UNIT - II

Problem solving and Game playing

Syllabus:

Problem solving: state-space search and control strategies: Introduction, general problem solving, characteristics of problem, exhaustive searches, heuristic search techniques, iterative-deepening a*, problem reduction, constraint satisfaction.

Game playing: Introduction, game playing, alpha-beta pruning, two-player perfect information games.

Outcomes:

Student will be able to:

General Problem Solving

- To build a system to solve a particular problem, we need to do four things:

  1. **Define the problem precisely**. This definition must include precise specifications of what the initial situation(s) will be as well as what final situations constitute acceptable solutions to the problem.

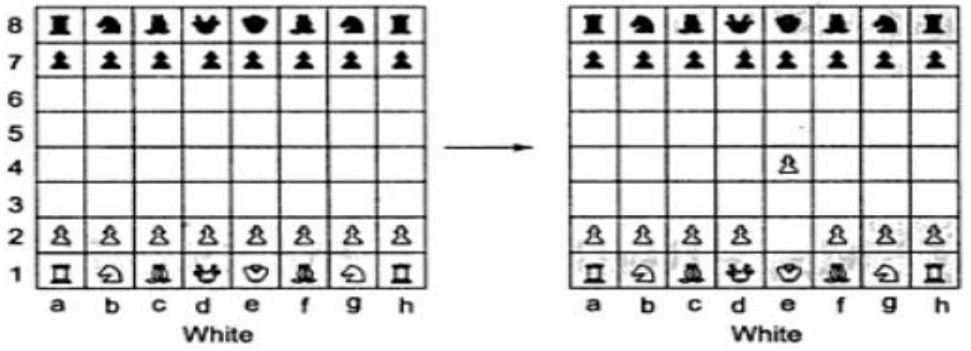
  2 **Analyse the problem**. A few very important features can have an immense impact on the appropriateness of various possible technique(s)for solving the problem.

  3. **Isolate and represent** the task knowledge that is necessary to solve the problem.

  4. **Choose the best problem-solving technique**(s) and apply it (them) to the particular problem.

State-Space Search

- To build a program that could "Play chess," we would first have to specify the starting position of the chess hoard, the rules that define the legal moves, and the board positions that represent a win for one side or the other.
- The starting position can he described as an 8-by-8 array where each position contains a symbol standing for the appropriate piece in the official chess opening position.
- We can define as our goal any board position in which the opponent does not have a legal move and his or her king is under attack.
- The legal moves provide the way of getting from the initial state to a goal state. They can he described easily as a set of rules consisting of two parts: a left side that serves as a pattern to he matched against the current board position and a right side that describes the change to be made to the board position in reflect the move.



- We have to write a very large number of rules since there have to be a separate rule for each of the 10^{120} possible board positions. Using so many rules poses two serious practical difficulties:
  - No person could ever supply a complete set of such rules. It would take too long and could certainly not be done without mistakes.
  - No program could easily handle all those rules. Although hashing scheme could be used to find the relevant rules for each move fairly quickly, just storing that many rules poses serious difficulties.
- We should write the rules describing the legal moves in as general a way possible. The convenient notation for describing patterns and substitutions is as follows:

White pawn at
Square(file e, rank 2)
AND
Square(file e, rank 3)
is empty
AND
Square(file e, rank 4)
is empty
→
move pawn from
Square(file e, rank 2)
to Square(file e, rank 4)

- A problem can be defined in a **"State Space",** where each state corresponds to a legal position of the board.
  - We can start at an initial state using a set of rules to move from one state to another, and attempting to end up in one of a set of final states.
- The state space representation forms the basis of most of the Al methods. Its structure corresponds to the structure of problem solving in two important ways:
  - It allows formal definition of a problem as the need to convert some given situation into some desired situation using a set of permissible operations.

- It permits us to define the process of solving a particular problem as a combination of known techniques) each represented as a role defining a single step in the space) and search, the general technique of exploring the space to try to find some path from the current state to a goal state. Search is a very important process in the solution of hard problems for which no more direct techniques are available.

Water Jug Problem

- Consider the following problem: A Water Jug Problem: You are given two jugs, a 4-gallon one and a 3-gallon one, a pump which has unlimited water which you can use to fill the jug, and the ground on which water may be poured. Neither jug has any measuring markings on it. How can you get exactly 2 gallons of water in the 4-gallon jug?
- **State Representation and Initial State** – we will represent a state of the problem as a tuple (x, y) where x represents the amount of water in the 4-gallon jug and y represents the amount of water in the 3-gallon jug. Note $0 \le x \le 4$, and $0 \le y \le 3$.
- Our Initial state: (0,0) Goal state = (2,y) where $0 \le y \le 3$.
- To solve this we have to make some assumptions not mentioned in the problem. They are:

  1. We can fill a jug from the pump.
  2. We can pour water out of a jug to the ground.
  3. We can pour water from one jug to another.
  4. There is no measuring device available. To solve the water jug problem, all we need is a control structure that 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 ii ii corresponds to a goal state. As long as it does not, the cycle continues. Clearly the speed with which the problem gets solved depends on the mechanism that is used to select the next operation to be performed.
- We must define a set of operators that will take us from one state to another:

1	(x, y) if $x < 4$	$\rightarrow (4, y)$	Fill the 4-gallon jug
2	(x, y) if $y < 3$	$\rightarrow (x, 3)$	Fill the 3-gallon jug
3	(x, y) if $x > 0$	$\rightarrow (x - d, y)$	Pour some water out of the 4-gallon jug
4	(x, y) if $y > 0$	$\rightarrow (x, y - d)$	Pour some water out of the 3-gallon jug
5	(x, y) if $x > 0$	$\rightarrow (0, y)$	Empty the 4-gallon jug on the ground
6	(x, y) if $y > 0$	$\rightarrow (x, 0)$	Empty the 3-gallon jug on the ground
7	(x, y) if $x + y \geq 4$ and $y > 0$	$\rightarrow (4, y - (4 - x))$	Pour water from the 3-gallon jug into the 4-gallon jug until the 4-gallon jug is full
8	(x, y) if $x + y \geq 3$ and $x > 0$	$\rightarrow (x - (3 - y), 3)$	Pour water from the 4-gallon jug into the 3-gallon jug until the 3-gallon jug is full
9	(x, y) if $x + y \leq 4$ and $y > 0$	$\rightarrow (x + y, 0)$	Pour all the water from the 3-gallon jug into the 4-gallon jug
10	(x, y) if $x + y \leq 3$ and $x > 0$	$\rightarrow (0, x + y)$	Pour all the water from the 4-gallon jug into the 3-gallon jug
11	$(0, 2)$	$\rightarrow (2, 0)$	Pour the 2 gallons from the 3-gallon jug into the 4-gallon jug
12	$(2, y)$	$\rightarrow (0, y)$	Empty the 2 gallons in the 4-gallon jug on the ground

Gallons in the 4-Gallon Jug	Gallons in the 3-Gallon Jug	Rule Applied
0	0	
		2
0	3	
		9
3	0	
		2
3	3	
		7
4	2	
		5 or 12
0	2	
		9 or 11
2	0	

- The first step toward the design of a program to solve a problem must be the creation of a formal and manipulable description of the problem itself. We should be able to write programs that can themselves produce such formal descriptions from informal ones. This process is called **Operationalization.**
- In order to provide a formal description of a problem, we must do the following:
  - **Define a state space** that contains all the possible configurations of the relevant objects (and perhaps some impossible ones). It is, of course, possible to define this space without explicitly enumerating all of the states it contains.
  - Specify one or more states within that space that describes possible situations from which the problem-solving process may start. These states are called the **Initial states.**
  - Specify one or more states that would be acceptable as solutions to the problem. These states are called **goal states.**
  - Specify a **set of rules** that describe the actions (operators) available.

- The problem can then be solved by using the rules, in combination with an appropriate **Control Strategy**, to move through the problem space until a path from an initial state to a goal state is found. Thus the process of search is fundamental to the problem-solving process.

Control Strategies

- It is important to decide which rule to apply next during the process of searching for a solution to a problem. This question arises when more than one will have its left side match the current state.
- The decisions made will have a crucial impact on how quickly a problem is finally solved.
- The following are the requirements of a control strategy:
  - The first requirement of a good control strategy is that **it causes motion**. Consider again the water jug problem. Suppose we implemented the simple control strategy of starting each time at the top of the list of rules and choosing the first applicable one. If we did that, we would never solve the problem. We would continue indefinitely filling the 4-gallon jug with water. Control strategies that do not cause motion will never lead to a solution.
  - The second requirement of a good control strategy is that it **be systematic**. Let us consider a simple control strategy for the water jug problem. On each cycle, choose at random from among the applicable rules. This strategy is better than the first. It causes motion. It will lead to a solution eventually. Sometimes

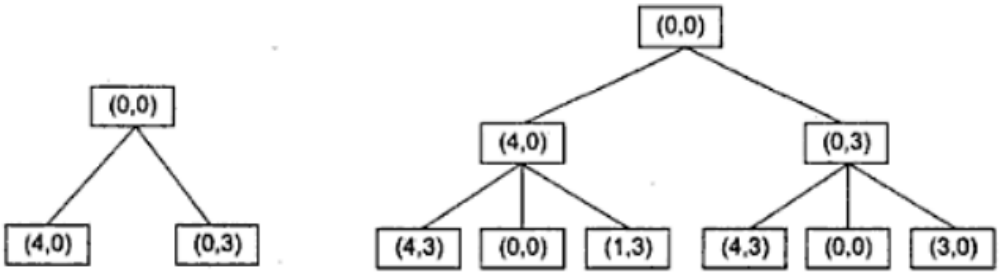
we arrive at the **same state several times** during the process and to use many more steps than are necessary. Because the control strategy is not systematic, we may explore a particular useless sequence of operators several times before we finally find a solution. The requirement that a control strategy be systematic corresponds to the need for **global motion (over the course of several steps) as well as for local motion** (over the course of a single step).

Breadth First Search

- For each leaf node, generate all its successors by applying all the rules that are appropriate. Continue this process until some rule produces a goal state. This process, called **Breadth-First search**.

Algorithm: Breadth-First Search

1. Create a variable called NODE-LIST and set it to the initial state.
2. Until a goal state is found or NODE-LIST is empty do:
   (a) Remove the first element from NODE-LIST and call it E. If NODE-LIST was empty, quit.
   (b) For each way that each rule can match the state described in E do:
      i. Apply the rule to generate a new state.
      ii. If the new state is a goal state, quit and return this state.
      iii. Otherwise, add the new state to the end of NODE-LIST.



- **Advantages of Breadth-First Search:**
  - Breadth-first search will not get trapped exploring a blind alley.
  - If there is a solution, then breadth-first search is guaranteed to find it. Furthermore, if there are multiple solutions, then a minimal solution will be found. This is guaranteed by the fact that longer paths are never explored until all shorter ones have already been examined.

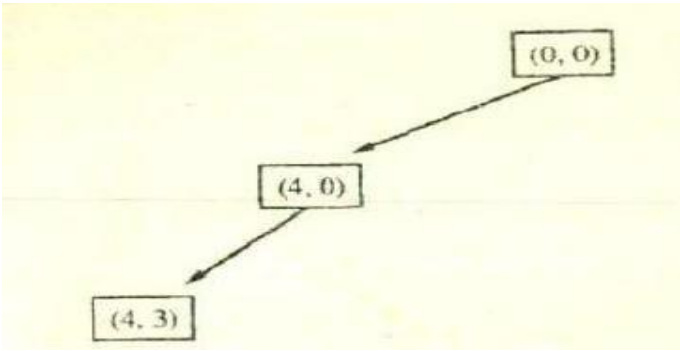
- **Disadvantages of Breadth-First Search:**
  - All of the tree that has so far been generated must be stored which requires more memory.
  - In breadth-first search, all parts of the tree must be examined to level *n* before any nodes on level n + 1 can be examined. This is particularly significant if many acceptable solutions exist.

Depth First Search

- We could pursue a single branch of the tree until it yields a solution or until a decision to terminate the path is made.
- It makes sense to terminate a path if it reaches a dead-end, produces a previous state, or becomes longer than some prespecified "futility" limit.
- In such a case backtracking occurs. The most recently created state from which alternative moves are available will be revisited and a new state will he created. This form of backtracking is called **Chronological Backtracking** because the order in which steps arc undone depends only on the temporal sequence in which the steps were originally made.
- The search procedure we have just described is called **Depth-First search.**

Algorithm: Depth-First Search

1. If the initial state is a goal state, quit and return success.
2. Otherwise, do the following until success or failure is signalled:
   (a) Generate a successor, E, of the initial state. It there are no more successors,
   signal failure.
   (b) Call Depth-First Search with E as the initial state.
   (c) If success is returned, signal success. Otherwise continue in this loop.



- **Advantages of Depth-First Search:**
  - Depth-first search requires less memory since only the nodes on the current path are stored.
  - By chance (or if care is taken in ordering the alternative successor states), depth first search may find a solution without examining much of the search space at all. Depth-first search can stop when one of solution is found.
- **Disadvantages of Depth-First Search:**
  - Depth- first searching may follow a single unfruitful path for a very long time.
  - Depth-first search, may find a long path to a solution in one part of the tree, when a shorter path exists in some other unexplored part of the tree.

Problem Characteristics

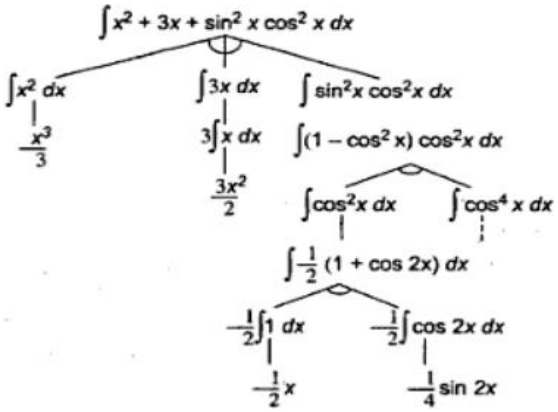
- *A* **heuristic** *is* a technique that improves the efficiency of a search process, possibly by sacrificing claims such as completeness. Heuristics are like tour guides. They are good to the extent that they point in generally interesting directions: they are bad to the extent that they may miss points of interest to particular individuals. But, on the average, they improve the quality of the paths that are explored.
- Heuristic search is a very general method applicable to a large class of problems; it encompasses a variety of specific techniques, each of which is particularly effective for a small class of problems. In order to choose the most appropriate method (or combination of methods for a particular problem, it is necessary to analyze the problem along several key dimensions:
  - Is the problem decomposable into a set of (nearly) independent smaller or easier sub problems?
  - Can solution steps be ignored or at least undone if they prove unwise?
  - Is the problem's universe predictable?
  - Is good solution to the problem obvious without comparison to all other possible solutions?
  - Is the desired solution a state of the world or a path to a state?
  - Is a large amount of knowledge absolutely required to solve the problem, or is knowledge important only to constrain the search?
  - Can a computer that is simple given the problem return the solution, or will the solution of the problem require interaction n between the computer and a person?

1. Is the Problem Decomposab1e?

- Suppose we want to solve the problem of computing the expression:

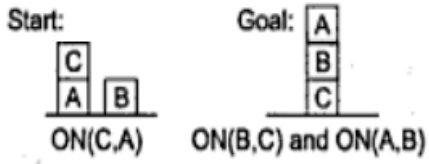
$$\int (x^2 + 3x + \sin^{2*} \cdot \cos^2 x)\, dx$$

- We can solve this problem by breaking it down into three smaller problems each of which we can then solve by using a small collection of specific rules.
- The figure shows the problem tree that will be generated by the process of problem decomposition as it can be exploited by a simple recursive integration program that works as follows:
- At each step. it checks to see whether the problem it is working on is immediately solvable. If so, then the answer is returned directly.
- If the problem is not easily solvable, the integrator checks to see whether it can decompose the problem into smaller problems. If it can, it creates those problems and calls itself recursively on them. Using this technique of problem decomposition we can often solve very large problems easily.

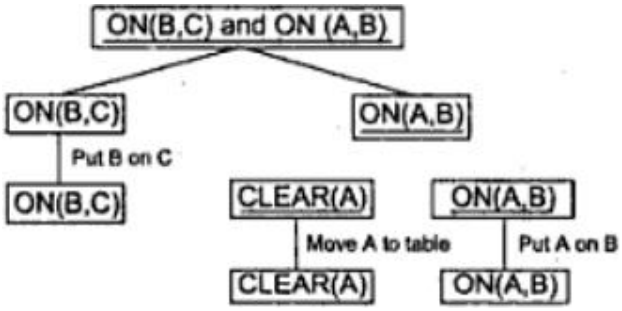


- **Example-Blocks World Problem:** Assume that the following operators are available-

1. CLEAR (*x*) [block *x* has nothing on it] → ON (*x*, Table) [pick up *x* and put it on the table]
2. CLEAR (*x*) and CLEAR (*y*) → ON (*x*, *y*) [put *x* on *y*]



- Applying the technique of problem decomposition to this simple blocks world example would lead to a solution tree such as that shown below:

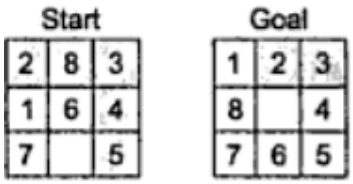


- In the figure, goals are underlined. States that have been achieved are not underlined. The idea of this solution is to reduce the problem of getting B on C and A on B to two separate problems.
- The **first** of these new problems, getting B on C. is simple, given the start state. Simply put B on C. The second sub goal is not quite so simple. Since the only operators we have allow us to pick up single blocks at a time, we have to clear off B by removing C before we can pick up A and put it on B This can easily be done.
- However, if we now try to combine the two sub solutions into one solution, we will fail. Regardless of which one we do first, we will not be able to do the second as we had planned.
- In this problem, the two sub problems are not independent. They interact and those interactions must be considered in order to arrive at a solution for the entire problem.

2. Can Solution Steps Be Ignored or Undone?

- **Theorem Proving:** We proceed by first proving lemma that we think will be useful. Eventually, we realize that the lemma is no help at all.
- Everything we need to prove the theorem is still true. Any rules that could have been applied can still be applied.
- We can just proceed as we should have in the first place. All we have lost is the effort that was spent exploring the blind alley.
- **The 8-Puzzle**: The 8-puzzle is a square tray in which are placed eight square tiles. The remaining ninth square is uncovered. Each tile has a number on it. A tile that is adjacent to the blank space can he slid into that space. A game consists of a starting position and a specified goal position. The goal is to transform the starting position into a goal position by sliding the tiles around.

➢



- In attempting to solve the 8-puzzle, we might make a stupid move. We might start by sliding tile 5 into the empty space. Having done that, we cannot change our mind and immediately slide tile 6 into the empty space since the empty space will essentially have moved.
- But we can backtrack and undo the first move, sliding tile 5 back to where it was. Then we can move tile 6. Mistakes can still be recovered from but not quite easily. An additional step must be performed to undo each incorrect step,
- The control mechanism for an 8.puzzle solver must keep track of the order in which operations are performed so that the operations can be undone one at a time if necessary.
- **Chess:** Now consider again the problem of playing chess. Suppose a chess-playing program makes a stupid move and realizes it a couple of moves later. It cannot simply play as though it had never made the stupid move. Nor can it simply back up and start the game over from that point. All it can do is to try to make the best of the current situation and go on from there.
- The three problems—theorem proving, the 8-puzzle, and chess—illustrate the differences between three important classes of problems:
  • **Ignorable**(e.g... theorem proving), in which solution steps can be ignored
  • **Recoverable**(e.g... 8-puzzle), in which solution steps can be undone.
  • **Irrecoverable**(e.g... chess), in which solution steps cannot be undone
- The **recoverability of a problem** plays an important role in determining the complexity of the control structure necessary for the problem's solution.
  - **Ignorable problems** can be solved using a simple control structure that never backtracks. Such a control structure is easy to implement.
  - **Recoverable problems** can be solved by a slightly more complicated control strategy that does sometimes make mistakes. Backtracking will be necessary to recover from such mistakes, so the control structure must be implemented using a push-down stack, in which decisions are recorded in case they need to be undone later.

- **Irrecoverable problems**, on the other hand, will need to be solved by a system that expends a great deal of effort making each decision since the decision must be final

3. Is the Universe Predictable?

- **8-puzzle:** Every time we make a move, we know exactly what will happen. This means that it is possible to plan an entire sequence of moves and be confident that we know what the resulting state will be. We can use planning to avoid having to undo actual moves, although it still be necessary to backtrack past those moves one at a time during the planning process. Thus a control structure that allows backtracking will be necessary.
- **Play Bridge:** One of the decisions we will have to make is which card to play on the first trick. What we would like to do is to plan the entire hand before making that first play. But now it is not possible to do such planning with certainty since we cannot know exactly where all the cards are what the other players will do on their turns. The best we can do is to investigate several plans and use probabilities of the various outcomes to choose a plan that has the highest estimated probability of leading to a good score on the hand.
- **Certain–outcome problems** (e.g... 8-puzzle): The open-loop approach will work fine since the result of an action can he predicted perfectly. Thus, planning can be used to generate a sequence of operators that is guaranteed to lead to a solution.
- **Uncertain–outcome problems** (e.g... Bridge): Planning can at best generate a sequence of operators that has a good probability of leading to a solution. To solve such problems, we need to allow for a process of plan revision to take place as the plan is carried out and the necessary feedback is provided. In addition to providing no guarantee of art actual solution, planning for uncertain -outcome problems has the drawback that it is often very expensive since the number of solution paths that need to be explored increases exponentially with the number of points at which the outcome cannot be predicted.

4. Is a Good Solution Absolute or Relative?

- **Simple facts problem:** Consider the problem of answering questions based on a database of simple facts, such as the following:

  1. Marcus was a man.
  2. Marcus was a Pompeian.
  3. Marcus was born in 40 A.D.
  4. All men are mortal.
  5. All Pompeians died when the volcano erupted in 79 A.D.
  6 No mortal lives longer than 150 years.
  7.11 is now 1991 A.D.

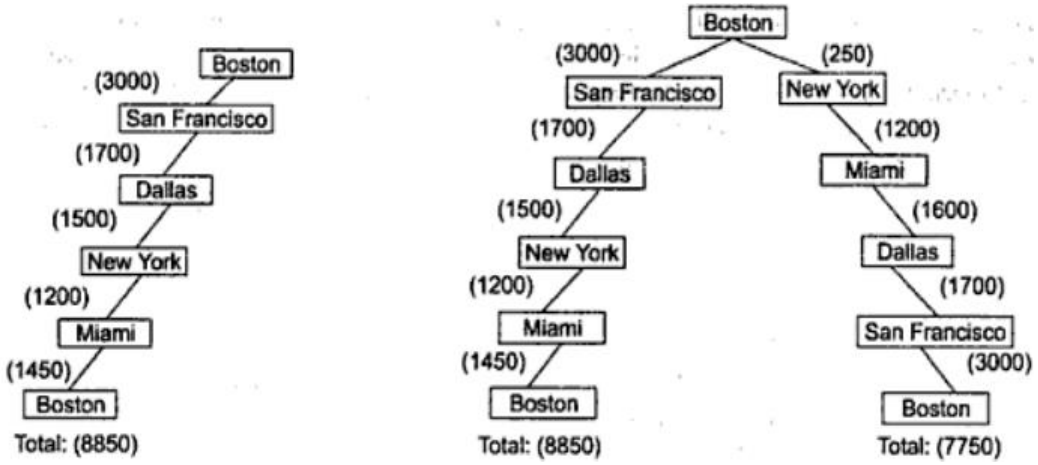
- Suppose we ask the question '**Is Marcus alive?'.** By representing each of these facts in a formal language such as predicate logic, and then using formal inference methods we can fairly easily derive an answer to the question. Since all we are interested in is the answer to the que7sion, it does not matter which path we follow. If we do follow one path successfully to the answer, there is no reason to go back and see it some other path might also lead to a solution.

		Justification
1.	Marcus was a man.	axiom 1
4.	All men are mortal.	axiom 4
8.	Marcus is mortal.	1, 4
3.	Marcus was born in 40 A.D.	axiom 3
7.	It is now 1991 A.D.	axiom 7
9.	Marcus' age is 1951 years.	3, 7
6.	No mortal lives longer than 150 years.	axiom 6
10.	Marcus is dead.	8, 6, 9
	OR	
7.	It is now 1991 A.D.	axiom 7
5.	All Pompeians died in 79 A.D.	axiom 5
11.	All Pompeians are dead now.	7, 5
2.	Marcus was a Pompeian.	axiom 2
12.	Marcus is dead.	11, 2

- **Travelling salesman problem:** Our goal is to find the shortest route that visits each city exactly once. Suppose the cities to be visited and the distances between them are as shown:

	Boston	New York	Miami	Dallas	S.F.
Boston		250	1450	1700	3000
New York	250		1200	1500	2900
Miami	1450	1200		1600	3300
Dallas	1700	1500	1600		1700
S.F.	3000	2900	3300	1700	

- One place the salesman could start is Boston. In that case, one path that might be followed is shown, which is 8850 miles long. But is this the solution to the problem? The answer is that we cannot be sure unless we also try all other paths to make sure that none of them is shorter. In this case, the first path is definitely not the solution to the salesman's problem.



- **Best-path problems**: These are, in general, computationally harder than Any-path problems. No heuristic that could possibly miss the best solution can be used.
- **Any-path problems**: can often be solved in a reasonable amount of time by using heuristics that suggest good paths to explore. If the heuristics are not perfect, the search for a solution may not be as direct as possible. So a much more exhaustive search will be performed.

5. Is the Solution a State or a Path?

- **Consistent interpretation for the sentence (NLU):** There are several components of this sentence, each of which, in isolation, may have more than one interpretation.

  "The Bank president ate a dish of pasta salad with the fork"

- Some of the sources of ambiguity in this sentence are the following:
  - The word "bank" may refer either to a financial institution or to a side of a river. But only one of these may have a president.
  - The word "dish" is the object of the verb "eat." It is possible that a dish was eaten. But it is more likely that the pasta salad in the dish was eaten.
  - Pasta salad is a salad containing pasta. But there are other ways meanings can be formed from pair of nouns. For example, dog food does not normally contain dog.
- Some search may be required to find a complete interpretation for the sentence. But to solve the problem of finding the interpretation we need to produce only the interpretation itself. No record of the processing by which the interpretation was found is necessary.
- **Water Jug Problem:** It is not sufficient to report that we have solved the problem and that the final state (2, 0). For this kind of problem,

what we really must report is not the final state but the path that we found to that state. Thus a statement of a solution to this problem must be a sequence of operations called a **"plan"** that produces the final state.

- The difference between these problems is:
  - Problems whose solution is a **state.**
  - Problems whose solution is a **path to a state.**
- In water jug problem, we must re-describe the states so that each state represents a partial path to a solution rather than just a single state of the world.

6. What Is the Role of Knowledge?

- **Playing chess**: Suppose you had unlimited computing power available. The knowledge that would be required by a perfect program is very little—just the rules for determining legal moves and some simple control mechanism that implements a search procedure.
- Additional knowledge about such things as good strategy, and tactics could of course help considerably to constrain the search and speed up the execution of the program.
- **Scanning daily newspapers**: to decide which are supporting the Democrats and which are supporting the Republicans in some upcoming election. Again assuming unlimited computing power, how much knowledge would be required by a computer trying to solve this problem? It would have to know such things as:

  • The names of the candidates in each party.

  • The fact that if the major thing you want to see done is have taxes lowered, you are probably supporting the Republicans.

  • The fact that it the major thing you want to see done is improved education for minority students, you are probably supporting the Democrats.

  • The fact that if you are opposed to big government you are probably supporting the Republicans.
- These two problems, chess and newspaper story understanding, illustrate the difference between problems for which a lot of knowledge is important only to constrain the search for a solution and those for which a lot of knowledge is required even to be able to recognize a solution.

7. Does the Task Require Interaction with a Person?

- Sometimes it is useful to program computers to solve problems in ways that the majority of people would not be able to understand. This is fine if the level of the interaction between the computer and its human users is **program-in solution-out**.

- But increasingly we are building programs that require intermediate interaction with people, both to provide additional input to the program and to provide additional reassurance to the user.
- We must distinguish between two types of problems:
- **Solitary:** in which the computer is given a problem description and produces an answer with no intermediate communication and with no demand for an explanation of the reasoning process
- **Conversational**: in which there is intermediate communication between a person and the computer, either to provide additional assistance to the computer or to provide additional information to the user, or both.

8. Problem Classification

- **Generic control strategy-Classification:** The task here is to examine an input and then decide which of a set of known classes the input is an instance of. Most diagnostic tasks, including medical diagnosis as well as diagnosis of faults in mechanical devices are examples of classification.
- **Propose and Refine**: Many design and planning problems can be attacked with this strategy.

Heuristic Search Techniques

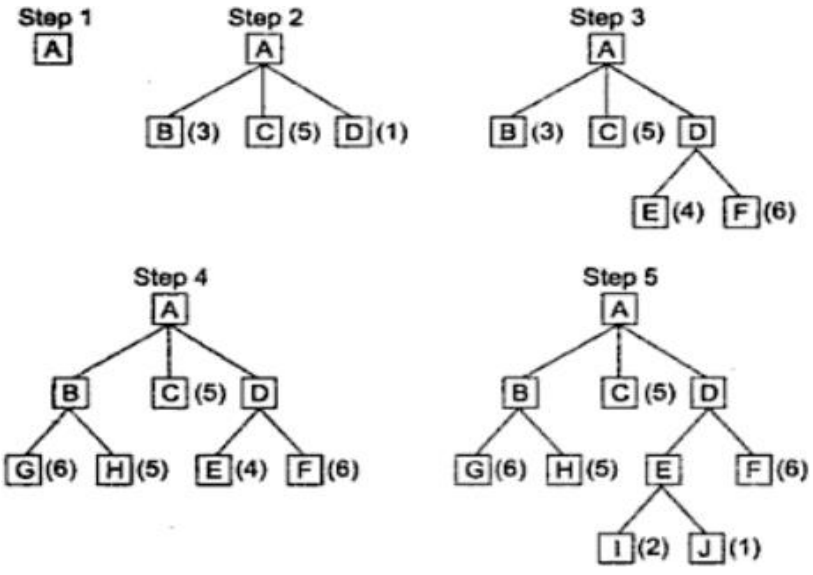
- A framework for describing search methods is provided and several general-purpose search techniques are discussed.
- There are many varieties of heuristic search. They can be described independently of any particular task or problem domain. But when applied to particular problems, their efficacy is highly dependent on the way they exploit domain-specific knowledge
- Over last three decades of AI research, these techniques continue to provide the framework into which domain-specific knowledge can be placed, either by hand or as a result of automatic learning. Thus they continue to form the core of most Al systems.

Best-First Search

- Best-first search, is a way of combining the advantages of both depth-first and breadth-first search into a single method.
- Depth-first search is good because it allows a solution to be found without all competing branches having to be expanded. Breadth-first search is good because it does not get trapped on dead-end paths.
- One way of combining the two is to follow a single path at a time, but switch paths whenever some competing path looks more promising than the current one does.
- At each step of the best-first search process, we select the most promising of the nodes we have generated so far. This is done by

applying an appropriate heuristic function to each of them, We then expand the chosen node by using the rules to generate its successors.

- If one of them is a solution, we can quit. If not, all those new nodes are added to the set of nodes generated so far. Again the most promising node is selected and the process continues. A bit of depth first searching occurs as the most promising branch is explored.
- But eventually, it a solution is not found, that branch will start to look less promising than one of the top-level branches that had been ignored. At that point, the now more promising, previously ignored branch will be explored. But the old branch is not forgotten, Its last node remains in the set of generated but unexpanded nodes. The search can return to it whenever all the others get bad enough, that it is again the most promising path.
- Initially, there is only one node, so it will be expanded. Doing so generates three new nodes. The heuristic function, is an estimate of the cost of getting to a solution from a given node, is applied to each of these new nodes.
- Since node D is the most promising, it is expanded next, producing two successor nodes, E and F. Now the heuristic function is applied to them.
- Now another path that going through node B looks more promising, so it is pursued, generating nodes G and H. But again when these new nodes are evaluated they look less promising than another path, so attention is returned to the path through D to E. E is then expanded, yielding nodes I and J.
- At the next step, J will be expanded, since it is the most promising. This process can continue until a solution is found.



- In best-first search, one move is selected, but the others are kept around so that they can be revisited later if the selected path becomes less promising.
- Further, the best available state is selected in best first search, even it that state has a value that is lower than the value of the state that was just explored.
- In a best-first search of a tree, it is sometimes important to search a graph instead so that duplicate paths will not be pursued.
- An algorithm to do this will operate by searching a directed graph in which each node represents a point in the problem space. Each node will contain, in addition to a description of the problem state it represents, an indication of how promising it is; a parent link that points back to the best node from which it came, and a list of the nodes that were generated from it.
- The parent link will make it possible to recover the path to the goal once the goal is found. The list of successors will make it possible, if a better path is found to an already existing node, to propagate the improvement down to its successors.
- We will call a graph of this sort an **OR graph**, since each of its branches represents an alternative problem-solving path.
- To implement such a graph-search procedure, we will need to use two lists of nodes:
  - **OPEN**—nodes that have been generated and have had the heuristic function applied to them but which have not yet been examined (i.e.. had their successors generated) OPEN is actually a priority queue in which the elements with the highest priority are those with the most promising value of the heuristic function.
  - **CLOSED**— -nodes that have already been examined. We need to keep these nodes in memory if we want to search a graph rather than a tree, since whenever a new node is generated; we need to check whether it has been generated before.
- We will also need a heuristic function that estimates the merits of each node we generate. The function f^1 that gives the true evaluation of the node.
- We define this function as the sum of two components: **g and h^1.**
- The **function g** is a measure of the cost of getting from the initial state to the current node.
- The **function h^1** is an estimate of the additional cost of getting from the current node to a goal state.
- The combined **function f^1** represents an estimate of the cost of getting from the initial state to a goal state along the path that generated the

current node. If more than one path is generated the node, then the algorithm will record the best one.

- This process can be summarized as follows:

Algorithm: Best-First Search

1. Start with OPEN containing just the initial state.
2. Until a goal is found or there are no nodes left on OPEN do:

(a) Pick the best node on OPEN.

(b) Generate its successors.

(c) For each successor do:

i. If it has not been generated before, evaluate it, add it to OPEN, and record its parent.

ii. If it has been generated before, change the parent if this new path is, better than the previous one. In that case, update the cost of getting to this node and to any successors that this node may already have.

- The best first search algorithm is a simplification of an algorithm called **A*.**
- This algorithm uses the same f^1, g, and h^1 functions, as well as the lists OPEN and CLOSED.

Iterative Deepening:

- A number of ideas for searching two-player game trees have led to new algorithm for single-agent heuristic search. One such idea is **"iterative deepening"** originally used in **CHESS**.
- Rather than searching to a fixed depth in the game tree, CHESS first searched only a single ply, applying its static evaluation function to the result of each of its possible moves.
- It then initiated a new minimax search, to a depth of two ply. This was followed by a three-ply search, then a four-ply search etc.
- The name "iterative deepening" derives from the fact that on each iteration, the tree is searched one level deeper.
- Since it is impossible to know in advance how long a fixed-depth tree search will take, a program may find itself running out of time. With iterative deepening, the current search can be aborted at any time and the best move found by the previous iteration can be played.
- Iterative deepening can also be used in improve the performance of the A* search algorithm.

Algorithm: Iterative-Deepening-A*

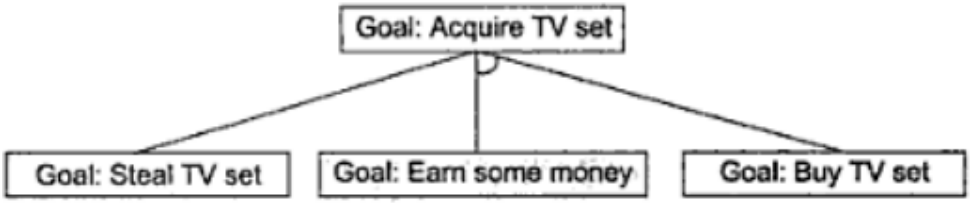
1. Set THRESHOLD = the heuristic evaluation of the start state.
2. Conduct a depth-first search, pruning any branch when its total cost function ($g + h^1$) exceeds THRFSHOLD. If a solution path is found during the search, return it.

3. Otherwise, increment THRESHOLD by the minimum amount it was exceeded during the previous step, and then go to Step 2.

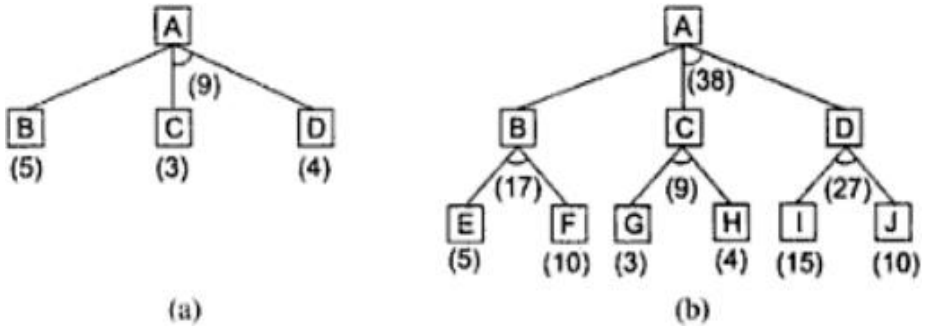
- Iterative-Deepening-A* (IDA*) is guaranteed to find an optimal solution provided that h^1 is an admissible heuristic.

Problem Reduction: AND-OR Graphs

- **AND-OR graph** (or tree), is useful for representing the solution of problems that can be solved by decomposing them into a set of smaller problems, all of which must then he solved.
- This decomposition, or reduction, generates arcs that we call AND arcs. One AND arc may point to any number of successor nodes all of which must be solved in order for the arc to point to a solution.
- Several arcs may emerge from a single node, indicating a variety of ways in which the original problem might be solved. This is why the structure is called not simply an AND graph but rather an AND-OR graph. AND arcs are indicated with a line connecting all the components.



- In order to find solutions in an AND-OR graph, we need an algorithm with the ability to handle the AND arcs appropriately. This algorithm should find a path from the starting node of the graph to a set of nodes representing solution states.
- The top node, A, has been expanded, producing two arcs, one leading to B and one leading to C and D. The numbers at each node represent the value of $\mathbf{f^1}$ at that node. Assume, that every operation has a uniform cost, so each arc with a single successor has a cost of I and each AND arc with multiple successors has a cost of 1 for each of its components.
- If we look just at the nodes and choose for expansion the one with the lowest; $\mathbf{f^1}$ value, we must select C.
- But using the information now available, it would be better to explore the path going through B since to use C we must also use D. (for a total cost of 9 (C+D+2) compared to the cost of 6 that we get by going through B.
- The problem is that the choice of which node to expand next must depend not only on the $\mathbf{f^1}$ value of that node but also on whether that node is part of the current best path from the initial node.



- The most promising single node is (G with an $\mathbf{f^1}$ value of 3). It is even part of the most promising arc G-H, with a total cost of 9. But that are is not part of the current best path since to use it we must also use the arc I-J with a cost of 27.
- The path from A through B, to E and F is better, with a total cost of 18. So we should not expand G next: rather we should examine either E or F.
- In order to describe an algorithm for searching an AND-OR graph, we need to exploit a value called **FULILITY**. If the estimated cost of a solution becomes greater than the value of FUTILITY, then we abandon the search.
- FUTILITY should be chosen to correspond to a threshold such that any solution with a cost above it is too expensive to be practical, even if it is found.

Algorithm: Problem Reduction

1 Initialize the graph to the starting node.

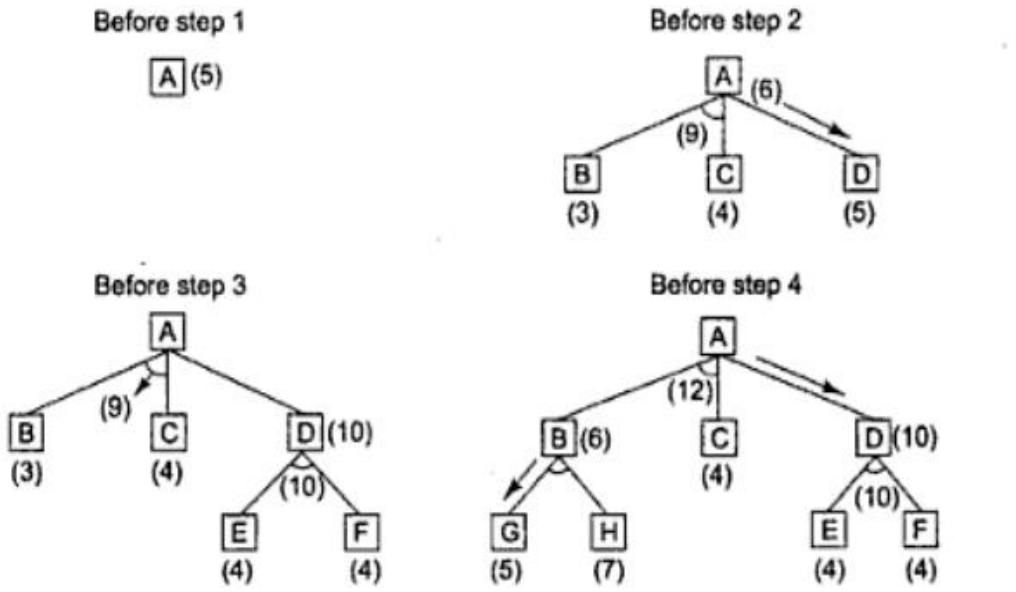
2. Loop until the starling node is labelled SOLVED or until its cost goes above FUTILITY:

(a) Traverse the graph, starting at the initial node and following the current best path, and accumulate the set of nodes that are or that path and not yet been expanded or labelled 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 $\mathbf{f^1}$ .If $\mathbf{f^1}$ of any node is 0, mark that node as SOLVED.

(c) Change the $\mathbf{f^1}$ 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. But now expanded nodes must be re-examined so that the best current path can he selected. Thus it is important that their **f^1** values be the best estimates available.



- At step 1, A is the only node, so it is at the end of the current best path. It is expanded, yielding nodes B, C and D. The arc to D is labelled as the most promising one emerging from A, since it costs *6* compared to B and C, which costs 9. (Marked arcs are indicated in the figures by arrows.)
- In step 2, node D is chosen for expansion. This process produces one new arc, the AND arc to E and F, with a combined cost estimate of 10. So we update the f^1 value of D to 10.
- Going back one more level, we can see that this makes the AND arc **B-C** better than the arc to D, so it is labelled as the current best path.
- At step 3, we traverse that arc from A and discover the unexpanded nodes B and C. If we want to find a solution along this path, we will have to expand both B and *C* eventually, so let's choose to explore B first. This generates two new arcs, the ones to G and to H.
- Propagating their f^1 values backward, we update f^1 of B to 6. This requires updating the cost of the AND arc B-C to 12 (6+4+2).

- After doing that, the arc to D is again the better path from A, so we record that as the current best path and either node E or node F will be 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.

The AO* Algorithm

- The problem reduction algorithm is a simplification of an AO^* algorithm.
- The AO^* algorithm will use a single structure GRAPH, representing the part of the search graph that has been explicitly generated so far.
- Each node in the graph will point both down to its immediate successors and up to its immediate predecessors. Each node in the graph will also have associated with it an **h^1 value, an estimate of the cost of a path from itself to a set of solution nodes**. So h^1 will serve as the estimate of goodness of a node.

Algorithm: AO* *

1. Let GRAPH consists of only the node representing the initial state. (Call this node INIT). Compute h^1 (INIT).

2. Until INIT is labelled SOLVED or until INIT's h^1 value becomes greater than FUTILITY, repeat the following procedure:

   (a) Trace the labelled arcs from INIT and select for expansion one of the yet unexpanded nodes that occurs on this path. Call the selected node NODE.

   (b) Generate the successors of NODE. If there are none, then assign FUTILITY as the h^1 value of NODE. 'This is equivalent to saying that NODE is not solvable. If there are successors, then for each one called SUCCESSOR that is not also an ancestor of NODE do the following:

      i. Add SUCCESSOR to GRAPH.

      ii. If SUCCESSOR is a terminal node, label it SOLVED and assign it, an h^1 value of 0.

      iii. If SUCCESSOR is not a terminal node, compute its h^1 value.

   (c) Propagate the newly discovered information up the graph by doing the following: Let S the a set of nodes that have been labelled SOLVED or whose h^1 values have been changed and so need to have values propagated back to their parents. Initialize S to NODE. Until S is empty, repeat the following procedure:

      i. If possible, select from S a node none of whose descendants in GRAPH occurs in S. If there is no such node, select any node from S. Call this node CURRENT, and remove it from S.

      ii. Compute the cost of each of the arcs emerging from CURRENT. The cost of each arc is equal to the sum of the $h^{1'}$ values of each of the nodes at the end of the arc plus whatever

the cost of the arc itself is. Assign as CURRENT's new h^1 value the minimum or the costs just computed for the arcs emerging from it.

iii. Mark the best path out of CURRENT by marking the arc that had the minimum cost as computed in the previous step.

iv. Mark CURRENT SOLVED if all of the nodes connected to it through the new labelled arc have been labelled SOLVED.

v. If CURRENT has been labelled SOLVED or if the cost of CURRENT was just changed, then its new status must be propagated back up the graph. So add all of the ancestors of CURRENT to S.

Constraint Satisfaction

- Many problems in Al can be viewed as problems of Constraint Satisfaction in which goal is to discover some problem state that satisfies a given set of constraints.
- Examples of this sort of problem include crypt arithmetic puzzles.
- Design tasks can also be viewed as Constraint Satisfaction problems in which a design must be created within fixed limit, on time, cost, and materials.
- By viewing a problem as one of constraint satisfaction, its often possible to reduce substantially the amount of search that, is required as compared with a method that attempts to form partial solutions.
- A constraint satisfaction approach to solving this problem avoids making guesses on particular assignments of numbers to letters. Instead, the initial set of constraints, which says that each number may correspond to only one letter and that the sums of the digits must be as they are given in the problem, is first augmented to include restrictions that can be inferred from the rules of arithmetic.
- Then, although guessing may still be required, the number of allowable guesses is reduced.
- Constraint satisfaction is a search procedure that operates in a space of Constraint sets.
- The initial state contains the constraints that are originally given in the problem description. A goal state is any state that has been constrained -enough." where "enough' must be defined for cacti problem.
- Constraint satisfaction is a two-step process. First, constraints are discovered and propagated as far as possible throughout the system. Then, if there is still not a solution, search begins. A guess about something is made and added as a new constraint. Propagation can then occur with this new constraint, and so forth.

- The first step, propagation, arises from the fact that there are usually dependencies among the constraints. These dependencies occur because many constraints involve more than one object and many objects participate in more than one constraint.
- So, for example, assume we start with one constraint: N = E + I. Then, if we added the constraint N = 3, we could propagate that to get a stronger constraint on E, namely that E = 2.
- Constraint propagation terminates for one of two reasons.
  - First, a contradiction may be detected. If this happens, then there is no solution consistent with all the known constraints. If the contradiction involves only those constraints that were given as part of the problem specification (as opposed to ones that were guessed during problem solving), then no solution exists.
  - The second possible reason for termination is that the propagation has run out of steam and there are no further changes that can be made on the basis of current knowledge. If this happens and a solution has not yet been adequately specified, then search is necessary to get the process moving again.
  - At this point, the second step begins. Some hypothesis about a way to strengthen the constraints must be made. ln case of the crypt arithmetic problem, for example this usually means guessing a particular value for some letter. Once this has been done, constraint propagation can begin again from this new state. If a solution is found, it can be reported. If still more guesses are required, they can be made. If a contradiction is detected, then backtracking can he used to try a different guess and proceed with it.

Algorithm: Constraint Satisfaction

1. Propagate available constraints. To do this, first set OPEN to the set of all objects that must have values assigned to them in a complete solution. Then do until an inconsistency is detected or until OPEN is empty:

(a) Select an object OB from OPEN. Strengthen as much as possible the set of constraints that apply to OB.

(b) If this set is different from the set that was assigned the last time OB was examined or if this is the first time OB has been examined, then add to 0PEN all objects that share any constraints with OB.

(c) Remove OB from OPEN.

2. If the union of the Constraints discovered above defines a solution, then quit and report the solution.

3. If the union of the constraints discovered above defines a contradiction then return failure.

4. If neither of the above occurs, then it is necessary to make a guess at something in order to proceed. To do this, loop until a solution is found or all possible solutions have been eliminated:

(a) Select an object whose value is not yet determined and select a way of strengthening the constraints on that object.

(b) Recursively inv006Fke constraint satisfaction with the current set of constraints augmented by the strengthening constraint just selected.

Problem:

```
  SEND
+ MORE
 .....
 MONEY
```

Initial State:

No two letters have the same value.
The sums of the digits must be as shown in the problem.

- The goal slate is problem state in which all letters have been assigned a digit in such a way that all the initial constraints are satisfied.
- The solution process proceeds in cycles. At each cycle, two significant things are done :
  - Constraints are propagated by using rules that correspond to the properties of arithmetic.
  - A value is guessed for some letter whose value is not yet determined.
- In the first step, it does not usually matter a great deal what order the propagation is done in, since all available propagations will be performed before the step ends.
- In the second step, the order in which guesses are tried may have a substantial impact on the degree of search that is necessary.
- A few useful heuristics can help to select the best guess to try first. For example, if there is a letter that has only two possible values and other with possible values, there is a better chance of guessing right on the first than on the second.
- Another useful heuristic is that if there is a letter that participates in many constraints then it is a good idea to prefer it to a letter that participates in a few. A guess on such a highly constrained letter will

usually lead quickly either to a contradiction (if it's wrong) or to the generation of many additional constraints (if it is right). A guess on a less constrained letter, on the other hand, provides less information.

- Let Cl, C2, C3, and C4 indicate the carry bits out of the columns, numbering from the right. Initially, rules for propagating constraints generate the following additional constraints:
  • M 1, since two single-digit numbers plus a carry cannot total more than 19; S=8 or 9, since S+M+C3>9 (to generate the carry) and M=1, S+l+C3>9. So S + C3 > 8 and C3 is at most 1.
  • O=0, since S + M(l)+C3(<= 1) must be at least 10 to generate a carry and it can be at most 11. But M is already 1. So O must be 0.
  • N = E or E+1, depending on the value of C2. But N cannot have the same value as E. So, N=E+l and C2 is 1.
  • In order for C2 to be1, the sum of N+ R+CI must be greater than 9, so N+ R must be greater than 18.
  • N + R cannot be greater than 18, even with a carry in, so E cannot be 9.
- At this point, let us assume that no more constraints can be generated. Then, to make progress from here, we must guess. Suppose E is assigned the value 2. (We chose to guess a value for E because it occurs three times and thus interacts highly with the other letters.) Now the next cycle begins. The constraint propagator now observes that:
  • N =3, since N = E+ 1.
  • R= 8 o 9, s ince R+ N (3)+C1 (1 or 0)=2 or 12. But since N is already 3, the sum of these non negative numbers cannot be less than 3. Thus R + 3 + (0 or 1)= 12 and R = 8 or 9.
  • 2 + D= Y or 2 + D = 10 + Y, from the sum in the rightmost column.

Game Playing

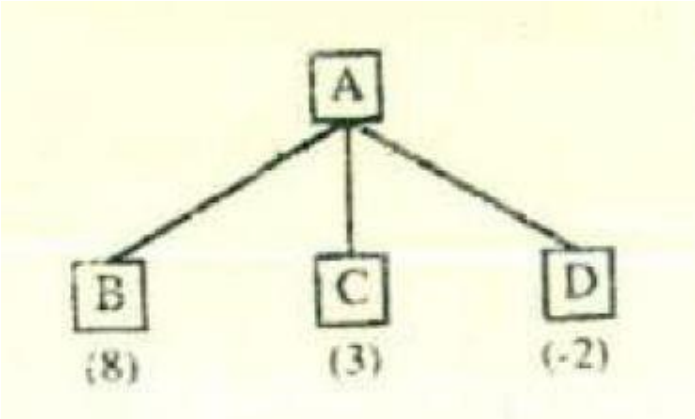
- **Charles Babbage**, the nineteenth-century computer architect, thought about programming his **Analytical Engine** to play chess and later of building a machine to play tic-tac-toe.
- **Claude Shannon** wrote a paper in which he described mechanisms that could be used in a program to play chess.
- **Alan Turing** described a chess-playing program, but he never built it.
- **Arthur Samuel,** succeeded in building the first significant, operational game-playing program. His program played checkers and, in addition to simply playing the game, could learn from its mistakes and improve its performance.
- There were two reasons that games appeared to be a good domain in which to explore machine intelligence:

- They provide a structured task in which it is very easy to measure success or failure.
- They did not obviously require large amounts of Knowledge. They were thought to be solvable by straightforward search from the starting state to a winning position.
- A program that simply does a straightforward search of the game tree will not be able to select even its first move during the lifetime of its opponent. Some kind of heuristic search procedure is necessary.
- We use **Generate-and-Test procedures** in which the testing is done after varying amounts of work by the generator.
  - At one extreme, the generator generates entire proposed solutions, which the tester then evaluates.
  - At the other extreme, the generator generates individual moves in the search space, each of which is then evaluated by the tester and the most promising one is chosen.
- To improve the **effectiveness** of a search-based problem-solving program, two things can be done:
  - Improve the generate procedure so that only good moves (or paths) are generated.
  - Improve the test procedure so that the best moves (or paths) will be recognized and explored first.
- The ideal way to use a search procedure to find a solution to a problem is to generate moves through the problem space until a goal state is reached.
- In order to choose the best move, the resulting board positions must be compared to discover which is most advantageous. This is done using a **static evaluation function**, which uses whatever information it has to evaluate so that the best next move can be chosen.

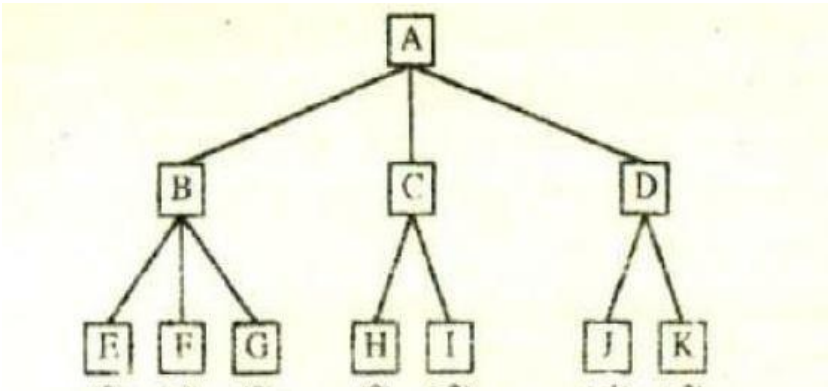
The Minimax Search Procedure

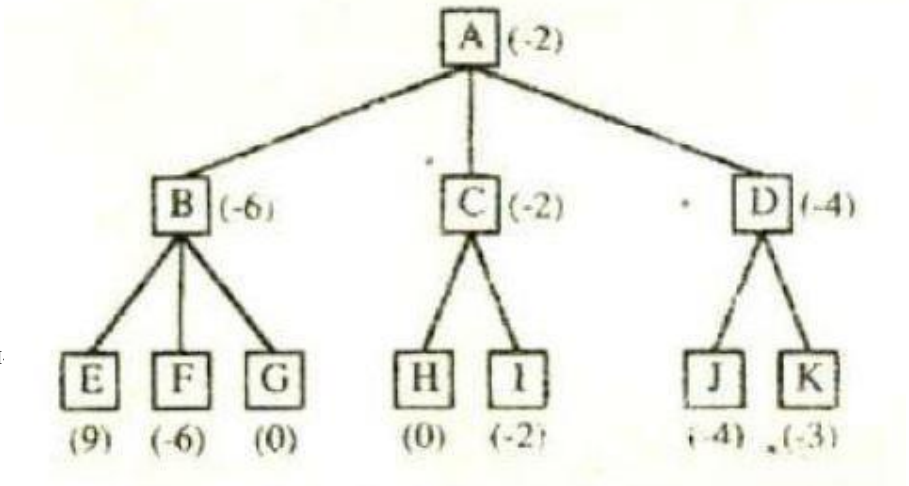
- The minimax search procedure is a depth-first, depth-limited search procedure.
- The idea is to start at the current position and use the plausible-move generator to generate the set of possible successor positions.
- We apply the static evaluation function to those positions and simply choose the best one. Then we can back that value up to the starting position to represent our evaluation of it.
- Our goal is to **maximize** the value of the static evaluation function of the next board position.
- Assume a static evaluation function that returns values ranging from - 10 to 10, with 10 indicating a win for us, and -10 a win for the opponent, and 0 an even match.

- Since our goal is to maximize the value of the heuristic function, we choose to move to B. Backing B's value up to A, we can conclude that A's value is 8.



- After our move, the situation would appear to be very good. But, if we look one move ahead, we will see that one of our pieces also gets captured and so the situation is not as favourable as it seemed.
- So we would like to look ahead to see what will happen to each of the new game positions at the next move which will be made by the opponent.
- Instead of applying the static evaluation function to each of the positions that we just generated, we apply the plausible-move generator, generating a set of successor positions for each position.





- But now we must take into account that the opponent gets to choose which successor moves to make and thus which terminal value should he backed up to the next level.
- Suppose we made move B. Then the opponent must choose among moves E, F, and G. The opponents goal is to minimize the value of the evaluation function, so he or she can be expected to choose move F. This means that if we make move B, the actual position in which we will end up one move later is very bad for us. If node E is selected, it is very good for us. Since at this level we are not the ones to move, we will not get to choose it.
- At the opponent's choice, the minimum value was chosen and backed up. At the level representing our choice, the maximum value was chosen.
- Once the values from the second ply are backed up, it becomes clear that the correct move for us to make at the first level, given the information, is C, since there is nothing the opponent can do from there to produce a value worse than –2.
- This process can he repeated for as many ply as time follows, and more accurate evaluations that are produced can be used to choose the correct move at the top level.
- The alternation of maximizing and minimizing at alternate ply when evaluations are being pushed back up corresponds to the opposing strategies of the two players and hence this method is called. **minimax.**
- The recursive procedure that relies on two auxiliary procedures that are specific to the game being played:
  - **MOVEGEN(Position,Player)** -The plausible-move generator, which returns the list of nodes representing the moves that can be made by player in Position. We call the two players PLAYER-ONE and PLAYER-TWO. In a chess program, we might use the names BLACK and WRITE instead.
  - **STATIC(Position,Player)**—The static evaluation function, which returns a number representing the goodness of position from the standpoint of Player.
- As with any recursive program. a critical issue in the design of the MINIMAX procedure is when to stop the recursion and simply call the static evaluation function.
- The following are the factors that may influence this decision. They include:
  - Has one side Won?
  - How many ply have we already explored?
  - How promising is this path?

- How much time is left?
- How stable is the configuration?

- For the general MINIMAX, we use a function **DEEP-ENOUGH**, which is assumed to evaluate all of these factors and to return TRUE if the search should be stopped at the current level and FALSE otherwise.
- The implementation of DEEP-ENOUGH will take two parameters: **Position and Depth**.
- It will ignore its Position parameter and simply return TRUE if its Depth parameter exceeds a constant cut off value.
- M1NIMAX as a recursive procedure needs to return two results:
  - The backed-up value of the path it chooses.
  - The path itself. We return the entire path even though probably only the first element, representing the best move from the current position, is actually needed.
- MINIMAX returns a structure containing both results: **VALUE and PATH.**
- We define the MININMAX procedure as a recursive function, called initially that takes three parameters.
  - A board position.
  - The current depth of the search, and
  - The player to move.
- So the initial call to compute the best move from the position CURRENT should be :
  - MI1N1MAX(CURRENT, O PLAYER-ONE) if PLAYER-ONE is to move,
  - MINIMAX (CURRENT, 0, PLAYER-TWO) if PLAYER TWO is to move.

Algorithm: MINIMAX (Position, Depth, Player)

1. If *DEEP-ENOUGH (Position, Depth) then* return the structure.

   VALUE= STATIC (Position, Player);

   PATH = nil

This indicates that there is no path from this node and that its value is that determined by the static evaluation function,

2. Otherwise, generate one more ply of the tree by calling the function **MOVEGEN (Position, Player**) and setting SUCCESSORS to the list it returns.

3. If SUCCESSORS is empty, then there are no moves to be made, so return the same structure that would have been returned it DEEP-ENOUGH had returned true.

4. If SUCCESSORS is not empty, then examine each element in turn and keep track of the best one. This *is* done as follows.

Initialize BEST-SCORE to the minimum value that STATIC can return It will be updated in reflect the best score that can be achieved by an element of SUCCESSORS**.**

For each element *SUCC* of SUCCESSORS, do the following:

(a)Set RESULT-SUCC to

MINIMAX(SUCC, *Depth + i, OPPOSITE(Player))*

(b) Set NEW_VALUE to –VALUE (RESULT-SUCC).

(c) If NFW-VALUE > BEST-SCORE, then we have found a successor that better than any that have been examined so far. Record this by doing the following:

i. Set BEST-SCORE to NEW-VALUE.

ii. The best known path is now from CURRENT to SUCC and then on to the appropriate path down from SUCC as determined by the recursive call to MINIMAX. So set BEST-PATH to the result of attaching SUCC to the front of PATH(RESUIT-SUCC);

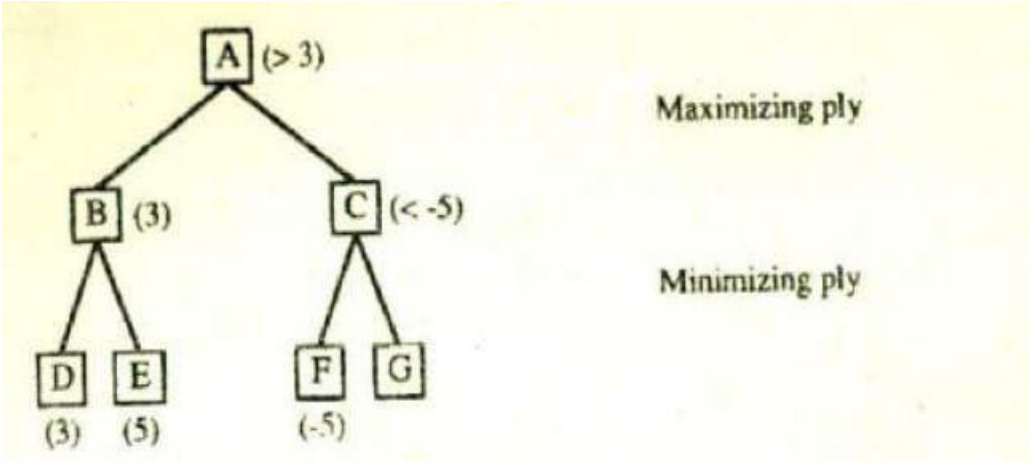
5. Now that all the successors have been examined, we know the value of Position as well as which Path to take from it. So return the structure:

VALUE = BEST-SCORE

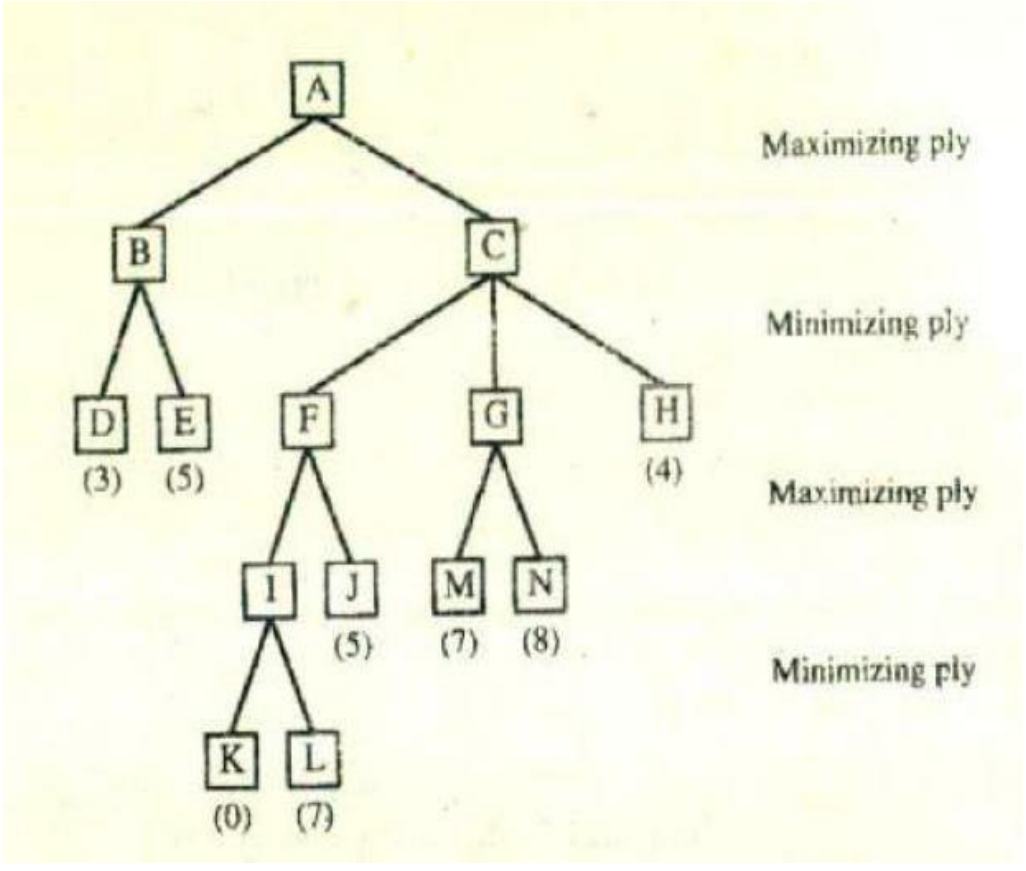
PATH = BEST- PATH

Alpha-Beta Cutoffs

- The minimax procedure is a depth-first process. One path is explored as far as time allows, the static evaluation function is applied to the game positions at the last step of the path, and the value can then be passed up the path one level at a time.
- The efficiency of depth-first procedures can be improved by using **branch-and-bound** techniques in which partial solutions that are clearly worse than known solutions can be abandoned early.
- For this, we require to store of the length of the best path found so far. If a later partial path outgrew that bound, it was abandoned.
- It is necessary to modify our search procedure to handle both maximizing and minimizing players.
- It is also necessary to modify the branch-and-bound strategy to include two bounds, one for each of the players. This modified strategy is called "**alpha-beta pruning"**. It requires the maintenance of two threshold values:
  - a lower bound on the value that a maximizing node may ultimately be assigned called **alpha.**
  - an upper bound on the value that a minimizing node may be assigned called **beta.**



- After examining node F, we know that the opponent is guaranteed a score of —5 or less at C (since the opponent is the minimizing player). But we also know that we are guaranteed a score of 3 or greater at node A, which we can achieve if we move to B.
- Any other move that produces a score of less than 3 is worse than the move to B, and we can ignore it.
- After examining only F, we are sure that a move to C is worse (it will be less than or equal to —5) regardless of the score of node G. Thus we need not bother to explore node G at all.



- **Alpha Value:** In searching the tree, the entire sub tree headed by B is searched, and we discover that at A we can expect a score of at least 3. When this alpha value is passed down to F, it will enable us to skip the exploration of L.
- The reason is as follows: After K is examined; we see that I is guaranteed a maximum score of 0, which means that F is guaranteed a minimum of 0. But this is less than alpha's value of 3, no more branches of I need be considered.
- The maximizing player already knows not to choose to move to C and then to I since, if that move is made, the resulting score will be no better than 0 and a score of 3 can be achieved by moving to B instead. Nov,
- **Beta Value:** After cutting off further exploration of I, J is examined yielding a value of 5, which is assigned as the value of F. This value becomes the value of beta at node C. It indicates that C is guaranteed to get a 5 or less.
- Now, we must expand G. First M is examined and it has a value of 7, which is passed back to G as its tentative value. But now 7 is compared to beta (5). It is greater, and the player whose turn it is at node C is trying to minimize. So this player will not choose G, which would lead to a score of at least 7, since there is in alternative move to F, which will lead to a score of 5. Thus it is not necessary to explore any of the other branches of G.
- The function **MINIMAX-A-B**, which requires four arguments:
  Position, Depth, Use-Thresh, and Pass-Thresh.
- The initial call, to choose a move for PLAYER-ONE from the position CURRENT should be

  MINIMAX-A-B(CURRENT
  0,
  PLAYER-ONE,
  maximum value STATIC can compute
  minimum value STATIC can compute)

  The initial values for Use-Thresh and Pass-Thresh represent the worst values that each side could achieve.

Algorithm: MINIMAX-A-B((Position, Depth, Player, Use-Thresh, Pass-Thresh)

1. If *DEEP-ENOUGH (Position, Depth) then* return the structure.
   VALUE= STATIC (Position, Player);
   PATH = nil

   This indicates that there is no path from this node and that its value is that determined by the static evaluation function,

2. Otherwise, generate one more ply of the tree by calling the function **MOVEGEN (Position, Player**) and setting SUCCESSORS to the list it returns.

3. If SUCCESSORS is empty, then there are no moves to be made, so return the same structure that would have been returned it DEEP-ENOUGH had returned true.

4. If SUCCESSORS is not empty, then examine each element in turn and keep track of the best one. This *is* done as follows.

Initialize BEST-SCORE to the minimum value that STATIC can return It will be updated in reflect the best score that can be achieved by an element of SUCCESSORS**.**

For each element *SUCC* of SUCCESSORS, do the following:

(a) Set RESULT-SUCC to

MINIMAX-A-B (SUCC, Depth + 1, OPPOSITE(Player),
-Pass-Thresh, -Use-Thresh).

(b) Set NEW-VALUE to —VALUE (RESULT-SUCC).

(c) If NEW-VALUE > Pass-Thresh, then we have found a successor that is better than any that have been examined so far. Record this by doing the following:

i. Set Pass- Thresh to NEW-VALUE.

ii. The best known path is now from CURRENT to SUCC and then on to the appropriate path from SUCC as determined by the recursive call to MINIMAX-A-B. So set BEST-PATH to the result of attaching SUCC to the front of PATH (RESULT-SUCC).

(d) If Pass-Thresh (reflecting the current best value) is not better than Use-Thresh, then we should stop examining this branch. But both thresholds and values have been inverted. So if Pass-Thresh >= (Use-Thresh, then return immediately with the value

VALUE = Pass-Thresh
PATH = BEST-PATH

5. Return the structure:

VALUE Pass-Thresh
PATH = BEST-PATH

UNIT-I
Assignment-Cum-Tutorial Questions
SECTION-A

Objective Questions

1. Define the term "state space".
2. Define the term "Operationalization".
3. *A*_____*is* a technique that improves the efficiency of a search process. [ ]

   (a) Heuristic (b) Control Strategy (c) GPS (d) none
4. List the steps for solving the problem.
5. In Theorem Proving, the solution steps can be_____. [ ]

   (a) Ignored (b) Recoverable (c) Irrecoverable (d) none
6. Problems in which solution steps cannot be undone are _ _ [ ]

   (a) Ignored (b) Recoverable (c) Irrecoverable (d) none
7. Travelling Salesman problem is an example of_____. [ ]

   (a) Best-Path (b) Any-Path (c) Both (d) none
8. _____are the problems in which the computer is given a problem description and produces an answer with no intermediate communication. [ ]

   (a) Conversational (b) Solitary (c) Ignorable (d) None
9. _____is the list of nodes that have been generated and have had the heuristic function applied to them but which have not yet been examined. [ ]

   (a) OPEN (b) CLOSED (c) NODES (d) none
10. The function _ is a measure of the cost of getting from the initial state to the current node. [ ]

    (a) f^1 (b) g (c) h^1 (d) none
11. _____is useful for representing the solution of problems that can be solved by decomposing them into a set of smaller problems, all of which must then he solved. [ ]

    (a) OR graph (b) AND graph (c) AND-OR graph (d) none
12. _____ function uses whatever information it has to evaluate so that the best next move can be chosen. [ ]

(a) static evaluation (b) dynamic evaluation

(c) Threshold evaluation (d) none

13. DEEP-ENOUGH will take two parameters:_____and _____.
14. MINIMAX returns a structure containing both results:_____and _____.
15. A lower bound on the value that a maximizing node may ultimately be assigned called _____. [ ]
(a) alpha (b) beta (c) gamma (d) none
16. An upper bound on the value that a minimizing node may be assigned called _____. [ ]
(a) alpha (b) beta (c) gamma (d) none

SECTION-B

SUBJECTIVE QUESTIONS

1. List the steps for General Problem Solving.

2. Give an example of a problem for which breadth -first search would work better than depth-first search. Give an example of a problem for which depth-first search would work better than breadth-first search.

3. Describe the state space of water jug problem and also explain its solution.

4. List the steps necessary to provide a formal description of a problem.

5. What are the requirements of a control strategy? Develop an algorithm for:

(i) Breadth First Search (ii) Depth First Search

6. Summarize the advantages and disadvantages of control strategies.

7. Outline the factors that are necessary for analyzing a problem to choose most appropriate heuristic method.

8. Explain non-decomposable problem with suitable example.

9. Distinguish between Ignorable, Recoverable and Irrecoverable problems with necessary examples.

10. Distinguish between:

(i) Certain Outcome Vs Uncertain Outcome problems

(ii) Best path and Any-path problems

(iii)Problems whose solution is a state and whose solution is a path to state.

(iv) Solitary Vs Conversational problems

11. Explain an algorithm for Best-first Search.
12. Explain Problem Reduction using AND-OR graph with an algorithm.
13. Explain AO* algorithm.
14. Explain the procedure of Constraint Satisfaction with an example.
15. Explain Minimax procedure with an example.
16. Explain Alpha-Beta Pruning with an example.
17. Solve the Criptarithmetic problem using Constraint Satisfaction:

```
  CROSS
  ROADS
 _______
 DANGER
 _______
```