexity = $O(d)$.

The value of cut-off depth (d) is important here as then otherwise the search will go on and on. Two cases arise—

1. If d is small, solution may not be found.
2. If d is large, time complexity will be more.

Advantages of DFS

DFS has some advantages. They are given below—

1. Memory requirements in dfs are less as only nodes on the current path are stored.
2. By chance, dfs may find a solution without examining much of the search space of all.

Disadvantages of DFS

This type of search can go on and on, deeper and deeper into the search space and thus, we can get lost. This is referred to as **blind alley**.

Depth first: Branch and Board

The Branch and Bound control strategy generates one path at a time. It also keeps track of the best path so found. This value is used as a bound on future candidates. As paths are constructed one at a time, the algorithm examines each partially completed path. If the algorithm determines that the best possible extension to a path branch will have greater cost than the board, it eliminates that partial path and all of its possible extensions. This reduces search considerably but still leaves an **exponential number** of paths.

This variation of depth-first, guarantees **optimal solution.** All other variations of dfs experience the undesirable property that obtained (if at all) is NOT guaranteed to be optimal. This is so because all these algorithms discontinue search once a solution is found. But this algorithm (in general) does not stop searching immediately after finding the first solution.

```
//Unbounded Dfs – Branch – and – Bound.
UDFBB(n){
if f(n) > cost (solution) then return;
if n is a goal node then
                solution: = superior(m, n);
```

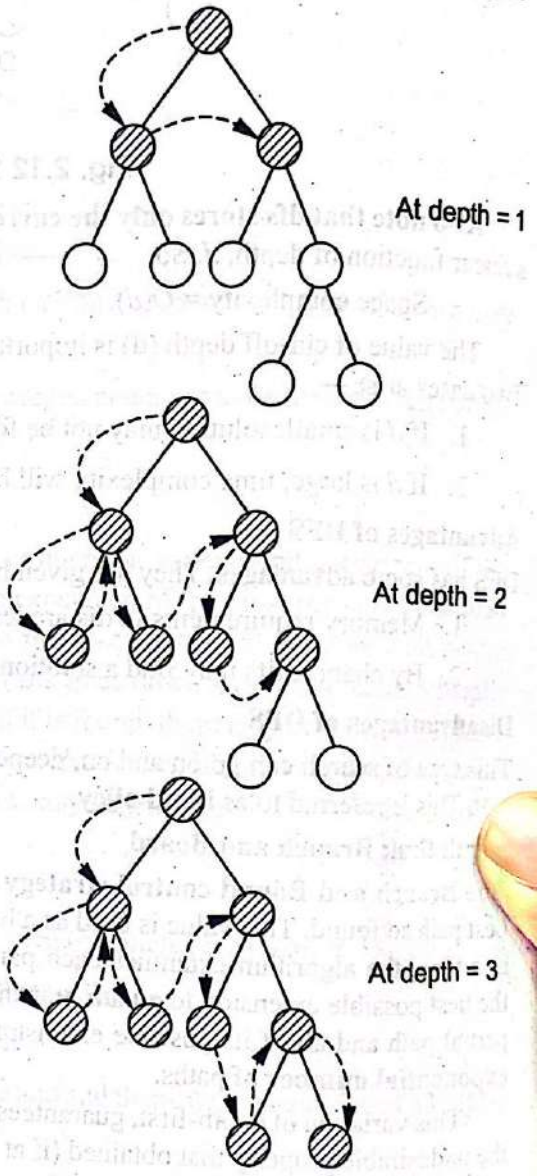
for each n_i of n do
UDFBB (n_i);
return; }

Here, superior (m, n) reports the better of the two goal nodes m and n, in terms of cost.

The **disadvantage** here is that this algorithm may blindly go deeper into some unpromising paths and thus generate to many nodes in the worst case.

2.2.2.3 *Depth-First Iterative Deepening Search*

It is a general strategy often used in combination with dfs that finds the best depth limit. It begins by performing dfs to a depth of one, then depth of two, depth of three and so on, until a solution is found or some maximum depth is reached. It combines the benefits of both bfs and dfs. For example, consider a tree as shown.



We have shown 3-stages of iterative—deepening search. **Please note here that the number of nodes expanded by iterative-deepening search is not much more than that would be using bfs.** Next we write an algorithm for depth first iterative deepening search—

Step-1: Start

Step-2: Initialize $d \leftarrow O$

Step-3: do

$d \leftarrow d + 1$;
DFID (root, d);
Until SUCCESS;

Step-4: End.

This algorithm starts with the smallest $d = 1$. It then increments d in a loop on encountering FAILURE until a SUCCES is found.

This search also guarantees to find a shortest path solution as it expands all nodes at a given depth before expanding nodes at a greater depth. **Also note that it performs wasted computations before reaching the goal depth.**

Time and space complexities

The time and space complexities of this search algorithm are also $O(b^d)$ and $O(d)$ respectively.

It can be shown that depth first iterative deepening is **asymptotically optimal**, among brute force tree searches, in terms of time, space and length of the solution. In fact, it is linear in its space complexity like dfs and is asymptotically optimal to bfs in terms of the number of nodes expanded. **Please note that in general, iterative deepening is preferred uninformed search method when there is a large search space and the depth of the solution is unknown. Also note that iterative deepening search is analogous to bfs in that it explores a complete layer going to the next layer.** It is better to develop an iterative analog to uniform – cost search, inheriting the depth –

first's guarantee to find a solution and avoiding its memory requirements. The basic idea is to use increasing path – cost limits instead of increasing depth limits. The resulting algorithm called iterative lengthening search, incurs substantial overhead as compared to uniform – cost search.

Advantages of Depth-first Iterative Deepening Search

1. It is guaranteed to find a shortest path solution.
2. It is a preferred uniformed search method when the search space is large and the depth of the solution is not known.

Disadvantages of Depth-first Iterative Deepening Search

1. It performs wasted computations before reaching the goal depth.
2. The time complexity is $O(b^d)$ i.e., exponential type only.

Is this search bfs or dfs?

This search is similar to both bfs as well as dfs.

How bfs? It is similar to bfs in that it explores a complete layer of new nodes at each iteration before going to the next layer.

How dfs? It is similar to dfs for a single iteration.

Why is it called as iterative?

This search terminates at the depth, d on each iteration if no goal has been found, removes all nodes from the queue, increments d by one and initiates the search again. Hence, it is called as an **iterative search.**

2.2.2.4 *Bidirectional Search*

This search is used when a problem has a single goal state that is given explicity and all the node generation operators have inverses. Searching is done by searching forward from the initial node and backward from goal node simultaneously. So, the program must save (store) the nodes so generated on both search frontiers until a common node is found. Both dfs and bfs can be modified slightly to perform this search.

Time and Space Complexities of bidirectional Search

To perform this search to a depth of k, the search is made from one direction and the nodes at depth k are stored. At the some time, a search to a depth of k and $k + 1$ is made from the other direction and all nodes generated are matched against the nodes stored from the other side. This process is repeated for lengths $k = 0$ to $d/2$ from both the directions. So, its **time and space complexities are both $O(b^{d/2})$ when node matching is done in constant time per node.**

Advantages of bidirectional search

1. It is used to solve the 8-puzzle problem.
2. Only slight modification of dfs and bfs can be done to perform this search.

Disadvantages of bidirectional search

1. Since the number of nodes increases as b^d with depth d, so such problems becomes intractable for large depths.
2. All node generation operators must have inverses also as only then this type of search can be used.

2.2.2.5 *Beam Search*

1. It is like Bfs in that it progresses level by level.
2. But beam search **moves downward** only through the best, W nodes at each level by applying heuristics, the other nodes are ignored.

3. Consequently, the number of nodes explored remains manageable even if there is a great deal of branching and the search is deep.
4. Whenever beam search is used, there are only W nodes under consideration at any depth rather than the exponentially explosive number of nodes as is done in bfs.
5. The **width of the beam** is fixed and whatever be the depth of the tree, the number of alternatives to be scanned is the product of the width of the beam and the depth.

2.2.3 Informed/Heuristic Search

The term **heuristic** is used for algorithms which find solutions among all possible ones. Recall that heuristic is a rule of thumb or judgment technique that leads to a solution but provides no guarantee of success. Still heuristic play an important role in searching process because of the exponential nature of most of the problems. **Please note that they help to reduce the number of alternatives from an exponential number to a polynomial number.**

So, we get a solution in a reasonable amount of time. The additional information about the properties of the specific domain which is built into the state and operator definitions is called as **heuristic information.**

And a search using this heuristic information is called as a **heuristic or informed search. A heuristic function determines a number which determines how good or bad a node can be.** If ev(n) is a best solution state and ev(n, p) is a best path then a heuristic is any information which shows how good or bad that path is likely to be so, we can expand that good node and can generate only its children. There is no blind expansion. So, pruning of the search tree occurs. In nutshell, the heuristics needed to solve problems are expressed as a heuristic function which maps the problem states into numbers. These numbers are then appropriately used to guide search. Heuristic search methods use some general purpose search techniques. The most simplest is the generate And-Test method which is discussed next.

2.2.3.1 Generate-And-Test Algorithm

It is a weak method meaning that it helps usually but not always.

It may or may not overcome the combinational explosion problem of AI. This algorithm works in two modules—

(*a*) **Generator module:** It creates the possible solutions.

(*b*) **Tester module:** It tests (or evaluates) each of the proposed solution either accepting or rejecting that solution.

Please understand that here an action may stop when one acceptable solution is found or action may continue until all possible solutions are found. Let us develop its algorithm now—

Step-1: **Generate** a possible candidate solution which can either be a point in the problem space or a path from the initial state.

Step-2: **Test** to see if this possible solution or candidate solution is a real solution by comparing the state reached with the set of goal states.

Step-3: If it is a real, acceptable solution then return, success else repeat from step-1 again (as it is a failure).

This algorithm is basically a dfs procedure only as complete solutions must be created before testing. **Please note that when it is used in a systematic form then it is simply an exhaustive search of the problem space.** It can also operate by generating solutions randomly and hence is often called as a **British Museum method.** However, in this method also there will be no guarantee that a solution will ever be found. So, we need a heuristic to sharpen the search. But this is useful for simple problems and is in efficient for problems having large state space search.

2.2.3.2 Hill climbing

It is so called because of the way the nodes are selected for expansion. At each point in the search path, a successor node that appears to lead most quickly to the top of the hill (goal) is selected for exploration. In this method of searching, the generate-and-test method is augmented by a heuristic foundation which measures the closeness of the current state to the goal state. Hill climbing can be employed if the path to the goal does not matter. It is a local search method in which we consider the state space landscape. This is shown in Fig. 2.12.

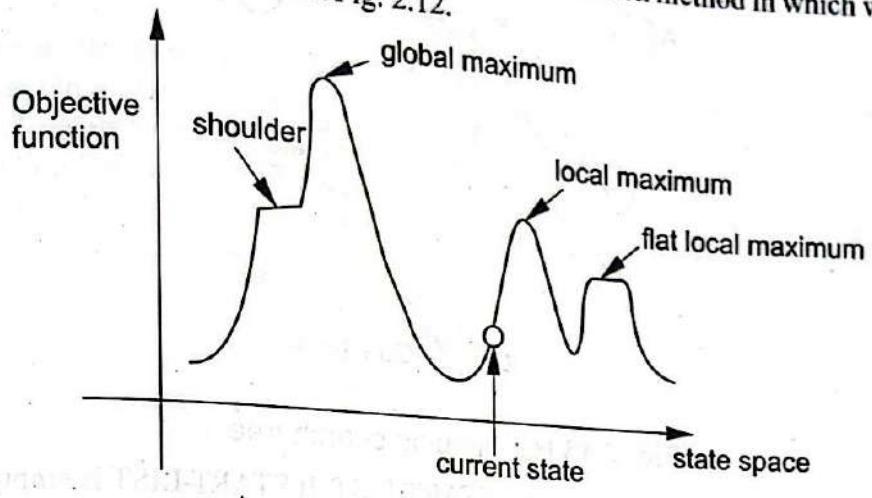


Fig. 2.12 State space landscape for local search.

A landscape has both "location" (defined by the state) and "elevation" (defined by the value of the heuristic cost function or objective function). The aim is to find **global minimum** (lowest valley) if the elevation corresponds to heuristic cost function and if the elevation corresponds to the objective function then the aim will be to find the highest peak or **global maximum.**

NOTE: You can convert from one to the other just by inserting a minus sign.

Local search algorithm explores this landscape. A **complete local search algorithm** always finds a goal if one exists. An **optimal algorithm** always finds a global minimum/maximum.

Hill climbing search modifies the current state to try to improve it. This is shown in Fig. 2.12 by an arrow. It is in this way that the cost function provides a task-specific knowledge into the control process.

So, a search tree is generated with the help of heuristic function. As explained earlier also, **hill climbing is like dfs** where the most promising child is selected for expansion. When the children have been generated, alternative choices are evaluated using the heuristic function. The path that appears most promising is then chosen. No further reference to the parent or other children is retained. **Please note that there is practically, no difference between hill-climbing and dfs except that the children of the node that has been expanded are sorted by the remaining distance.** Let us now write an algorithm for hill-climbing—

Step-1: Put the initial node on a list, START-LIST.

Step-2: If (START-LIST is empty) or (START-LIST is GOAL) then terminate search.

Step-3: Remove the first-node from the START-LIST. Call this node, n.

Step-4: If (n = GOAL) then terminate search with SUCCESS.

Step-5: Else if node, *n*, has successors then generate all of them. Find how far they are from the goal node. Sort them by the remaining distance from the goal and add them to the beginning of START-LIST.

Step-6: Goto step-2.

Let us consider a tree and apply our algorithm now.

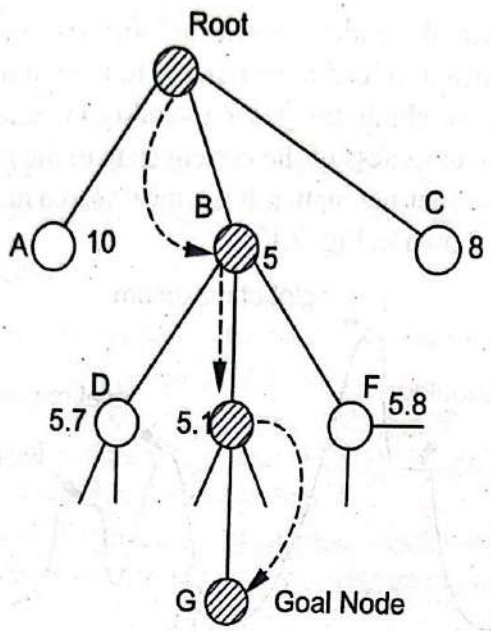


Fig. 2.13 Hill climbing search tree.

Here, we put the initial node i.e. root on a START-LIST. If START-LIST is empty or if START-LIST is our goal itself then the search terminates. Else we remove this node (n) from the START-LIST and generates its successors. We sort this START-LIST on the basis of the distances from the goal (see figure) and keep on repeating this process till a goal is found. **Please note that a new state has to be better, meaning that if we consider the cost function then it means a lower value and if we consider the objective function then it means a higher value.** If the algorithm is constantly following the direction which gives the fastest rate of increase then it is called as the **steepest ascent hill climbing.** But if the lowest cost is best then it is a **down hill algorithm. Also note that this algorithm is called as a steepest rise with back-tracking search method because it is prepared to back up and try a different path if the steepest one leads to a dead end.** Hill climbing is sometimes called as a **greedy local search** because it grabs a good neighbour state without thinking ahead about where to go next. But hill climbing suffers from following problems—

I Local maximum

It is a state which is better than all of its neighbours but is not better than some other states which are farther away. **Please note that at local maxima, all moves appear to make the things worse.** They are sometimes frustrating also as they often occur almost within slight of a solution. They are also called as **foot-hills.** It is shown in Fig. 2.14.

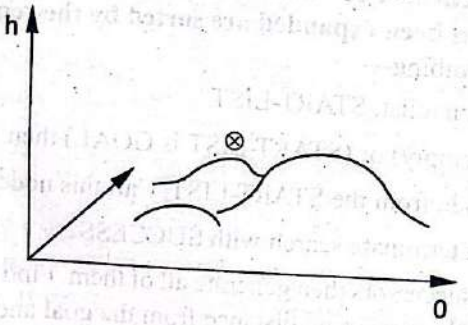


Fig. 2.14 Local maximum or (foot hill).

Solution to this problem

1. One possible solution is **backtracking.**

   We can backtrack to some earlier node and try to go in a different direction to attain the global peak. We can maintain a list of paths almost taken and go back to one of them if the path that was taken leads to a dead end.

2. Another solution can be a list of promising plan.

II Plateau

It is a flat area of the sarch space in which a whole set of neighbouring states (nodes) have the same value. On a plateau, it is not possible to determine the best direction in which to move by making local comparisons. Actually, plateau is an area of the state space landscape where the evaluation function is flat. It can be a **flat local maximum** (from which no uphill exist exists) or a **shoulder** from which it is possible to make progress. A plateau is shown in Fig. 2.15.

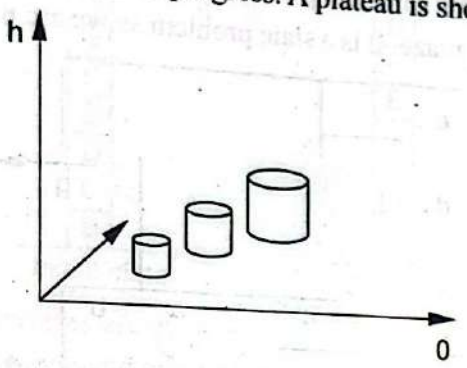


Fig. 2.15 Plateau

Solution to this problem

1. A **big jump** in some direction can be done in order to get to a new section of search space. This method is recommended as in a plateau all neighbouring points have the same value.
2. Another solution is to apply small steps several times in the same direction. This depends on the rules available.

III Ridge

It is a special kind of local maximum. It is an area of the search space which is higher than the surrounding areas and that itself has a slope. This is shown in Fig. 2.16.

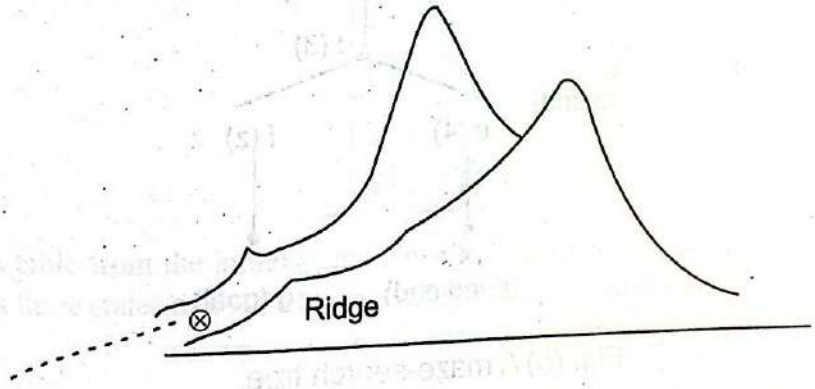


Fig. 2.16 Ridge.

We cannot travel the ridge by single moves as the orientation of the high region compared to the set of available moves makes it impossible.

Solution to Ridge problem

1. Trying different paths at the same time is a solution. We can apply two or more rules before doing the test. This corresponds to moving in several directions at once.

2. Bidirectional search can be useful here.

Advantages of Hill climbing

It can be used in continuous as well as discrete domains.

Disadvantages of Hill climbing

1. It is not an efficient method.
2. It is not suited to problems where the value of the heuristic function drops off suddenly when the solution may be in sight.
3. It is a local method as it looks at the immediate solution and decides about the next step to be taken rather than exploring all consequences before taking a move.

We are in a position to solve some examples now.

EXAMPLE 1. Consider a bullet-maze problem.

The figures below shows a maze. It is a state problem as we are not keen in the shortest path but in the goal state only.

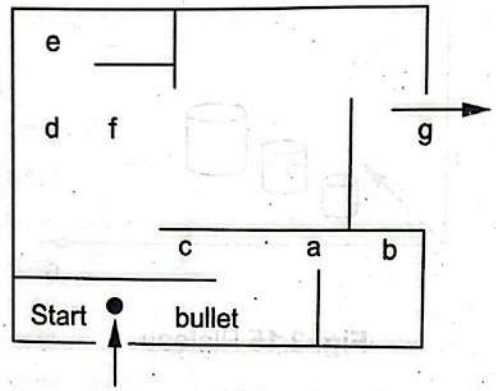


Fig. (*a*) Maze

You are also given a maze-search tree as follows—

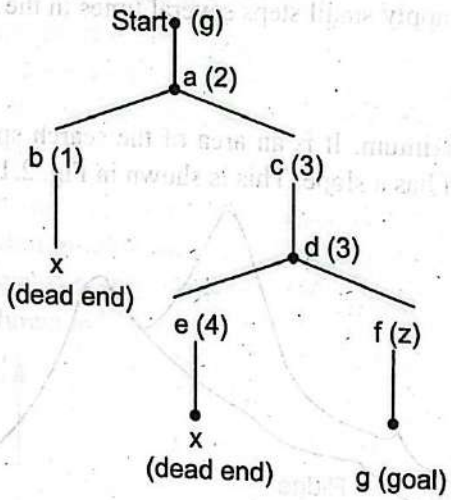


Fig. (*b*) A maze-search tree.

(Fig. a)—The start is marked with a bullet and the exist (goal state) is marked as g. Other letters mark the choice points in the maze.

(Fig. b)— The numbers in brackets show the heuristic evaluation function for each node.

The evaluation function chosen is the distance measured from the node to the goal.

Explain how it leads to dead ends?

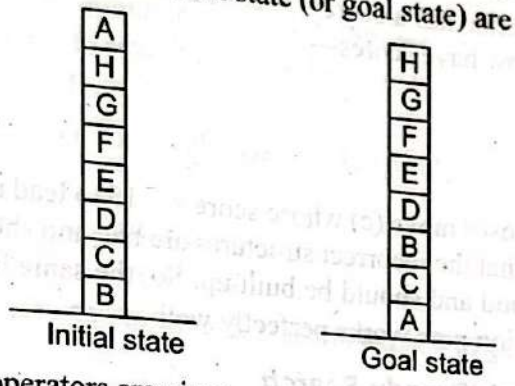
Which node is local minima? Is there a plateau problem?

SOLUTION 1. At node 'a' (fig. b) it appears at first that 'b' is the most promising direction. But this leads to a dead end.

From node 'b', whatever path we take, takes us away from the goal. Hence, 'b' is called as **local minimum**. A simple search will stop at 'b' and never reach the goal state (g) which is global minimum.

The value of heuristic function does not change between 'c' and 'd'. That is, it is 3. So, there is no progress. What if our search space is large? There may be a whole area of the search space with no change of heuristic. When this happens, our heuristic fails to give any guidance about possible direct path.

EXAMPLE 2. Consider the **block-world problem**. It has 8 blocks, A to H which are similar and equal to each other. There initial state and final state (or goal state) are as follows:



Also the following two operators are given —

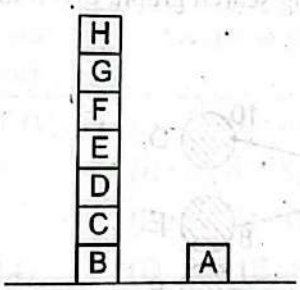
01 : Pick up one block and put it on table.

02 : Pick up one block and put it on another.

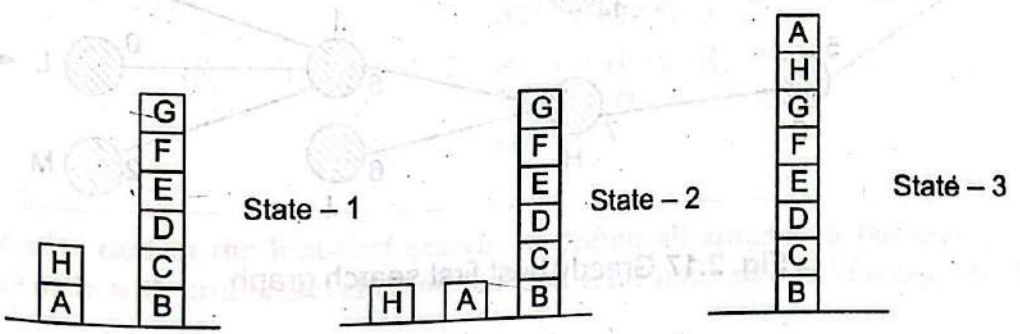
Let the heuristic function be—Add one point for every block which is resting on the thing it is supposed to be resting on. Subtract one point for every block which is sitting on the wrong thing.

Correspondingly, the initial state has a score of 4. Goal state has a score of 8. **What problems can be faced if this problem is solved? Can you define the heuristic function globally taking the whole structure of block as a single unit?**

SOLUTION 2. The most natural move could be to move block-A onto the table. This is an intermediate state and is shown below—



Moving A on table from the initial state (above fig.) has a score of 6. This move is allowed and this state produces three states as shown below with scores of 4 each (using our operators—01 and 02)



Now, hill climbing will stop because all these states have the same score. **Please note that they have a produced a score of 4 which is less than that of the intermediate state with score of 6** (from where these states arised).

We say now that the process has reached a local maxima. Our problem is that the current state appears to be better than any of its successor states as more blocks rest on the correct objects. To solve this problem, we disassemble a good local structure as it is wrong in the global context. This fault is inherent in our heuristic function now—A global heuristic function is defined which takes the whole structure of blocks as a single unit.

For each block which has the correct support structure i.e., if the complete structure below it is exactly as it should be then add one point for every block in the support structure. For each block which has an incorrect support structure, subtract one point for every block in the existing support structure. So, now the goal state has a score = – 28. The initial state has the score = – 21. The three states produced from this now have scores—

$a = -28$

$b = -16$

$c = -15$.

Thus, hill climbing chooses move (c) whose score = – 15 to lead us to a solution. This modified heuristic function assumes that the incorrect structures are bad and should not be accepted. And that the correct structures are good and should be built up. So, the same hill-climbing procedure which failed earlier heuristic function now works perfectly well.

2.2.3.3. Best-First Search/Greedy Search

Actually, best-first search is a combination of depth-first (dfs) and breadth-first (bfs) search methods. The approach used here is to select that single path at a time which is more promising one than the current path. The most promising node is chosen for expansion i.e., all of its successors are generated. Again the most promising node is chosen. **Please note that this changes the search from dfs to bfs. Also note that if we find that after the expansion, no node is more promising then we can backtrack.** This search uses again a heuristic function called as an **evaluation function** which is an indicator of how far a node is from the goal node.

Please remember that the goal nodes have an evaluation function value of zero.

Let us take an example to understand best-first search now.

EXAMPLE. Consider the following search graph, given in Fig. 2.17.

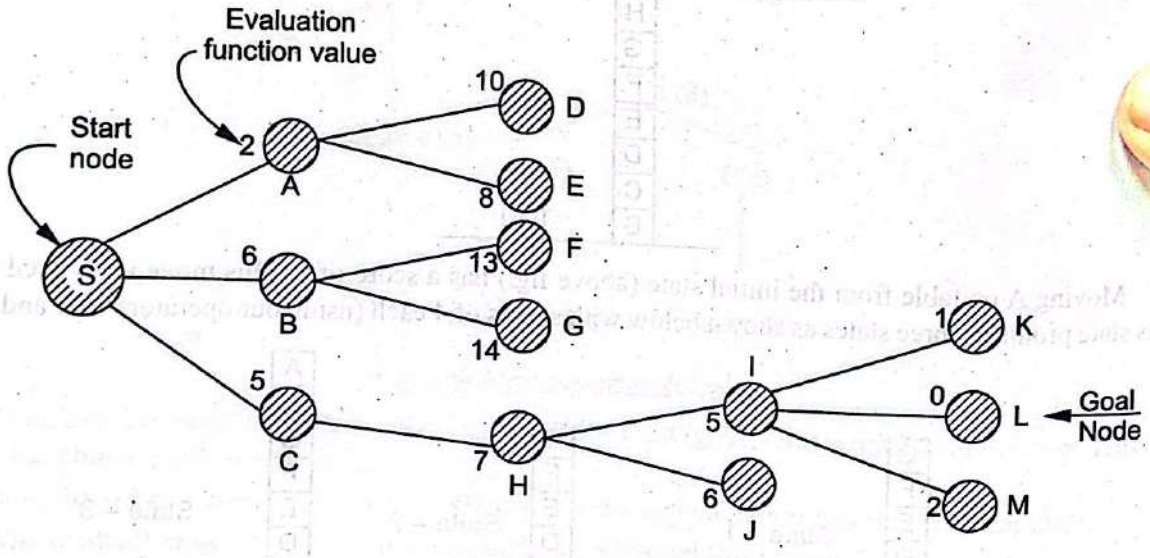


Fig. 2.17 Greedy best first search graph.

Working: We start with a start-nodes, S. Now, S has three children i.e., A, B and C with their heuristic function values 2, 6 and 5 respectively. These weights show approximately, how far they are from goal node. So, we write, children of S are —

(A : 2), (B : 6), (C : 5)

Out of these, the node with minimum value is (A : 2). So, we select A and its children are explored (or generated).

It's children are—

(D : 10) and (E : 8)

The search process now has four nodes to search for, namely—

(B : 6), (C: 5), (B: 10) and (E : 8)

Out of these, node-C has the minimal value of 5. So, we select it and expand. So, we get (H : 7) as its child. Now, the nodes to search are as follows—

(B : 6), (D: 10, (E: 8) and (H : 7)

Out of these, node-B is minimal so, we choose it and expand it to get its children—

(F : 13) and (G : 14)

At this point of time, the nodes available are :

(D : 10), (E : 8), (H : 7), (F : 13), (G : 14)

Out of these, minimum is (H : 7). So, we select it and expand to get—

(I : 5) and (J : 6)

So, nodes available for search now are —

(D : 10), (E : 8), (F :13), (G : 14), (I : 5), (J : 6)

Out of these, minimum is (I : 5), so we expand it now.

It's children are—(K : 1), (L : 0) and (M : 2).

So, available nodes are—

(D : 10), (E : 8), (H : 7), (F : 13), (G : 14)

(J : 6), (K : 1), (L : 0) and (M : 2)

Out of these, minimal is (L : 0) which is our goal-node. So, search stops here. Let us tabulate this in a tabular form now.

Step	Node being expanded	Children (on expansion)	Available nodes (to search)	Node Chosen
1.	S	(A : 2), (B : 6), (C : 5)	(A : 2), (B : 6), (C : 5)	(A : 2)
2.	A	(D : 10), (E: 8)	(B : 6), (C : 5), (D : 10), (E : 8)	(C : 5)
3.	C	(H : 7)	(B : 6), (D : 10) , (E : 8), (H : 7)	(B : 6)
4.	B	(F : 13), (G : 14)	(D : 10), (E : 8), (H : 7), (F : 13), (G : 14)	(H : 7)
5.	H	(I : 5), (J : 6)	(D : 10), (E : 8), (F : 13) (G : 14), (I : 5), (J : 6)	(I : 5)
6.	I	(K : 1), (L : 0), (M : 2)	(D : 10), (E : 8), (H : 7), (F : 13), (G : 14), (J : 6),(K : 1), (L : 0), (M : 2)	Goal node is found. So, search stops now.

Please note that in the best-first search, we 'jump all around' in the search graph to identify the node with minimal evaluation function value (indicated in braces). Also note that

the paths found by this search algorithm are likely to give faster solutions because it expands a node that seems to be closer to the goal. But there is no guarantee on this.

Comparison of hill climbing and best-first searches

In hill climbing search, we sort the children of the first node being generated. On the other hand, in **best-first search,** we have to sort the entire list to identify the next node to be expanded. Unlike hill climbing, this search retains all estimates computed for previously generated nodes and makes solution based on the best among them all.

Comparison of Best-first search and depth-first search

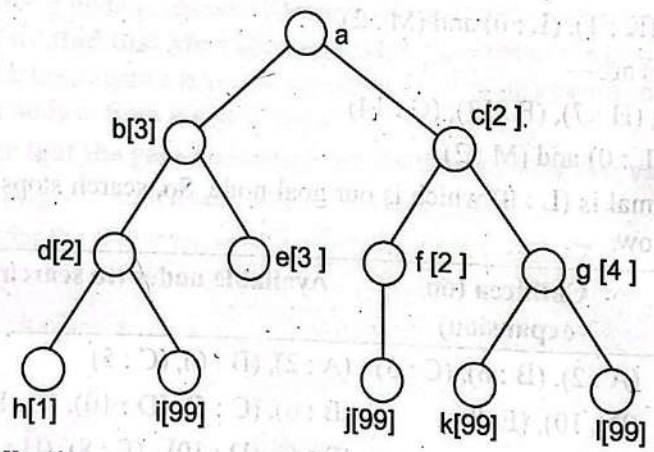
Best-first search resembles depth-first search in the way it prefers to follow a single path all the way to goal but will back up when it hits a dead end. It suffers from the same defects as depth first search i.e., it is non-optimal and is incomplete because it can go all along an infinite path and will never return to try other path possibilities.

We develop an algorithm for Best-first search now.

1. Place the starting node S (initial state) on the queue.
2. If the queue is empty, return failure and stop.
3. If the first element on the queue is a goal node g, return success and stop. Otherwise,
4. Remove the first element from the queue, expand it and compute the estimated goal distances for each child. Place the children on the queue (at either end) and assign all queue elements in ascending order corresponding to goal distance from the front of the queue.
5. Return to step 2.

We are in a position to solve some examples now.

EXAMPLE 1. Consider the search tree given below—



Show how the following searches are performed—

(*a*) **Hill climbing**

(*b*) **Best-first search?**

SOLUTION 1.

(*a*) **Using Hill Climbing**

The heuristic function value is an estimate of the distance to the goal and is shown in brackets. **Please note that lesser is its value, the better it is**

Using hill climbing we get—

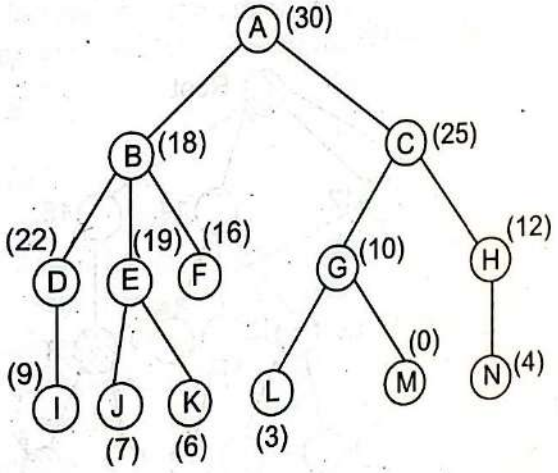
1. Start at a (start-node)
2. Children of a = [b(3), c(2)]

3. Out of these best child is c(2).
4. Children of c = [f(2), g(4)]
5. Best child is f(2) now.
6. Children of f = [j(99)]—not a goal.
7. Backtrack to g through c.
8. Children of g = [k(99), l(3)]
9. Best child, *l* is goal node.

(b) **Using Best-first search**

1. Start at a, the start node
2. Children of a [b(3), c(2)]
3. Best-first = c
4. Children of c = [f(2), g(4)]
5. Consider [f(2), g(4), b(3)]
6. Best-first = [f(2)]
7. Children of f = [j(99)]
8. Consider [j(99), g(4), b(3)]
9. Best first = b(3)
10. Children of b = [d(2), e(3)]
11. Consider [d(2), e(3), j(99), g(4)]
12. Best first = d(2)
13. Children of d = [h(1), i(99)]
14. Consider [h(1), i(99), e(3), j(99), g(4)]
15. Best-first-h, is a good node

EXAMPLE 2. Using search tree given below, list the elements of the queue just before the next node is expanded. Use best first search where the numbers correspond to estimated cost-to-goal for each corresponding node.



SOLUTION 2. The table shows the cost from the initial node to the goal node

From Node	To Node	Cost of Node	Possible Dueue	Cost so far	Final Best First Dueue
A	B	30	A – B	48	✓
	C	30	A – C	55	×
B	D	18	A – B – D	70	×
	E	18	A – D – E	97	×
	F	18	A – D – F	64	✓
F	Next Node	16	A – B – F – Next Node	64	✓

The path is A – B – F – Next node in the given trce.

2.2.3.4 *Branch and Bound Search*

This strategy applies to problems having a **graph search space where more than one alternative path may exist between two nodes.** This strategy saves all path lengths (or costs) from a node to all generated nodes and chooses the shortest path for further expansion. It then compares the new path lengths with all old ones and again chooses the shortest path for expansion. In this way, any path to a good node is certain to be a minimal length path.

The time required depends on the order in which the paths are explored. But this method is inadequate for solving larger problems.

It's **algorithm** is as follows—

1. Form a queue of partial paths. Let the initial queue be of zero-length, zero-step path from the root node to no where.
2. Until (queue is empty) or (goal is found) find if the first path in the queue reaches the goal node.
   (*a*) If the first path reach the goal node, do nothing.
   (*b*) If the first path does not reach the goal node then.
      (*i*) Remove the first path from the queue.
      (*ii*) Form new path from the removed path by extending one step.
      (*iii*) Sort the queue in increasing order of the cost accumulated so far.
3. If the goal has been found then return SUCCESS else return FAILURE.
4. End Branch-and-Bound algorithm.

The search graph is shown in Fig. 2.18.

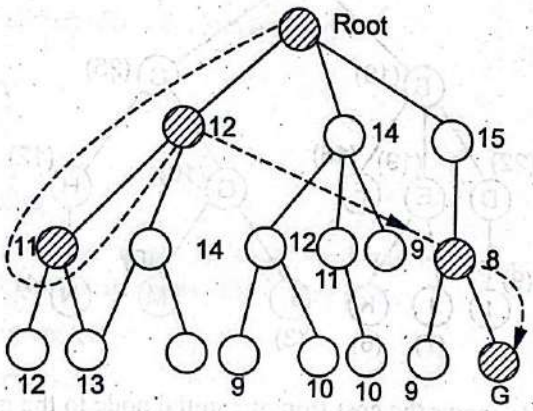


Fig. 2.18 Branch-and-bound search.

Using Branch-and-Bound on Traveling Salesman Problem (TSP)

Let us assume that there are a number of cities to be visited. From the starting city, the algorithm will begin generating complete paths. But it has to keep track of the shortest path found so far. **Please note that no further exploration of any path will be done if its partial length becomes greater than the shortest path found so far.** This technique still guarantees a shortest path for our TSP problem.

2.2.3.5 A* Algorithm

Basic principle—"Sum the cost and the evaluation function values for a state to get its "goodness" worth and use this as a yardstick instead of the evaluation function value in best first search. The sum of the evaluation function value and the cost along the path leading to that state is called as a fitness number." Before discussing about the fitness number, let us see what is **evaluation function value** and the **cost functions**. In best-first search, we use **evaluation function value** i.e., a value that estimates how far a particular node is from the goal. On the other hand, **cost functions** show how much resources like time, energy, money etc. have been spent in reaching a particular node from the start. While evaluation function values deal with the future, cost function's values deal with the past. **However, please note that the cost function values are more concrete than the evaluation function values as the cost function values are really expanded. Also note that if it is possible to obtain both, the evaluation function values and the cost function values then A* algorithm can be used.**

What is a fitness number?

A fitness number, $f'(n)$, is a heuristic function that estimates the merits of each node that we generate. So, this algorithm can search more promising paths first. **Dash** in the function indicates that it is an approximation to a function, $f(n)$, that gives the actual or true evaluation of the node. It is given as follows—

$$\boxed{f'(n) = g'(n) + h'(n)} \quad \ldots(1)$$

where $f'(n)$ – is the estimated cost of the cheapest solution through n.

$h'(n)$ – is an estimate of the additional cost of getting from the current node to a goal state.

$g'(n)$ – is an estimate of getting from the initial node to the current node.

The corresponding true values are $f(n)$, $g(n)$ and $h(n)$ of $f'(n)$, $g'(n)$ and $h'(n)$ respectively.

For state space tree problems,

$$\boxed{g'(n) = g(n)} \quad \ldots(2)$$

Because there is only one path and the distance $g'(n)$ will be known to be true minimum from the start to the current node n. In case of graphs, there can be alternate paths from the start node to n. So, for state space tree problem, we can also represent the heuristic function as —

$$\boxed{f'(n) = g(n) + h'(n)} \quad \text{(from equation –1 and 2)}$$

How to select $h'(n)$ and $g(n)$?

The function $g(n)$ lets us choose which node has to be expanded next not only on the basis of how good the node itself looks [as measured by $h'(n)$] but also on the basis of how good the path to the node was. By introduction of $g(n)$ into $f'(n)$ we will not be always choosing the node to expand that appears to be closest to the goal. This is important in cases where we care about the path we find if our concern is only getting to a solution by any means we can define $g(n) = 0$. In this way we will be choosing the node that seems closest to the goal. On the other hand, if we want to find a path involving lowest number of steps, then we can set the cost of going from a node to its successor as a constant (usually 1). In this way A* algorithm can be used whether we want to find a **minimal-cost overall path or simply any path as quickly as possible.**

$h'(n)$ is a measure of the cost of getting from the node to a solution. **Therefore, good nodes get low values and bad nodes get high values.** Hence, minus sign has to judiciously placed. This is opposite to that $g(n)$ which has to be non-negative. If $g(n)$ has a negative value then paths that traverse cycles in the graph will appear to get better as they get longer. Now, if $h'(n)$ estimates $h(n)$ perfectly, then A* will converge immediately to the goal with no search. If it is always zero, the search strategy will be random. If the value of $g(n)$ is always one, the search will be breadth first. Now a question comes to our mind that what happens if $h'(n)$ is neither perfect nor zero. There are two possibilities. One is that overestimates $h(n)$ and the other is that it underestimates $h(n)$. Let us now examine what happens in both the cases.

Case I: $h'(n)$ underestimates $h(n)$

Consider the situation as shown in Fig. 2.19. Assuming that the cost of all arcs is 1, Fig. 2.19 (a) shows that initially all nodes except A are on OPEN. For each node, $f'(n)$ is indicated as the sum of $h'(n)$ and $g(n)$. From the figure we observe that node B has the lowest $f'(n)$ which is 4. So it is expanded first. Suppose B has only one successor E and appears to be three moves away from a goal. Now $f'(E)$ is 5 which is same as $f'(C)$. Suppose we decide to allow the same path which we are currently following. Then, we will expand E next. Suppose it too has a single successor F which is three moves from the goal. This has been shown in Fig. 2.19 (b), i.e., the situation two steps later after B and E have been expanded. Now, $f'(F) = 6$ which is greater than $f'(C)$. So we will expand C next. Thus, we see that by underestimating $h'(B)$ we have wasted some effort. Thus, we find that B was farther away than we initially thought and we go back to try another path.

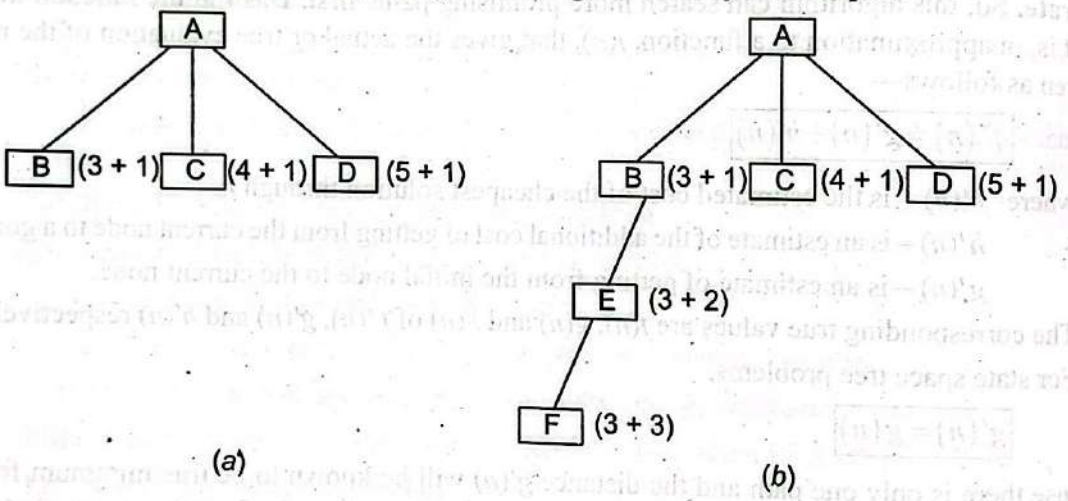


Fig. 2.19 $h'(n)$ overestimates $h(n)$.

Case II: $h'(n)$ overestimates $h(n)$

Now consider the situation shown in Fig. 2.20.

Here also we expand B in the first step. At the second step, we again expand E. At the next step we expand F and finally we generate G. The solution path has length 4. But suppose if there was a direct path from D to a solution giving a path of length 2, we will never find it. Thus, by overestimating $h'(D)$ we make D look so bad that we may find some other solution which is worse without even expanding D.

In general, we can say that if $h'(n)$ overestimates $h(n)$, we cannot be guaranteed of finding the cheapest path solution unless we expand the entire graph until all paths are longer than the best solution.

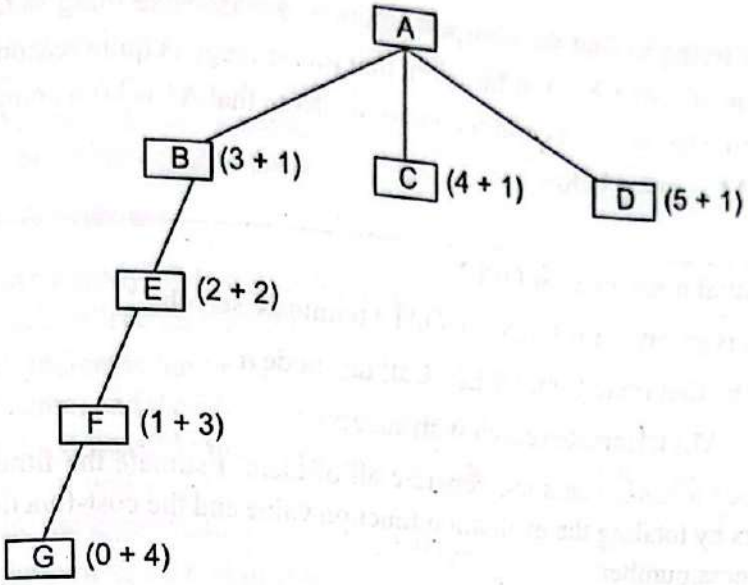


Fig. 2.20 $h'(n)$ overestimates $h(n)$.

Now we will see how the $f'(n)$ function is evaluated in A* search. Consider Fig. 2.21 again with the same evaluation function values as in Fig. 2.17. Now associated with each node are three numbers, **the evaluation function value, the cost function value and the fitness number.**

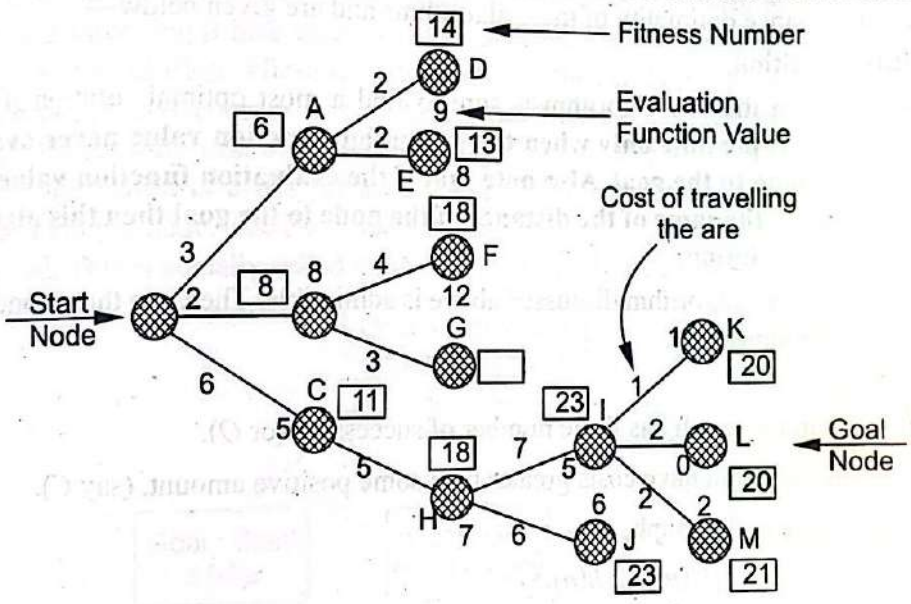


Fig. 2.21 A sample tree with fitness number used for A* search.

The fitness number, as stated earlier, is the total of the evaluation function value and the cost-function value. For example, for node K the fitness number is 20, which is obtained as follows:

(Evaluation function of K) + (cost function involved from start node S to node K)

= 1 + (Cost function from S to C + Cost function from C to H + Cost function from H to I + Cost function from I to K)

= 1 + 6 + 5 + 7 + 1 = 20.

While best-first search uses the evaluation function value only for expanding the best node, A* uses the fitness number for its computation.

Thus, if we are trying to find 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 quite reasonable provided that the heuristic function $h'(n)$ satisfies certain conditions. Note that A* is both complete and optimal.

Algorithm for A* is given below:

Algorithm for A*

1. Put the initial node on a list OPEN
2. If (OPEN is empty) or (OPEN = GOAL) terminate search
3. Remove the first node from OPEN. Call this node *a*
4. IF (*a* = GOAL) terminate search with success
5. Else if node a has successors, generate all of them. Estimate the fitness number of the successors by totaling the evaluation function value and the cost-function value. Sort the list by fitness number.
6. Name the new list as CLOSED.
7. Replace OPEN with CLOSED.
8. Goto Step 2.

Let us now discuss some common properties of heuristic search algorithms. These properties evaluates the performance optimality of these algorithms and are given below—

1. Admissibility condition

By algorithm we mean that the algorithm is sure to find a most optimal solution if one exists. **Please note that this is possible only when the evaluation function value never overestimates the distance of the node to the goal. Also note that if the evaluation function value which is a heuristic one is exactly the same of the distance of the node to the goal then this algorithm will immediately give the solution.**

For example, the A* algorithm discussed above is admissible. There are three conditions to be satisfied for A* to be admissible.

They are as follows—

1. Each node in the graph has finite number of successors (or O).
2. All arcs in the graph have costs greater than some positive amount, (say C).
3. For each node in the graph, n,

$$h'(n) \leq h(n).$$

This implies that the heuristic guess of the cost of getting from node n to the goal is never an overestimate. This is known as a **heuristic condition.**

Only if these three conditions are satisfied, A* is guaranteed to find an optimal (least) cost path. **Please note that A* algorithm is admissible for any node n if on such path, $h'(n)$ is always less than or equal to $h(n)$.** This is possible only when the evaluation function value never overestimates the distance of the node to the goal. Although the admissibility condition requires $h'(n)$ to be a lower bound on $h(n)$, **it is expected that the more closely $h'(n)$ approaches $h(n)$, the better is the performance of the algorithm.**

If $h(n) = h'(n)$ –an optimal solution path would be found without over expanding a node off the path. We assume that one optimal solution exists.

If $h'(n) = 0$ –A* reduces to blind uniform cost algorithm or breadth-first algorithm.

Please note that the admissible heuristics are by nature optimistic because they think that the cost of solving the problem is less than it actually is because $g(n)$ is the exact cost for

each n. Also note that $f(n)$ should never overestimate the true cost of a solution through n. For example, consider a network of roads and cities with roads connecting these cities. Our problem is to find a path between two cities such that the mileage/fuel cost is minimal. Then an admissible heuristic would be to use distance to estimate the costs from a given city to the goal city. Naturally, the air distance will be either equal to the real distance or will underestimate it i.e., $h'(n) \leq h(n)$.

2.2.3.6 Problem Reduction

OR graph: As we have already studied Best-first search tree. It's search graph can be explored to avoid duplicate paths. In such a **directed search graph**, each **node** represents a point in the problem space.

Each node also indicates how much promising it is, a parent link (pointing back to the best node form which it came) and the list of nodes generated (or explored) from it. **Please note that the parent link will make it possible to recover the path to the goal once the goal is found.** The list of successors (explored) will make it possible if a better path is found to an already existing node and also to propagate the improvement down to its successors.

This graph is known as an OR graph.

It is so called because each of its branches represents an alternative problem solving path. The techniques that we have studied so far, all are the example of OR graph only as in them we try to find a single path to the goal. **OR graph finds a single path.**

AND-OR graph (or tree)

We already know about the **divide-and-conquer strategy** i.e., **a solution to a problem can be obtained by decomposing it into smaller sub-problems. Each of this sub-problem can then be solved to get its sub-solution. These sub-solutions can then be recombined to get a solution as a whole. This is called as decomposition or reduction.** This method generates arcs which are called as **AND arcs. Please note that one AND arc may point to any number of successor nodes, all of which must be solved in order for an arc to point to a solution.** Here also several arcs may emerge from a simple node. Each will indicate the different ways in which the original problem might be solved. This is actually called as an AND-OR graph or AND/OR tree. For example.

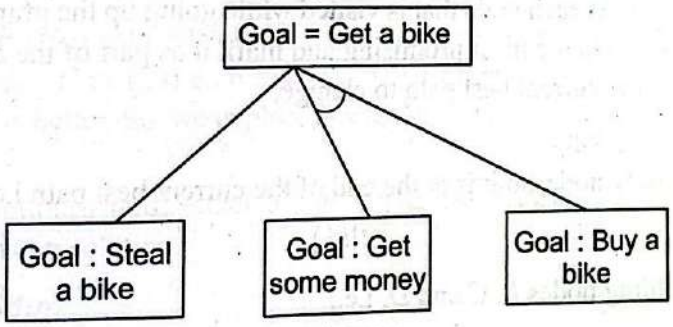


Fig. 2.22 AND/OR graph example.

Actually, best-first is not suitable for searching AND-OR graphs. Consider a small AND/OR tree shown in Fig. 2.23.

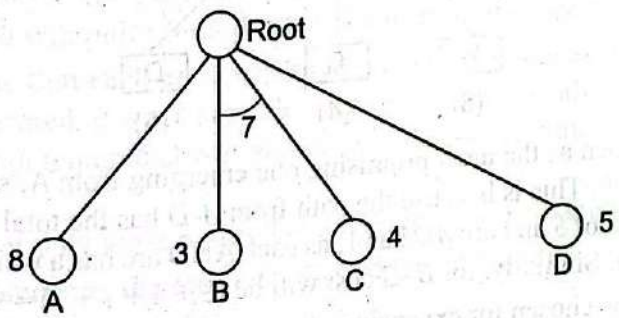


Fig. 2.23 Sample AND/OR tree.

Here, the numbers show the approximate value of the heuristic function ($f'(n)$) at that node. In Fig. 2.23, we observe that the minimal is B which has the $f'(n)$ value of 3. **But please note that B forms a part of the AND graph** and so we need to consider the other branch also of this AND tree i.e., C which has a weight of 4. So, our estimate now is (3 + 4) = 7. Now, this estimate is more costlier than that of branch D i.e. 5. So, we explore node-D instead of C as it has the lowest value. This process goes on till we search our goal node. **From this we observe that the choice of node to expand depends on it's $f'(n)$ value and also on whether that node is part of the current best path from the initial node.**

Algorithm to search AND-OR graph

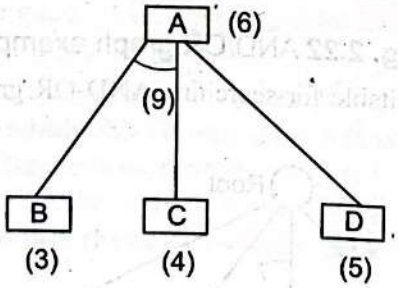
In order to develop an algorithm for searching an AND-OR graph, we define a term called as **Futility**. If the estimated cost of a solution becomes greater than the value of FUTILITY, then the search is abandoned. So, futility is like a **threshold value** i.e., any solution with a cost above it, is too expensive to be practical, even if it could ever be found. Let us write an algorithm of problem reduction now.

1. Initialize the graph to the starting node.
2. Loop until the starting node is labeled SOLVED or until its cost goes above FUTILITY:
   (*a*) Traverse the graph, starting at the initial node and following the current best path and accumulate the set of nodes that are on that path and have not yet been expanded or labeled as solved.
   (*b*) Pick one of these unexpanded nodes and expand it. If there are no successors, assign FUTILITY as the value of this node. Otherwise, add its successors to the graph and for each of them compute $f'(n)$. If $f'(n)$ of any node is O, mark that node as SOLVED.
   (*c*) Change the $f'(n)$ estimate of the newly expanded node to reflect the new information provided by its successors. Propagate this change backward through the graph. If any node contains a successor arc whose descendants are all solved, label the node itself as SOLVED. At each node that is visited while going up the graph, decide which of its successor arcs is the most promising and mark it as part of the current best path. This may cause the current best path to change.

Consider the following e.g.,

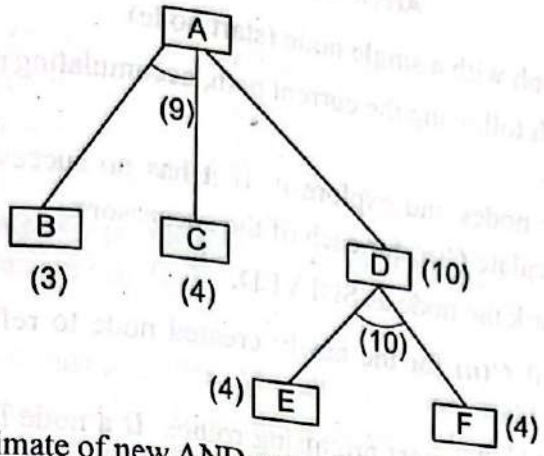
At step 1, *A* is the only node, so it is at the end of the current best path i.e., Before step 1:

It is expanded, yielding nodes *B*, *C* and *D*, i.e.,



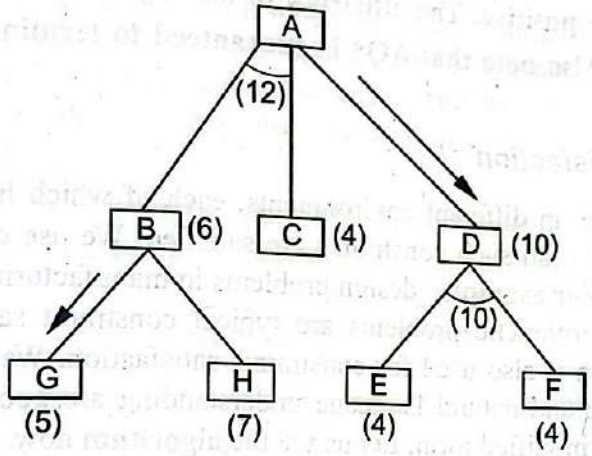
The arc to *D* is labeled as the most promising one emerging from A, since it costs 6 compared to B and C, which costs 9. This is because the path from *A-D* has the total cost of (*D* + the cost of arc *AD*). Now, *D* has cost of 5 and arc *AD* has 1; as each AND arc has a weight of one. So, total cost is 5 + 1 =6 for *A–D* path. Similarly, for *B–C* cost will be (3 + 4 + 1 + 1) = 9.

In step (2), node *D* is chosen for expansion.



∴ Combined cost estimate of new AND-arc to E and F is 4 + 4 + 1 + 1 = 10.

So, we update the $f'(n)$ value of D to 10. Going back one more level, we see that this makes the AND arc B –C better than arc to D, so it is labeled as the current best path by arrows. **At step 3,** we traverse that arc from A and discover the unexpanded nodes B and C. Let's choose to explore B first. This generates two new arcs, the ones to G and to H i.e.,



Propagating their $f'(n)$ values backward, we update $f'(n)$ of B to 6. This requires updating the cost of the AND arc $B - C$ to 12 (i.e., 6 + 4 + 2 =12). Comparing this with arc to D, which has the cost of 10, we find it is better. So, we explore D and thus either node–E or F is chosen for expansion at step-4.

This process continues until either a solution is found or all paths have led to dead ends, indicating that there is no solution.

2.2.3.7 *AO* algorithm*

Our real-life situations cannot be exactly decomposed into either AND tree or OR tree but is always a combination of both. So, we need an AO* algorithm where O stands for 'ordered'. Instead of two lists OPEN and CLOSED of A* algorithm, we use a single structure GRAPH in AO* algorithm. It represents a part of the search graph that has been explicitly generated so far. **Please note that each node in the graph will point both down to its immediate successors and to its immediate predecessors. Also note that each node will have some $h'(n)$ value associated with it**. But unlike A* search , $g(n)$ is not stored. It is not possible to compute a single value of $g(n)$ due to many paths to the same state. It is not required also as we are doing top-down traversing along the best-known path.

This guarantees that only those nodes that are on the best path are considered for expansion. Hence, **$h'(n)$ will only serve as the estimate of goodness of a node.**

Next, we develop an AO* algorithm.

Step-1: Create an initial graph with a single node (start node).

Step-2: Transverse the graph following the current path, accumulating nodes that have not yet been expanded or solved.

Step-3: Select any of these nodes and explore it. If it has no successors then call this value— FUTILITY else calculate $f'(n)$ for each of the successors.

Step-4: If $f'(n) = 0$; then mark the node as SOLVED.

Step-5: Change the value of $f'(n)$ for the newly created node to reflect its successors by back propagation.

Step-6: Wherever possible use the most promising routes. If a node is marked as SOLVED then mark the parent node as SOLVED.

Step-7: If the starting node is SOLVED or value is greater than FUTILITY then stop, else repeat from step-2.

Please note that AO* will always find a minimum cost solution if one exists, if $h'(n) < h(n)$ and that all arc costs are positive. The efficiency of this algorithm will depend on how closely $h'(n)$ approximates $h(n)$. **Also note that AO* is guaranteed to terminate even on graphs that have cycles.**

2.2.3.8 *Constraint Satisfaction*

In the real-world, we work in different environments, each of which has some constraints. Our solution should be such that all such constraints are satisfied. We use extensive domain specific and heuristic knowledge. **For example,** design problems in manufacturing, optimal tour problems and so on. Even crypta-arithmetic problems are typical constraint satisfaction problems. The term **relaxation algorithm** is also used for constraints satisfaction. Waltz algorithm for labeling line drawings in a picture and natural language understanding are good examples of constraint satisfaction algorithm in a modified form. Let us see the **algorithm** now.

1. Until a complete solution is found or until all path have led to dead ends, do:
    1. Select an unexpanded node of the search graph.
    2. Apply the constraint inference rules to the selected node to generate all possible new constraints.
    3. If the set of constraints contains a contradiction, then report that this path is a dead-end.
    4. If the set of constraints describes a complete solution, then report success.
    5. If neither a contradiction nor a complete solution has been found, then apply the problem space rules to generate new partial solutions that are consistent with the current set of constraints. Insert these partial solutions into the search graph.

To see how this algorithm works, consider the following problem.

Problem **(cryptarithematic problem)**

```
  SEND
+ MORE
------
 MONEY
```

Constraints

1. Values are to be assigned to letters from 0 to 9 only.
2. No two letters should have the same value.
3. If the same letter occurs more than once, it must be assigned the same digit each time.
4. The sum of the digits must be as shown in the problem.

Initial Problem State

S = ?	M = ?	C1 = ?
E = ?	O = ?	C2 = ?
N = ?	R = ?	C3 = ?
D = ?	E = ?	C4 = ?

Goal States: A goal state is a problem state in which all letters have been assigned a digit in such a way that all constraints are satisfied.

SOLUTION

The solution process proceeds in cycles. At each cycle, two vital things are done—

(*a*) Constraints are propagated by using rules which correspond to the properties of arithmetic. A value is guessed for some letter whose value is not yet determined, as this affects the effort involved in search.

(*b*) More constraints are added/generated due to propagating constraints.

At each cycle, there may be several choices to apply. A few useful heuristics can help to select the best rule to apply first. **For example**, if there is a letter that has only two possible values and another with six possible values then there is a better chance of guessing right on the first than on the second. This procedure can be implemented as dfs also so that a large number of intermediate positions in both the problem and the constraint spaces will not have to be stored. We can also store all constraints in one central database and also to record at each node the changes that must be undone during backtracking.

The results of first few cycles of processing of this crytarthimetic problem are shown in Fig. 2.24. Here, C1, C2, C3, C4 are the carry bits out of the columns, from right to left.

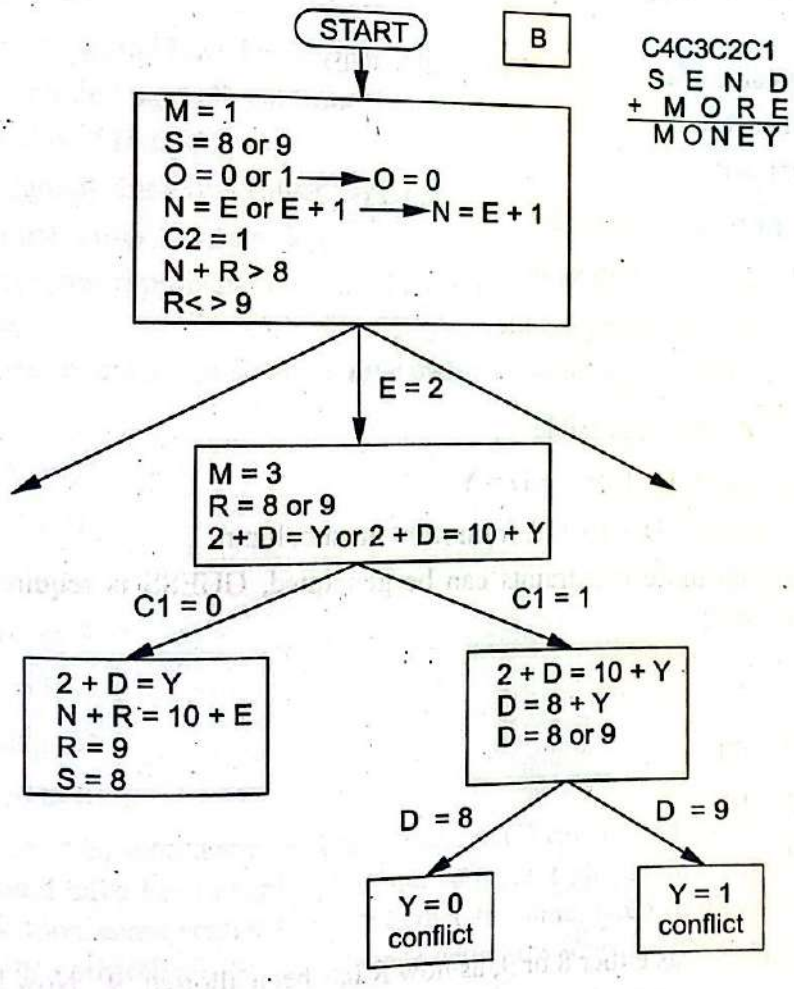


Fig. 2.24 Solution to crytarthimetic problem

Cycle 1

(*i*) M = 1 because two single digit numbers plus a carry cannot total more than 19.

(*ii*) S = 8 or 9 (minimum values) because S + M + C3 should be > 9 (to generate a carry) and M = 1, therefore S + 1 + C3 > 9 or S + C3 > 8, as C3 can be 0 or 1. So S > 8 or > 7 depending upon C3 = 0 or 1.

(*iii*) O = 0 because S + M + C3 ≤ 1.

O must be atleast 10 to generate a carry and it can be at most 11. That is O can be zero or 1. As M is already 1 and no two digit should have the same value, so O is zero (0).

(*iv*) N = E or E + 1 depending on the value of C2. But N cannot have the same value as E. So N = E + 1 and C2 = 1.

(*v*) In order for C2 to be 1, the sum of N + R + C1 must be greater tan 9, so N + R must be greater than 8.

(*vi*) N + R cannot be greater than 18, even with a carry in so, E cannot be 9.

E + O + C2 = N

When C2 = 0 and O = 0

E = N which is not possible.

At this point let's assume that no more additional constraints are generated so a GUESS is made.

Suppose value of E (which occurs the maximum number of times (3) and so interacts maximum with other letters).

Cycle 2 begins

Let E = 2 (the next value after 0 and 1, already allocated)

(*i*) So N = 3 because E + 1 = N

as R + N + C1 = E

R + 3 + C1 (0 or 1) = 2 or 12

R + 3 + 0 = 2, R = –1 not possible

R + 3 + 1 = 2, R = –2 not possible

R + 3 + 0 = 12, R = 9, possible

R + 3 + 1 = 12, R = 8, possible

(*ii*) D + E = Y when C1 = 0 or 2 + D = Y

and when C1 = 1 or 2 + D = 10 + Y, from right most column.

Again assuming no more constraints can be generated, GUESS is required and now C1 is chosen for GUESS. Let

C1 = 1

Cycle 3 begins

(*i*) D + E = Y + C1

2 + D = Y + 10

3 + R = 12

R = 9

(*ii*) S = 8 because S was either 8 or 9, as now R has been allotted '9'. Now let us assign value to C1, 0 or 1.

First let us assign C1 = 10.

$D + E = Y + 10$ (because C1 = 1)

$D + E = 10 + Y$

$2 + D = 10 + Y$ and for this to happen

Either $D = 8$

or $D = 9$

So C1 cannot be 1.

Now consider C1 = 0.

then $8 + D = Y$, Y can be 0 or 1

$D = -8$ (when $Y = 0$)

$D = -7$ (when $Y = 0$)

So C1 cannot be 0 either.

Thus D has four values, two negative and two positive.

When $D = 8$

$Y = 10$ i.e., $= 0$

and when $D = 9$

$Y = 11$ i.e., 1

So $Y = 0$ and $Y = 1$

lead to conflict.

This contradiction could have been explained through another type of arguments:

Earlier S = 8 or 9 and again R = 8 from II loop and when C1 = 1, D = 8 or 9

And now D = 8 or 9 from C1 = 1.

Three letters cannot share two values, so $C1 \neq 1$.

Comments on Constraints Propagation

The rules of the constraint propagation should not infer spurious constraints. Nor do they need to infer all legal rules

For example, we could adopt an alternative path which gives C1 = 0.

Let $C1 = 1$

Then $E + D = Y + C1$

or $E + D = Y + 10$

$2 + D = Y + 10$

D would be either 8 or 9

8 or 9 is assigned to R

is assigned to D

is assigned to S

Two values cannot be shared among three letters, so C1 cannot be 1. If this were realized earlier fruitless search could have been avoided. But the constraint propagation rules used were not that sophisticated so it took some search steps. Whether the time taken by the actual search route is more or less than the constrained method depends upon how long it takes to perform the reasoning required for constraint propagation.

We are in a position to solve a problem now.

EXAMPLE 1. Trace the constraint satisfaction procedure solving the following cryptarithmatic program:

```
  C R O S S
+ R O A D S
-----------
D A N G E R
```

SOLUTION 1.

```
  C R O S S
+ R O A D S
-----------
D A N G E R
```

(*i*) D = 1 since it is generated because of carry which at most can be 1($\because 9 + 9 = 18$).

(*ii*) C + R = 10 or 12 to 18 depending up carry C4.

(*iii*) Since, we have no path to move on, we guess the values for S.

(*a*) C + R = A

+ 4 =

(*b*) R + O = N

4 + =

(*c*) O + A = G

(*d*) S + D = E

2 + 1 = 3

(*e*) S + S = R

2 + 2 = 4

Thus we get: A C D E G N O R S

↓ ↓ ↓ ↓

1 3 4 2(guess)

Now, lets say 'O' = 5 (any thing other than 1, 2, 3, 4)

if O = 0 then R + 0 = N

$\Rightarrow$ 4 + 0 = 4 (Not possible)

$\therefore$ guess O = 5

(*a*) C + R = A

+ 4 =

(*b*) R + O = N

4 + 5 = 9

(*c*) O + A = G

5 + =

(*d*) S + D = E

2 + 1 = 3

(*e*) S + S = R

2 + 2 = 4

$\therefore$ A C D E G N O R S

↓ ↓ ↓ ↓ ↓ ↓

1 3 9 5 4 2

(guess)

Here, C + R should be 10, 12, 13, 14, 16, 16 or 17 as $C_4 \neq 1$

$\therefore$ for C we can have,

for C = 6 (for total of 10) we have A = 0

and A + 0 = G i.e. 5 + 0 = 5 i.e. G = 5

$\Rightarrow$ O = G = 5 (not possible)

$\Rightarrow$ For O = 6, 7, 8 too we are not able to get values for C s.t. C + R > 10 (no value in this case can be (=O) as it will result in same value for 2 different characters) and for O = 6 we get repeating values.

$\therefore$ Our guess for S = 2 is wrong.

(*iv*) For S = 3,

(*a*) C + R = A
9 + 6 = 15

(*b*) R + O = N
6 + 2 = 8

(*c*) O + A = G
2 + 5 = 7

(*d*) S + D = E
3 + 1 = 4

(*e*) S + S = R
3 + 3 = 6

$\therefore$ A C D E G N O R S
↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓ ↓
5 9 1 4 7 8 2 6 3
(guess) (guess)

This solution has no repetitions thus we get—

C R O S S		9 6 2 3 3
+ R O A D S	→	+ 6 2 5 1 3
D A N G E R		1 5 8 7 4 6

2.2.3.9 *Means-End Analysis (MEA)*

Means-end-analysis is a special type of knowledge-rich search that allows both backward and forward searching.

Principle: "It (MEA) allows us to solve the major parts of a problem first and then go back and solve the smaller problems that arise while assembling the final solution".

Technique: It is based on the use of the operators which transform the state of the world. **The difference between the current state and the goal state is detected.**

Once such a difference is isolated, an operator that can reduce the difference has to be found. But sometimes it is not possible to apply this operator to the current state. So, we try to get a sub-problem out of it and try to apply our operator to this new state. If this also does not produce the desired goal state then we try to get second sub problem and apply this operator again. This process may be continued. **But please note here that if the difference is chosen correctly and if the operator is really effective at reducing the difference then the two sub problems are easier to solve than the original problem.**

Why is it called as Means-End-Analysis?

Given a description of the desired state of the world (**the end**) it works backwards, selecting and using operators that will achieve it (**the means**). Hence, the name, Means-End-Analysis (MEA).

MEA works incrementally

The above technique of MEA works incrementally and not as a single step. This is so because the operators are applied recursively till a goal state is achieved.

MEA works on three primary goals.

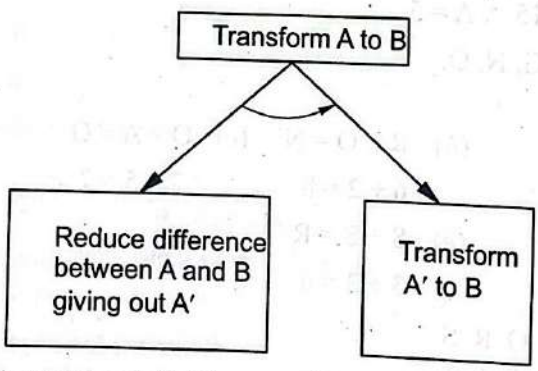
They are as follows—

(*a*) Transformation.

(*b*) Reduction

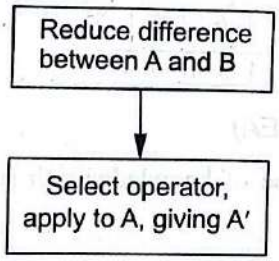
(*c*) Application

Transformation means to transform object A into object B i.e.,



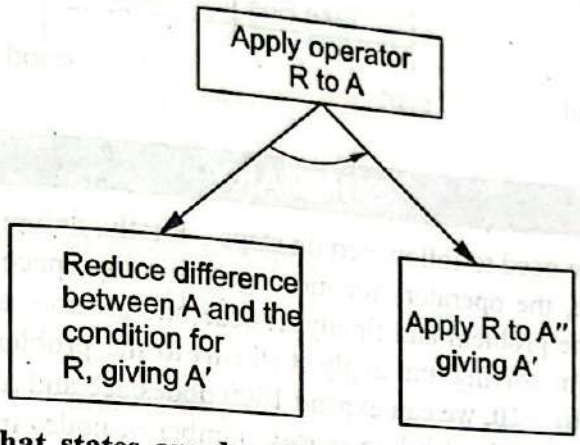
This is an AND graph which subdivides a problem into an intermediate problem and then transforms that problem into the goal state B. **Please note that this process terminates when there is no difference between A and B or we can say when the goal is reached.**

Reduction means to reduce the difference between object-A and object-B by modifying object-A i.e.,



Please note that the goal of this operation is to reduce the difference between the object A and object-B by transforming object A into object-A', nearer the goal B, by means of a relevant operator (say, R).

Application means to apply the operator R to object-A. This will again be a AND graph showing the goal of reducing the difference between object-A and the pre-conditions required for the operator R, giving intermediate object A". Operator R is then applied to A", transforming it to A', which is close to the goal B. That is,



Please understand that states are described in some structured way like by predicate calculus and moves are performed by the operators. Each operator has a—

(*a*) **Pre-condition:** It puts some constraints on the states it uses.

(*b*) **Post-condition:** It says which conditions will be true when the operator has been used.

For examples —

(*a*) GPS was the first AI program to exploit MEA.

(*b*) STRIPS (A Robot Planner) is an advanced problem solver that incorporates MEA.

Algorithm for Means-End-Analysis (MEA)

We given an algorithm for MEA now.

Step-1: Until the goal is reached or no more procedures are available.

Step-2: Describe the current state, the goal state and the difference between the two.

Step-3: Use the difference between the current state and goal state, possibly with the description of the current state or goal state, to select a promising procedure.

Step-4: Use the promising procedure and update the current state.

Step-5: If goal is reached then success else it is a failure.

We are in a position to solve an example now.

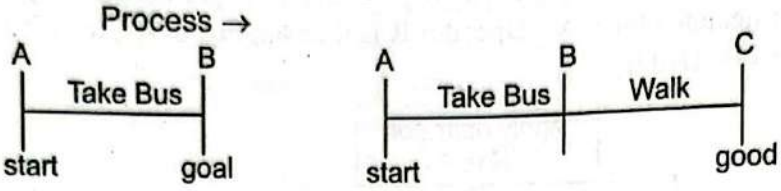
EXAMPLE 1. Show how Mean-End-Analysis (MEA) can be used to solve the problem of getting from one place to another. Assume that the available operators are walk, drive, take the bus, take a cab and fly.

SOLUTION 1. Means-End Analysis—

Available Operators	Preconditions	Results
1. Walk		at (man, loc)
2. Drive	have (man, car) ^ (on obj, bus)	at (man, loc)^ at (obj, loc)
3. Take bus	On (man, bus)^ on (obj, bus)	at (man, loc)^ at (obj, loc)
4. Take cab	hire (man, cab) ^ on (obj, cab)	at (obj, loc)
5. Fly	book (man, airbus)^ on (obj, airbus)	at (man, loc) ^ at (obj, loc)

∴ **Difference Table** is as follows

	Walk	Drive	Take bus	Teke cab	Fly
move man	*	*	*		*
move obJ		*	*	*	*



SUMMARY

To solve an AI problem we need to follow certain steps—Firstly, define the problem precisely i.e., specify the problem space, the operators for moving within the space and the starting and goal state. Secondly, analyze the problem and finally, choose one or more techniques for representing knowledge and for problem solving and apply it (them) to the problem. In BFS, if we assume that the branching factor (b) = 10, we can expand 1000 nodes/sec and also assume 100 bytes/node storage then the observe that as the depth increases, number of nodes increases and both time and memory requirement also increases. One such snapshot is given below—

Depth	Nodes	Time	Memory
0	1	1 ms	100 bytes
2	111	0.1 sec	11 KB
4	11, 111	11 secs	1 MB
6	10^6	18 min	111 MB
8	10^8	31 hrs	11 GB
10	10^{10}	128 days	1 TB
12	10^{12}	35 yrs	111 TB
14	10^{14}	3500 yrs	11, 111 TB

This table shows how memory requirements increases as the depth increases.

MULTIPLE CHOICE QUESTIONS [MCQS]

1. States are operated upon by a set of

(*a*) Operators (*b*) Components (*c*) State space (*d*) None of the above.

2. Nodes are states and arcs represent

(*a*) Directions (*b*) Actions (*c*) Functions (*d*) None of the above.

3. The process of creating a formal description of a problem using the knowledge about the given problem, so as to create a program for solving a problem is called as

(*a*) Standardization (*b*) Operationalization

(*c*) Generalization (*d*) None of the above.

4. A set of all possible states of a given problem is known as

(*a*) State (*b*) Space search (*c*) State space (*d*) None of the above.

5. The process of decomposing up of a complex problem into a set of primitive sub problems, finding solutions to them and then integrating them to get a complete solution as a whole is known as

(a) Expansion (b) Problem reduction
(c) Production systems (d) None of the above.

6. Production systems were proposed by
(a) Foreign bank (b) Codd (c) Emit post (d) None of the above.

7. The maximum length of operator-application-sequence is known as
(a) Depth cut-off (b) Depth of tree
(c) Search path (d) None of the above.

8. Uniformed search is also known as a
(a) Heuristic search (b) Blind search
(c) Goal search (d) Beam search.

9. The time complexity of the bfs algorithm
(a) 0 (bd) (b) 0 (b^d) (c) 0 (d^b) (d) 0 (b + d)

10. Blind alley means
(a) Applying two or more searches (b) Search that can go on and on
(c) Blind search only (d) None of the above.

11. A heuristic is a
(a) Rule of thumb (b) Mathematics formulae
(c) Equation (d) None of the above.

12. The most common problem with all AI solution is
(a) Polynomial explosion (b) Combinational explosion
(c) Nuclear explosion (d) None of the above.

13. Which module creates the possible solutions in Generate-And-Test-Algorithm
(a) OS module (b) Kernel module
(c) Generator module (d) None of the above.

14. An area of the search space which is higher than the surrounding areas is a
(a) Ridge (b) Plateau (c) Hill (d) None of the above.

15. Which of the following is a greedy search
(a) BFS (b) DFS
(c) Best-First Search (d) None of the above.

16. Goal nodes have an evaluation function value of
(a) Zero (b) One (c) Two (d) Three

17. The sum of the evaluation function value and the cost along the path leading to the state is called as
(a) Fitness number (b) Whole number
(c) Complex number (d) None of the above.

18. Constraint satisfaction is also referred to as
(a) Divide-and-conquer algorithm (b) Greedy algorithm
(c) Relaxation algorithm (d) None of the above.

19. Mean-End-Analysis (MEA) allows
   (*a*) Backward searching (*b*) Forward searching
   (*c*) Both 'a' and 'b' (*d*) None of the above.

20. MEA works on three primary goals which are
   (*a*) Transformation, scaling and rotation
   (*b*) Transformation, reduction and application
   (*c*) Scaling, reduction and application
   (*d*) None of the above.

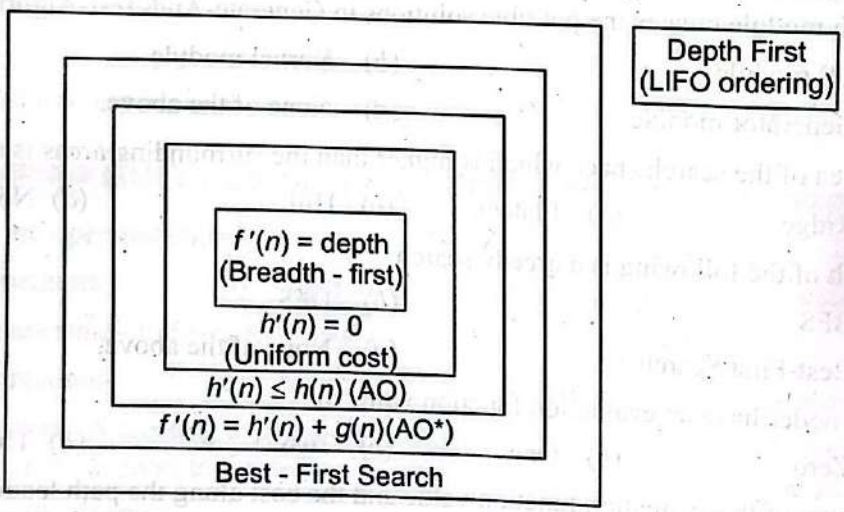
ANSWERS

1. (*a*)	2. (*b*)	3. (*b*)	4. (*c*)	5. (*b*)
6. (c)	7. (*a*)	8. (*b*)	9. (*b*)	10. (*b*)
11. (*a*)	12. (*b*)	13. (*c*)	14. (*a*)	15. (*c*)
16. (*a*)	17. (*a*)	18. (*c*)	19. (*c*)	20. (*b*)

CONCEPTUAL SHORT QUESTIONS WITH ANSWERS

Q. 1. What is the relationship between various search algorithms-Breadth-first, uniform cost, A0*, A* and Best-First search?

Ans. As shown in Figure, when $h'(n) = 0$ for all nodes, we have uniform-cost algorithm. When $f'(n) = g(n) = \text{depth}\ (n)$, we have breadth-first search which spreads out along contours of equal depth. Please note from above diagram that both the uniform-cost and breadth-first algorithms are special cases of A* (with $h'(n) = 0$), so they are both admissible.

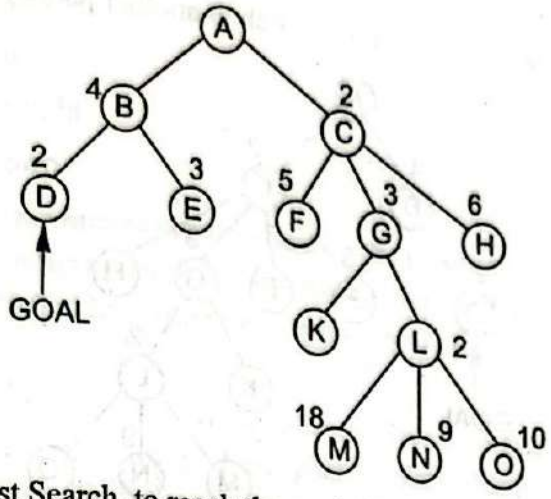


Q2. What is Search Control Knowledge?

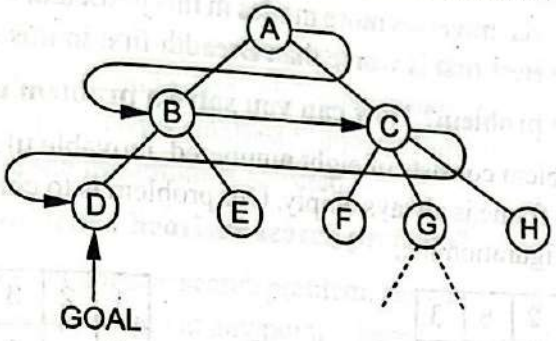
Ans. A large KB contains thousands of rules when there are many possible paths for reasoning, it is critical that fruitless ones are not pursued. Knowledge about which paths are most likely to lead quickly to a goal state is called as a **search control knowledge.**

Q3. When would best-first search be worse than simple breadth-first search?

Ans. CASE WHEN BEST-FIRST SEARCH IS WORSE THAN BREADTH-FIRST SEARCH
Consider a graph—



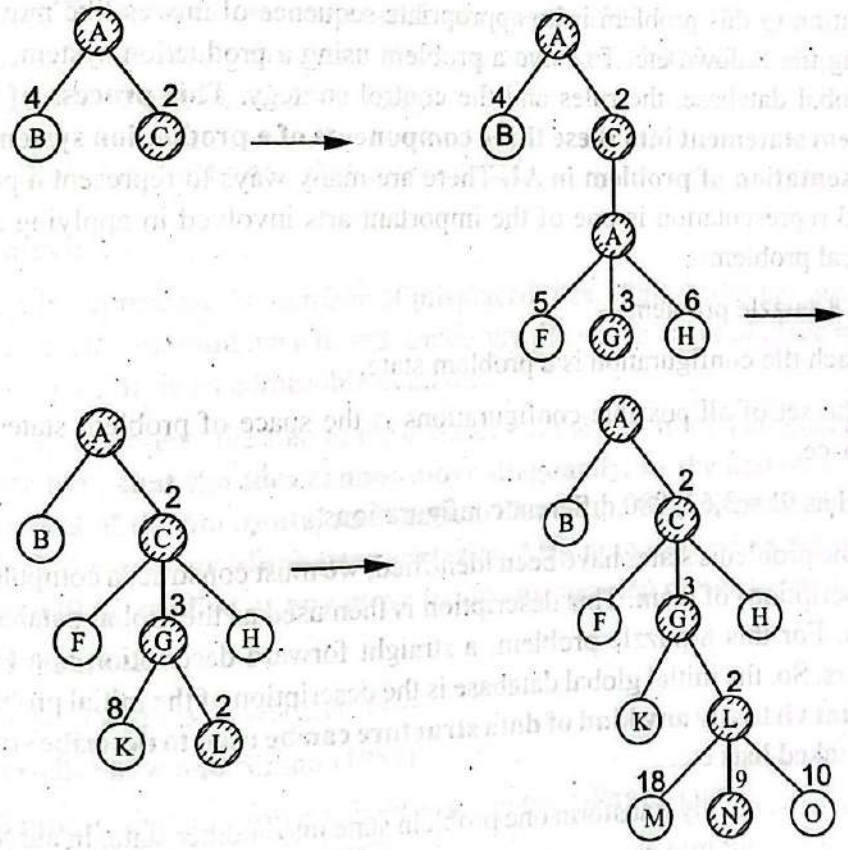
Using Breadth First Search, to reach the goal 'D' we would transverse as—



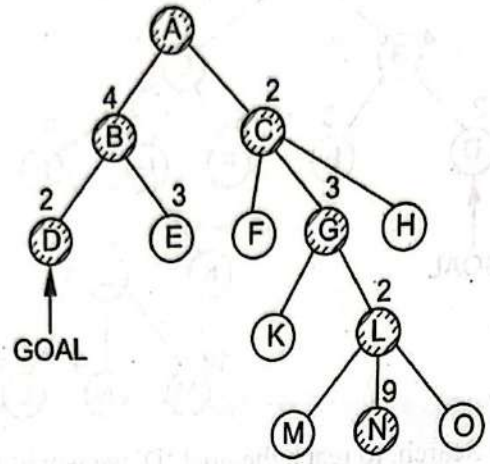
Path $\Rightarrow A \rightarrow B \rightarrow C \rightarrow D$

No. of modes traversed = 4.

Using Best First Search, to reach the goal 'D' we would traverse as—



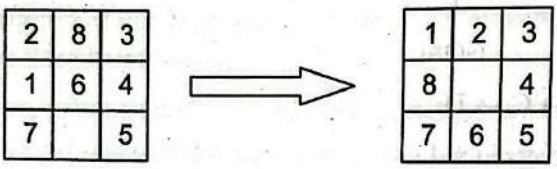
which is not the goal, hence we now switch to another path-B



Thus, Best first search traverses more modes in this particular example than the breadth-first search, and hence Best first is worse than Breadth first in this example.

Q4. What is 8-puzzle problem? How can you solve a problem using production system?

Ans. The 8-puzzle problem consists of eight numbered, movable tiles set in a (3 × 3) frame. One cell of this (3 × 3) frame is always empty. Our problem is to change the initial configuration into the goal configuration i.e.,



(Initial State) (Goal State)

A solution to this problem is an appropriate sequence of moves like moving tile 6 down, moving tile 8 down etc. To solve a problem using a production system, we must specify the global database, the rules and the control strategy. **This process of transforming a problem statement into these three components of a production system is called as the representation of problem in AI.** There are many ways to represent a problem selecting a good representation is one of the important arts involved in applying AI techniques to practical problems.

In our 8-puzzle problem—

(*a*) Each tile configuration is a problem state.

(*b*) The set of all possible configurations is the space of problem states or the problem space.

(*c*) It has 9! = 3,62, 880 different configurations.

Once the problems states have been identified, we must construct a computer representation (or description) of them. This description is then used as the global database of production system. For this 8-puzzle problem, a straight forward description is a (3 × 3) matrix of numbers. So, the initial global database is the description of the initial problem state. **Please note that virtually any kind of data structure can be used to describe states,** like vectors, trees, linked lists etc.

A **move** will actually transform one problem state into another state. In our 8-puzzle problem there can be four moves—

(*a*) Move empty space (blank) to the left

(*b*) Move blank up

(*c*) Move blank to right

(*d*) Move blank down.

Please note that these moves are modeled by the production rules which operate on the state descriptions in the appropriate manner. Also note that the rules have preconditions which must be satisfied by a state description in order for them to be applicable to that state description.

We have to reach our goal state. We can also specify some True/False condition on states to serve as a goal condition. It is the problem-goal condition that forms a basis for the termination condition of the production system. The control strategy repeatedly applies rules to state descriptions until a description of a goal state is produced. It also keeps track of the rules which have been applied so that it can compose them into the sequence representing the problem solution. We may add some constraints also to our solution i.e., to find an optimal (minimum) path solution (with smallest number of moves). For this, costs are associated with each move and the minimum is selected.

Q. 5. Is our 8-puzzle problem a heuristic search problem?

Ans. The 8-tile problem is a heuristic search problem. Herein, we have (8×4) possible tile moves. Most of these moves are illegal at any point. These illegal moves are visible to humans but not to our computer. A computer will simply swap it with a neighbouring tile. So, we use heuristics. The average solution cost for a randomly generated 8-tile puzzle is about 22 steps. The branching factor is about 3. This is because when the empty tile is in **middle**, there are **four** possible moves, when it is in a **corner** there are **two** and when it is on an **edge**, there are **three. Thus, an exhaustive search to a depth of 22 would have about** $33^{22} \approx 3.1 \times 10^{10}$ states. We can bring this down also by a factor of 1,70,000 if we keep track of repeated states. This is so because there are only—

$\frac{9!}{2}$ = 1,81,440 distinct states that are reacheable.

Heuristics used

1. Say, h_1 represents the number of misplaced tiles. If all 8-tiles are out of position then the start state would have $h_1 = 8$. Since any tile which is out of place must be moved at least once, h_1 is an admissible heuristic.
2. Say, h_2 represents the sum of the distances of the tiles from their goal positions. **Please note here that the tiles cannot move diagonally, so the distance we will count is the sum of the horizontal and vertical distances. This distance is called as the city block distance or Manhattan distance. Also note that when we move one tile, h_2 can still be applied as any move brings us one step closer to the goal.**

Q. 6. What is heuristics?

Ans. According to various AI researchers—

1. Newell, Shaw and Simon (1957)

"A process that may solve a given problem but offers no guarantees of doing so is called as a heuristic for that problem".

2. Marvin Minisky (1961)

"A heuristic is any method or trick used to improve the efficiency of a problem-solving program."

3. Feigenbaum and Feldman (1963):

"A state heuristic is rule of thumb, strategy, trick, simplification or any other kind of device which drastically limits search for solutions in large problem spaces. Heuristics do not guarantee optimal solution; in fact, they do not guarantee any solution at all. It can be said for a useful heuristic is that it offers solutions which are good enough most of the time."

Judea Pearl (1984):

"It is the nature of good heuristics both that they provide a simple means of indicating which among several course of action is to be preferred and that they are not necessarily guranteed to identify the most effective course of action, but do so sufficiently often. It is often said that heuristic methods are unpredictable, they work wonders most of the time but may fail miserably some of the time".

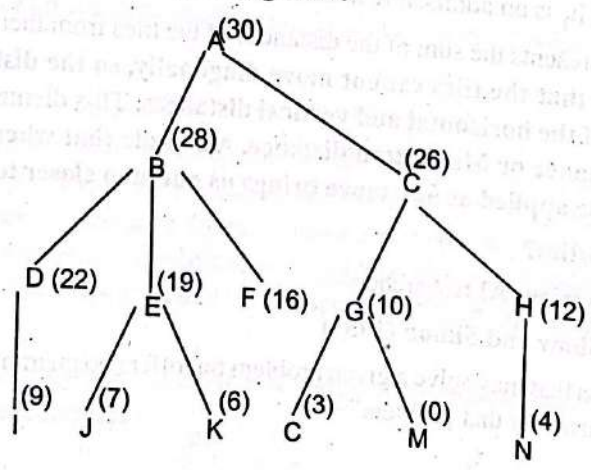
Q. 7. Distinguish between Heuristic algorithms and solution guaranteed algorithms?

[Cochin Univ., B. Tech (CSE) 7th Sem. Oct. 2000]

Ans. Let us tabulate the difference between the two

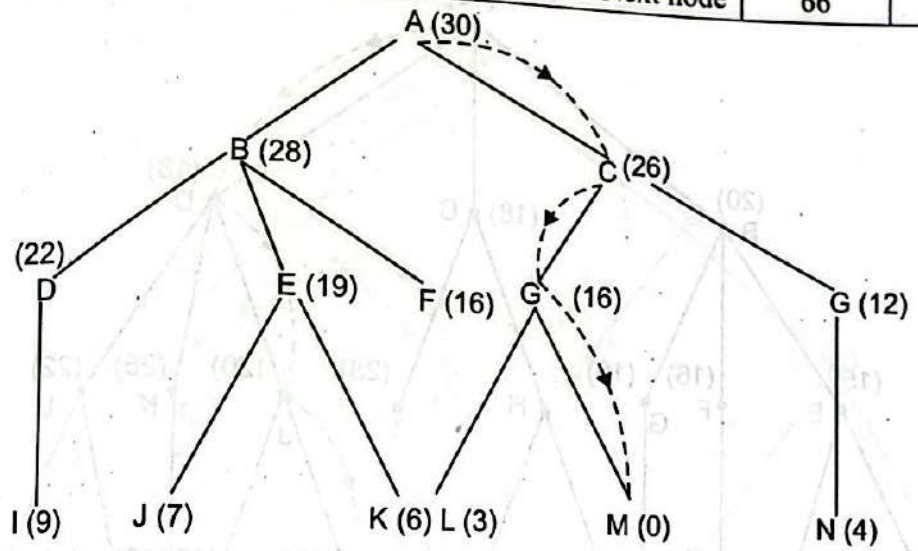
Heuristic Algorithms	Solution guaranteed algorithm
1. These algorithms are **non-exact algorithms** that do not guarantee that the best will be found. So, they are considered to be approximate algorithms.	1. **These algorithms are also called as exact-algorithms** as they guarantee to find an optimal solution.
2. They find solutions close to the best one.	2. They guarantee to find optimal solutions.
3. They find fast solutions.	3. They take exponential time to do so.
4. It is called as heuristic because it must prove that the best solution is the best.	4. It always finds an optimal solution.
For e.g., Local search methods, simulated annealing etc.	For e.g.: Branch and Bound, dynamic programming, Branch and cut etc.

Q. 8. Using the search tree given below, list the elements of the queue just before that next node is expanded. Use best first search where the number correspond to estimated cost-to-goal for each corresponding node.

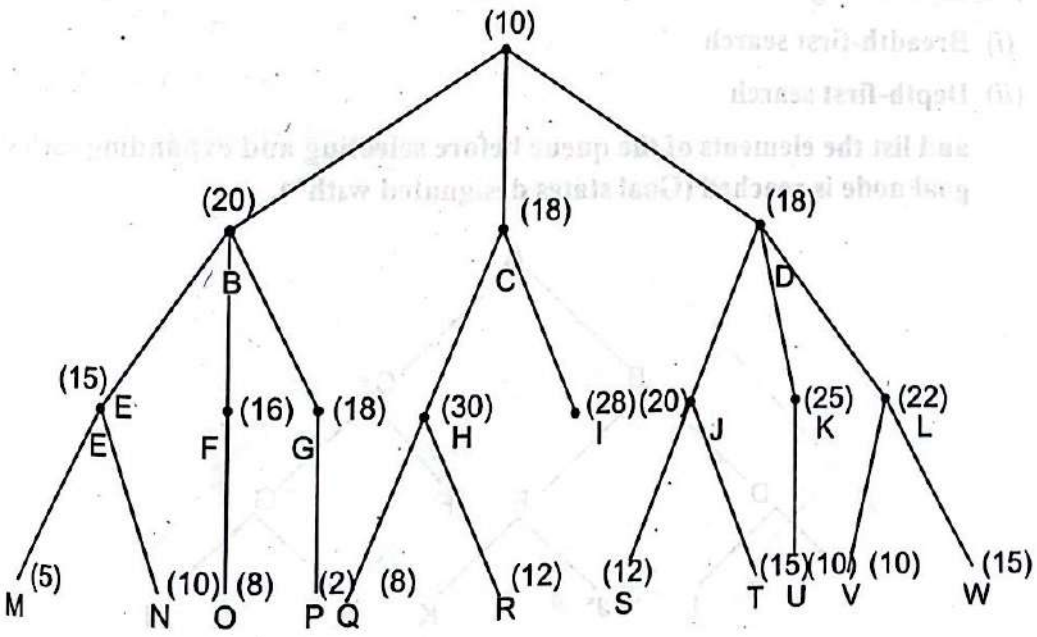


Ans. The best-first-search's cost is calculated in the following manner.

From node	To node	Cost of node	Possible Queue	Cost so far	Final best First queue
A	B	30	A-B	30	✓
	C	30	A-C	30	✓
B	D	28	A-B-D	58	×
	E	28	A-B-E	58	×
	F	28	A-B-F	58	×
C	G	26	A-C-G	56	✓
	H	26	A-C-H	56	✓
G	L	10	A-C-G-L	66	✓
	M	10	A-C-G-M	66	✓
H	N	12	A-C-H-N	68	×
L	Next node	3	A-C-G-L-Next Node	69	×
M	Next node	0	A-C-G-H-Next node	66	✓



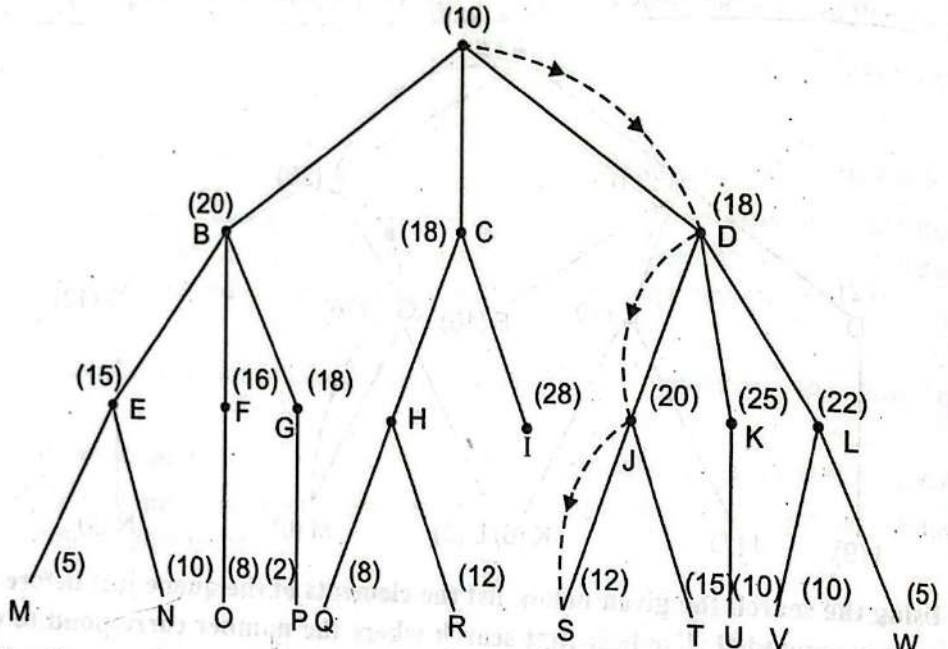
Q. 9. Using the search the given below, list the elements of the queue just before that next node is expanded. Use best-first search where the number correspond to estimated cost-to-goal for each corresponding node.



Ans. The table showing best-first search cost is as follows.

From node	To node	Cost of node	Possible queue	Cost so far	Final best First queue
A	B	10	A-B	30	×
	C	10	A-C	28	✓
	D	10	A-D	28	✓
D	J	20	A-B-J	48	×
	K	20	A-D-K	53	×
	L	20	A-D-L	50	×
C	H	18	A-C-H	58	×
	I	18	A-C-I	56	×
D	S	20	A-D-J-S	60	✓
	T	20	A-D-J-T	63	×

The best first search has path as A-D-J-S-Next node with cost 60.

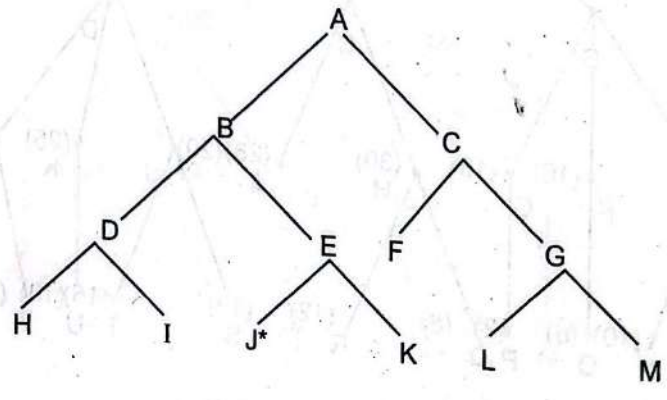


Q. 10. For the search given below, use

(*i*) Breadth-first search

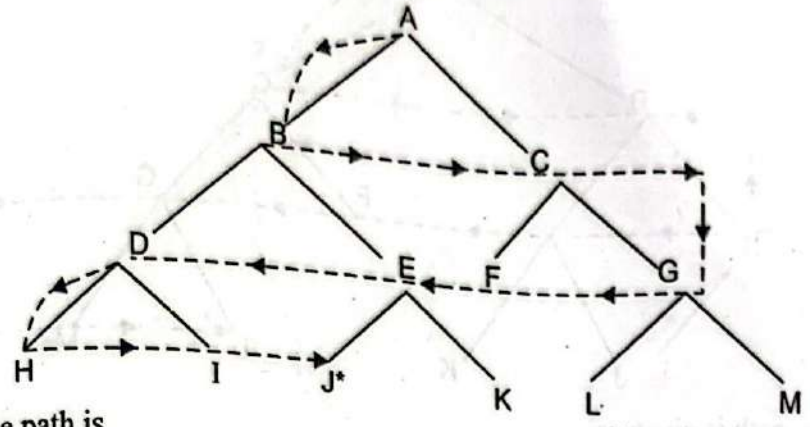
(*ii*) Depth-first search

and list the elements of the queue before selecting and expanding each state until goal node is reached (Goal states designated with*).



Ans. **(i) Breadth first search:**

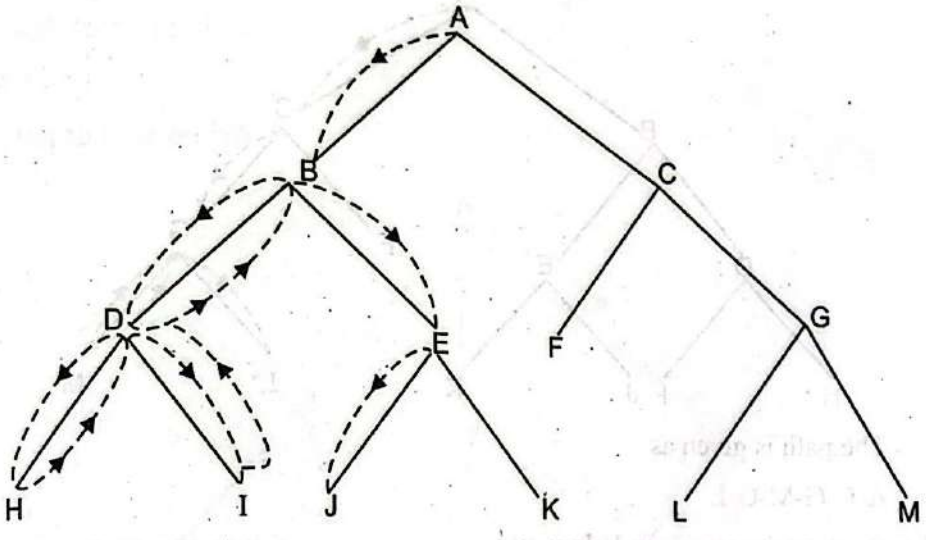
The given goal is J.



The path is

A-B-C-G-F-E-D-H-I-J.

(ii) Depth-first-search:



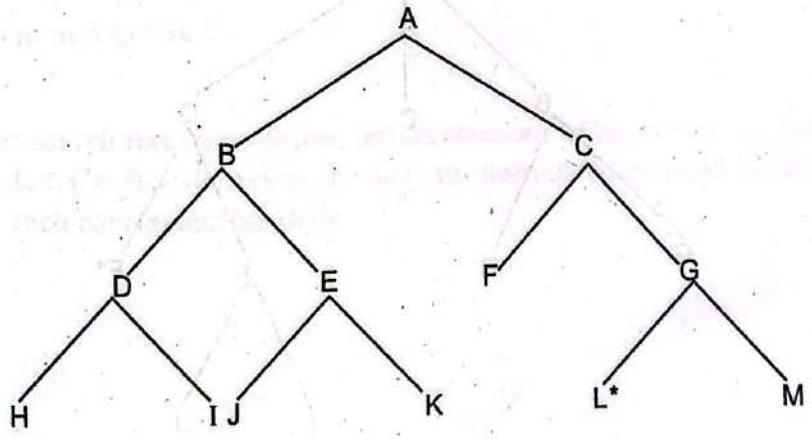
A-B-D-H-I-D-B-E-J.

Q. 11. For the search tree given below, use

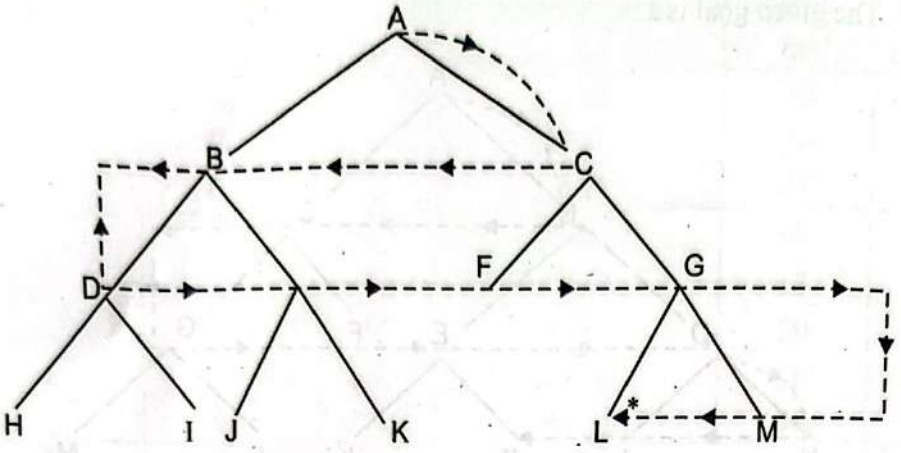
(i) Breadth first search.

(ii) Depth-first search.

Goal is designed by



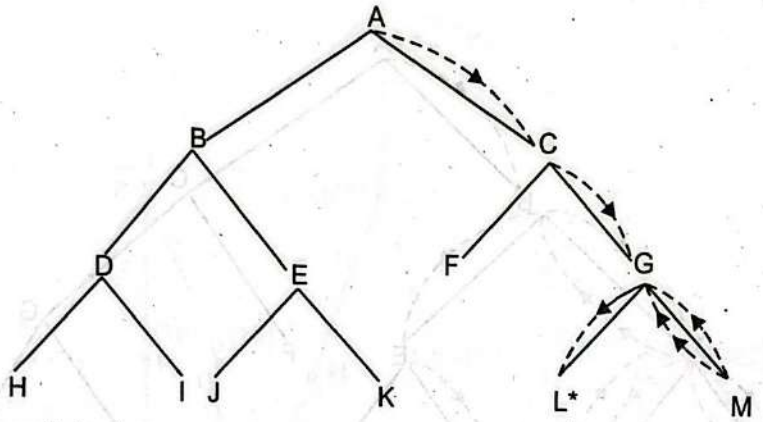
Ans. **(*i*) Breadth first search:**



The path is given as

A-C-G-M-G-L

(*ii*) Depth–first search:



The path is given as

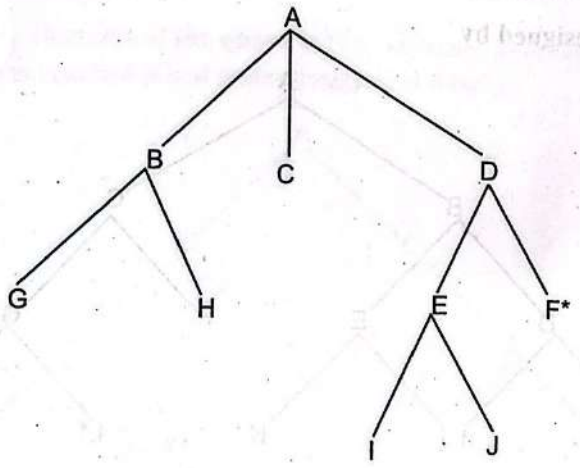
A-C-G-M-G-L

Q. 12. For the search tree given below, use

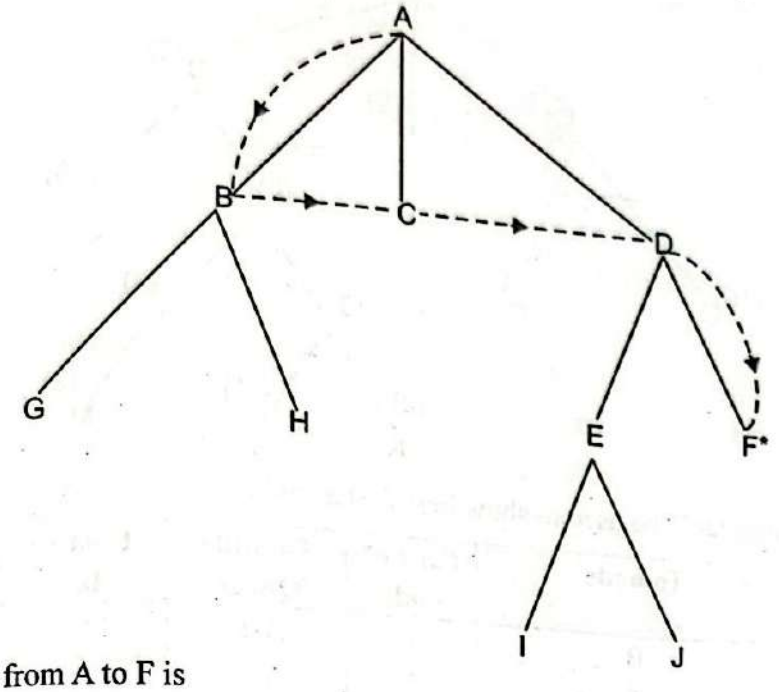
(*i*) Breadth-first search

(*ii*) Depth first search

Goal is shown as



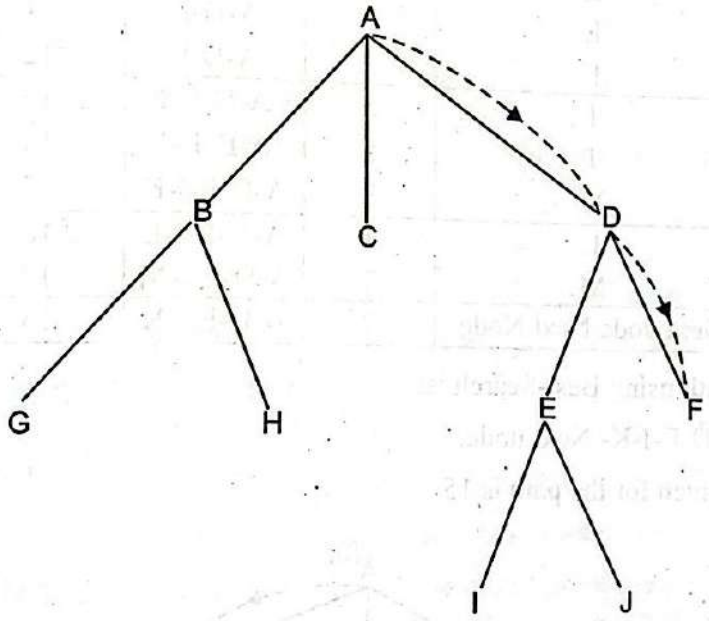
Ans. **(i) Breadth first search**



The path from A to F is

A-B-C-D-F

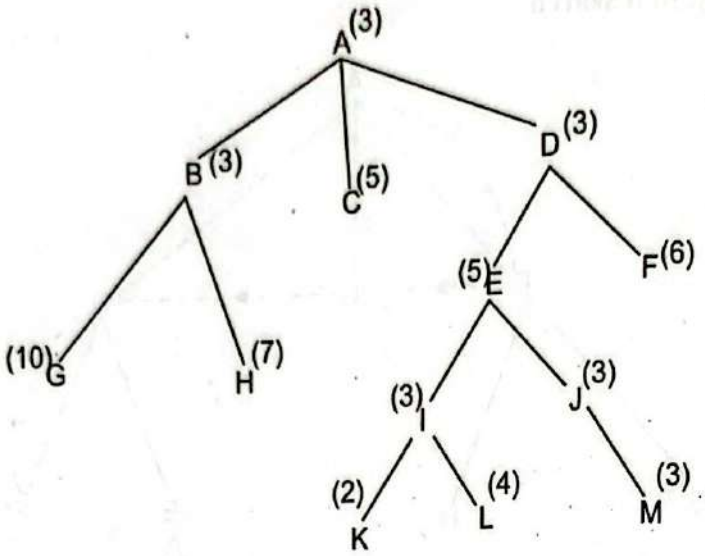
(ii) Depth first search



The path from A to F is

A-D-F

Q. 13. Using the search tree given below, list the elements of the queue just before next node is expanded. Use Best-first-Search where the number correspond to estimated cost-to-goal for each corresponding node.



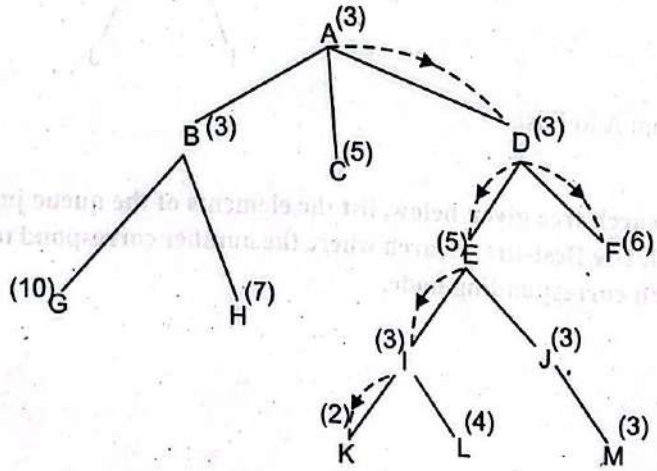
Ans. The following table be used to show best first search.

From node	To node	Cost of node	Possible Queue	Cost so far	Final best First queue
A	B	3	A-B	6	✓
	C	5	A-C	8	×
	D	3	A-D	6	✓
B	G	3	A-B-G	16	×
	H	3	A-B-H	13	×
D	E	3	A-D-E	11	✓
	F	3	A-D-F	12	×
E	I	5	A-D-E-I	13	✓
	J	5	A-D-E-J	14	✓
I	K	3	A-D-E-I-K	15	✓
	L	3	A-D-E-I-L	18	×
J	M	3	A-D-E-J-M	17	×
K	Next node Next Node	2	A-D-E-I-K	15	✓

The given path using Best-Search is:

A-D-E-I-K- Next node.

The cost is given for the path is 15



EXERCISE QUESTIONS

Q. 1. Given an example of a real world problem which can be effectively solved by "Hill climbing Algorithm. Explain the terms plateau and ridge.

[GGSIPU, B. Tech (CSE)-8th Sem., May 2008]

Q. 2. (*a*) Explain means end analysis with an example.

(*b*) What do you understand by admissible heuristic? Develop an admissible heuristic for any problem of your own. **[GGSIPU, B. Tech (CSE) 8th Sem., May 2009]**

Q. 3. (*a*) Compare uniformed search with informed search.

(*b*) Discuss problem reduction strategy of a problem with an example.

[GGSIPU, B. Tech (CSE)-8th Sem., May 2009]

Q. 4. (*a*) What are the problems with hill climbing and how can they be solved?

(*b*) Explain Means End Analysis Search technique with a suitable example.

(*c*) State the best first search algorithm.

(*d*) Define the following terms in reference to Steepest Ascent Hill Climbing:

(*i*) Local Maximum

(*ii*) Plateau.

(*iii*) Ridge. **[RTM, Nagpur University B.E., (IT) 7th Sem. Summer 2009]**

Q. 5. (*a*) Write the difference between :

(*i*) OR graph and AND-OR graph

(*ii*) Generate and test and hill climbing.

(*b*) How state space search work on state space of an AI problem? Explain it with the help of water jug problem.

(*c*) Explain advantages and disadvantages of dfs and bfs.

(*d*) What are various issues to be considered in the design of search problems/programs.

[RTM, Nagpur University, B.E. (C.T.)-7th Sem., Winter 2009]

Q. 6. Using a suitable example, illustrate steps of A* search. Why is A* search better than Best First Search. **[DTU (DCE)-ME (CS)-2005]**

Q. 7. Explain BFS algorithm. **[Cochin University, B.Tech (CSE)-8th Sem. Oct. 2004]**

Q. 8. (*a*) Why Hill-climbing is named so? Illustrate using an example.

(*b*) Discuss and compare Blind Search with Heuristic based search. Which one is better? Discuss using an example. **[GGSIPU, B.Tech (CSE)-8th Sem. May 2011]**