unit-1

Definition: Artificial intelligence (AI), also known as machine intelligence, that focuses on building and managing technology that can learn to autonomously make decisions and carry out actions on behalf of a human being.

The task domains of AI are

The domain of AI is classified into different task categories such as **Mundane/General tasks, Formal tasks and Expert tasks.**

Mundane tasks are the ones that we(the humans) do on regular basis without any special training such as computer vision, speech recognition, Natural language processing, generation and translation etc. Common sense, reasoning and planning are the common characteristics of these tasks.

Formal tasks - are the ones where there is an application of formal logic, some learning etc. Verifications, Theorom proving etc are the common characteristics. Games such as Chess Checkers, Go etc are classified in these task.

Then there are

expert task which comes under functional expert domain such as engineering, fault finding, manufacturing planning, medical diagnosis etc.

Expert tasks- Tasks which requires high analytical and thinking skills, a job only a professionals can do.

Task Domains of Artificial Intelligence		
Mundane (Ordinary) Tasks	**Formal Tasks**	**Expert Tasks**
Perception • Computer Vision • Speech, Voice	• Mathematics • Geometry • Logic • Integration and Differentiation	• Engineering • Fault Finding • Manufacturing • Monitoring
Natural Language Processing • Understanding • Language Generation • Language Translation	Games • Go • Chess (Deep Blue) • Ckeckers	Scientific Analysis
Common Sense	Verification	Financial Analysis
Reasoning	Theorem Proving	Medical Diagnosis
Planing		Creativity
Robotics • Locomotive		

Tic Tac Toe

The game Tic Tac Toe is also known as Noughts and Crosses or Xs and Os ,the player needs to take turns marking the spaces in a 3x3 grid with their own marks,if 3 consecutive marks (Horizontal, Vertical,Diagonal) are formed then the player who owns these moves get won.

There are 4 programs each program increase in

- Their complexity
- Use of generalizations
- Clarity of Knowledge
- The extensibility of their approach. Thus, they move toward being representations of what we call AI techniques.

Program 1 – blind method or Random method

Anine-element vector representing the board, where the elements of the vector correspond

to the board positions as follows:

1	2	3
4	5	6
7	8	9

An element contains the value 0 if the corresponding square is blank, 1 if it is filled with an X, or 2 if it is filled with an O.

Movetable A large vector of 19,683 elements (3^9), each element of which is a nine-element vector

Algorithm

To make a move, do the following:

1. View the vector Board as a ternary (base three) number. Convert it to a decimal number.

2. Use the number computed in step 1 as an index into Movetable and access the vector stored there.

3. The vector selected in step 2 represents the way the board will look after the move that should be made.

So set Board equal to that vector.

Comments

This program is very efficient in terms of time. And, in theory, it could play an optimal game of tic-tac-toe.

But it has several disadvantages:

- Takes lot of space
- Should specify entries in moveTable
- Movetable entries are determined and entered without any errors

Program 2– Numeric method

Board

A nine clement vector representing the board, as described for Program 1. But instead of using the numbers 0,1, or 2 in each clement, we store 2 (indicating blank), 3 (indicating X), or 5 (indicating O).

Turn

An integer indicating which move of the game is about to be played; 1 indicates the first move, 9 the last

The Algorithm

The main algorithm uses three subprocedures:

Make 2

Retums 5 if the center square of the board is blank, that is, if Board[5] = 2. Otherwise, this function returns any blank non corner square (2, 4, 6, or 8).

Posswin(p)

Returns 0 if player p cannot win on his next move; otherwise, it returns the number of the square that constitutes a winning move. This function will enable the program both towin and to block the opponent's win. Posswin operates by checking, one at a time, eachof the rows, columns, and diagonals. Because of the way values are numbered, it can testan entire row (column or diagonal) to see if it is a possible win by multiplying the valuesof its squares together. If the product is 18 (3 x 3x 2), then X can win. If the product is 50(5 x 5 x 2), then O can win. If we find a winning row, we determine which element isblank, and return the number of that square.

Go(n)

Makes a move in square n. This procedure sets Board[n] to 3 if Turn is odd, or 5 if Turn is even. It also increments Turn by one.

Program 3 – Even-odd algorithm

This program is identical to Program 2 except for one change in the representation of the board. We againrepresent the board as a nine-element vector, but this time we assign board positions to vector elements as follows:

8	3	4
1	5	9
6	7	2

Notice that this numbering of the board produces a magic square: all the rows, columns, and diagonals sumup to 15. This means that we can simplify the process of checking for a possible win. In addition to markingthe board as moves are made, we keep a list, for each player, of the squares in which he or she has played. To check for a possible win for one player, we consider each pair of squares owned by that player and computethe difference between 15 and the sum of the two squares. If this difference is not positive or if it is greaterthan 9, then the original two squares were not collinear and so can be ignored. Otherwise, if the squarerepresenting the difference is blank, a move there will produce a win. Since no player can have more than foursquares at a time, there will be many fewer squares examined using this scheme than there were using themore straightforward approach of Program 2. This shows how the choice of representation can have a majorimpact on the efficiency of a problem-solving program.

Program 4 – Min Max algorithm

Board Position

A structure containing a nine-element vector representing the board, a list of board positions that could result from the next move, and a number representing an estimate of howlikely the board position is to lead to an ultimate win for the player to move.

The Algorithm

To decide on the next move, look ahead at the board positions that result from each possible move. Decidewhich position is best (as described below), make the move that leads to that position, and assign the rating of that best move to the current position.

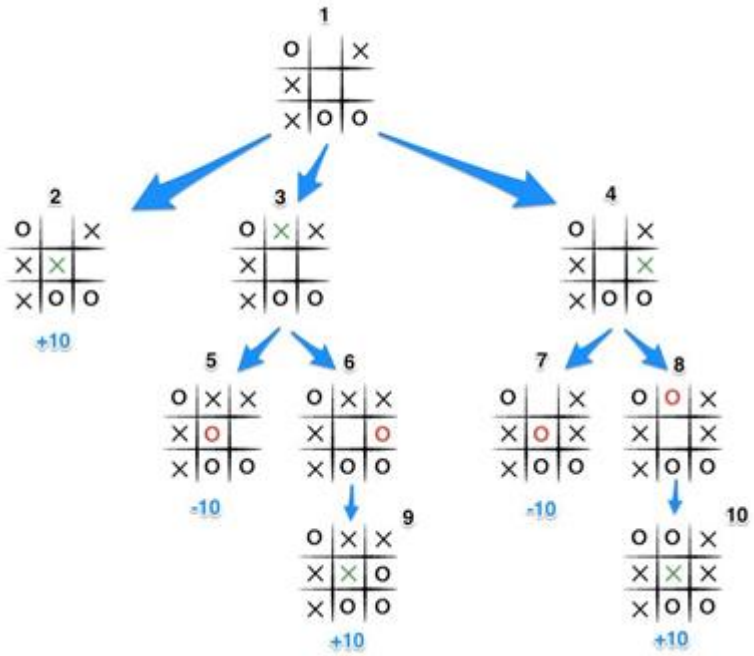
To decide which of a set of board positions is best, do the following for each of them:

1. See if it is a win. If so, call it the best by giving it the highest possible rating.

2. Otherwise, consider all the moves the opponent could make next. See which of them is worst for us (by recursively calling this procedure). Assume the opponent will make that move. Whatever rating thatmove has, assign it to the node we are considering.

3. The best node is then the one with the highest rating.

This algorithm will look ahead at various sequences of moves in order to find a sequence that leads to awin. It attempts to maximize the likelihood of winning, while assuming that the opponent will try to minimize that likelihood.



Criteria For Success

One of the most important questions to answer in any scientific or engineering research project is "How will we know if we have succeeded?",Artificial Intelligence too is not an exception. The questions arised are as follows:

"How will we know if we have constructed a machine that is intelligent?"

"What is intelligence?"etc..

But can we do anything to measure our progress....?

In 1950, Alan Turing proposed the following method for determining whether machine can think. His method was known as "Turing Test". To conduct this test, we need two people and the machine to be evaluated. One person plays the role of the interrogator, who is in a separate room from the computer and the other person. The interrogator can ask questions to either the computer or the person by typing questions and receiving the typed responses. However, the interrogator knows them only as A and B and aims to determine, which is the person and which is the machine. The goal of the machine is to fool the interrogator.

So, for example, if the interrogator asked the question "How much is 12,324 times 73,981?" it could wait several minutes and then respond with the wrong answer [Turing, 1963].

The more serious issue is the amount of knowledge that a machine would need to pass the Turing test. It will be a long time before the computer passes the Turing test. Some people believe none ever will. Can we measure the achievement of AI in more restricted domains?

Often the answer to this question is "yes". Sometimes, it is possible to get a fairly precise measure of the achievement of a program. For example, a program can acquire a chess rating in the same way as a human player. The rating is based on the ratings of players whom the program can beat. Already programs have acquired chess ratings higher than majority of human players. For other domains, a less precise measure of a program's achievement is possible. For example, DENDRAL is a program that analyzes organic compounds to determine their structure. It is hard to get a precise measure of DENDRAL's level of achievement compared to human chemists, but it has produced analyzes that have been published as original research results. Thus it is certainly performing competently.

In other technical domains, it is possible to compare time as programs typically require minutes to perform tasks for which previously required hours of a skilled engineer's time.

If our goal in writing a program is to stimulate human performance at a task, then the measure of success is the extent to which the program's behaviour corresponds to that performance. In this we do not simply want a program that does all possible tasks. We want the one that fails when people do. Various techniques developed by psychologists for comparing individuals and for testing models can be used to do this analysis.

We are forced to conclude that the question of whether a machine has intelligence or can think is too difficult to answer precisely. But it is often possible to construct a computer program that meets some performance standard for a particular task. That does not mean that the program does the task in a best possible way. It means only that we understand at least one way of doing a part of a task. When we set out to design an AI program, we should attempt to specify the possible criteria for success for that particular program functioning in its restricted domain. For the moment, that is the best we can do.

2.1 Problem as a state space search

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

- Define the problem precisely. This definition must include precise specifications of what the initial situations will be as well as what final situations constitute acceptable solutions to the problem.
- Analyse the problem. A few very important features can have an immense impact on the appropriateness of various possible techniques for solving the problem.
- Isolate and represent the task knowledge that is necessary to solve the problem.
- Choose the best problem-solving techniques and apply them to the particular problem.

Defining the problem as a state space search

Two problems are defined under this technique

1. **Chess Game**

2. **Water Jug Problem**

CHESS GAME:

To build a problem that could "Play Chess," we would first have to specify the starting position of the chess board, the rules that define the legal moves, and the board positions that represent a win for one side or the other. In addition, we must make explicit the previously implicit goal of not only playing a legal game of chess but also winning the game, if possible.

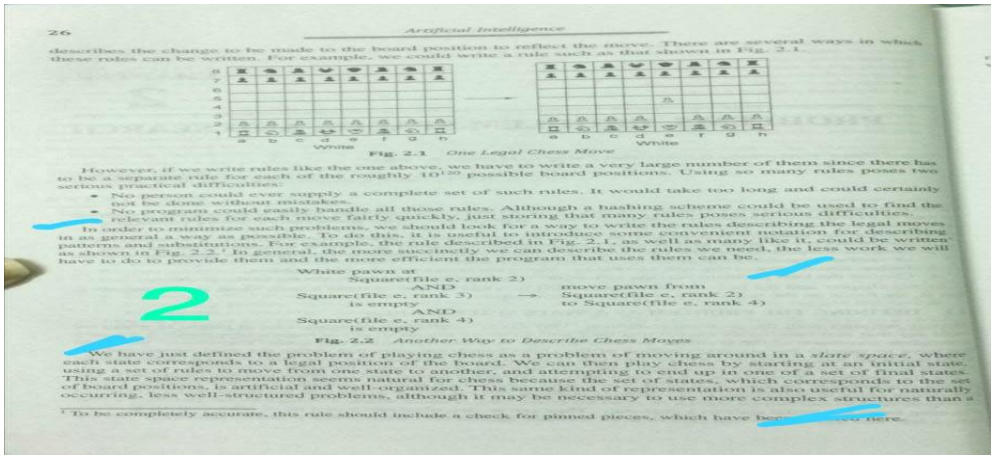
For the problem "play chess," it is fairly easy to provide a formal and complete problem description. The starting position can be described as an 8x8 array where each position contains a symbol standing for the appropriate piece in the official chess opening position. We can define as our goal any board 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 be described easily as a set of rules consisting of two parts: a left side that serves as a pattern to be matched against the current board position and a right side that describes the change to be made to the board position to reflect the move. There are several ways in which these rules can be written.

For example, we could write a rule such as that shown,

However, if we write rules like the one above, we have to write a very large number of them, since there has to be a separate rule for each of the roughly 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 a hashing scheme could be used to find the relevant rules for each move fairly quickly, just storing that many rules posses serious difficulties.

In order to minimise such problems , we should look to write the rules describing the legal moves in a general way as possible. To do this, we have to give some convenient notations for describing patterns and substitutions. By doing like this , the less work we will have to do to provide them and more efficient the program.



We have just defined the problem of playing chess as a problem of moving around in a state space, where each state corresponds to a legal position of the board. We can then play chess by starting at an initial state. Using a set of rules to move from one state to another and attempting to end up in one set of final states. The state space representation forms the basis of most of the AI methods.

Its structure corresponds to the structure of problem solving in two important ways:

- It allows for a formal definition of a problem as the need to convert some given situation inti some desired situation using a set of permissible operations.
- It permits us to define the process of solving particular problem as a combination of known techniques 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.

2. WATER JUG PROBLEM:

You are given two jugs , a 4-gallon one and a 3-gallon one. Neither has any measuring markers on it. There is a pump that can be used to fill the jugs with water. How can you get exactly 2 gallons of water into the 4-gallon jug?

- The state space for this problem can be described as the set of ordered pairs of integers(x,y), such that x=0,1,2,3, or 4 and y=0,1,2, or 3;
- x represents the number of gallons of water in the 4-gallon jug, and
- y represents the quantity of water in the 3-gallon jug.
- The start state(0,0). The goal state is (2,n) for any value of n (since the problem does not specify how many gallons need to be in the 3-gallon jug).

As in the chess problem, they are represented as rules whose left sides are matched aganist the current state and whose right sides describe the new state that results from applying the rule. Notice that in order to describe the operators completely , it was necessary to amke explicit some assumptions not mentioned in the problem statement. We have assumed that we can fill a jug from the pump, that we can pour water out of a jug on to the ground, that we can pour water from one jug to another, and that there are no other measuring devices available. Additional assumptions such as these are almost always required when converting from a typical problem statement given in English to formal representation of the problem suitable for use by a problem.

To solve the water jug problem, all we need, in addition to the problem description given above , 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 if it 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.

For the water jug problem , as with many others, there as several sequences of operators that solve the problem.

Often, a problem contains the explicit or implied statement that the shortest or cheapest such sequence be found. If present, this requirement will have a significant effect on the choice of an appropriate mechanism to guide the search for a solution.

For example, the first rule says, “If the 4-gallon jug is not already full, fill it.”

This rule could, however, have been written as, “Fill the 4-gallon jug,” since it is physically possible to fill the jug even if it is already full.

It is stupid to do so since no change in the problem state results, but it is possible. By encoding in the left sides of the rules constraints that are not strictly necessary but that restrict the application of the rules to states in which the rules are most likely to lead to a solution, we can generally increase the efficiency of the problem-solving program that uses the rules.

Production Rules for the water Jug Problem:

- (x , y) -> (4 , y) Fill the 4-gallon jug

  If $x < 4$

- (x , y) -> (x , 3) Fill the 3-gallon jug

  If $y < 3$

- (x , y) -> (x-d , y) Pour some water out of the 4-gallon jug

  If x > 0

- (x , y) ->(x, y-d) Pour some water out of the 3-gallon jug

  If y > 0

- (x , y) ->(0 , y) Empty the 4-gallon jug on the ground

  If x > 0

- (x,y) ->(x , 0) Empty the 3-gallon jug on the ground

  If y > 0

- (x, y) ->(4,y-(4-x)) Pour water from the 3-gallon jug into the

  If x+y≥4 and y>0 4- gallon jug until the 4-gallon jug is full

- (x , y) ->(x-(3-y) , 3) Pour water from the 4-gallon jug into the

  If x+y≥3 and x > 0 3-gallon jug until the 3-gallon jug is full

- (x , y) ->(x+y , 0) Pour all the water from 3-gallon jug into

  If x+y≤4 and y>0 the 4-gallon jug

- (x , y) ->(0, x+y) Pour all the water from the 4-gallon jug

  If x+y≤3 and x>0 into the 3-gallon jug.

- (0 , 2) ->(2 , 0) pour the 2 gallons from the 3-gallon jug into the 4-gallon jug.

- (2 , y) ->(0 , y) Empty the 2 gallons in the 4-gallon jug on

  the ground

One Solution to the water jug problem:

Gallons in the 4-gallon jug	Gallons in the 3-gallon jug	Rule Applied
0	**0**	
0	**3**	**2**
3	**0**	**9**
3	**3**	**2**
4	**2**	**7**
0	**2**	**5 or 12**
2	**0**	**9 or 11**

- The statespace for this problem can be describes as the set of ordered pairs of integers (x, y; xϵ4 gallons jug and yϵ3 gallons jug)

  X=0,1,2,3,4

  Y=0,1,2,3

- The starting state of the 4-gallon jug and 3-gallon jug is (0,0)
- The goal state is (2,n) for any value og 'n' spedcified to 3-gallon jug.

Formal Description of the problem:

- Define a state space that contains all the possible configurations of the relevany objects, it is possible to define this space without explicitly enumerating all of the states it contains.
- Specify one or more states within that space is described possible situations from which the problem solving process may start. These states are called as initial states.
- Specify one or more states that wouls be acceptable as solutions to the problem. These states are called as goal states(or) final states.
- Specify a set of rules that desxribes the actions (operators) available. Doing this will reguire giving to the following issues.
  - What unstated assumptions are present in the informal problem description?
  - How general should the rule be?
  - How much of the work required to solve the problem should be pre computed and representedin the rules?

2.2. Production Systems

Production systems provide structures for describing and performing the search process. Do not get confused by the other uses of the word production, such as to describe what is done in factories.

A production system consists of:

• A set of rules, each consisting of a left side (pattern) that determines the applicability of the rule and a right side that describes the operation to be performed if the rule is applied.

• One or more knowledge/ databases that contain whatever information is appropriate for the particular task. Some parts of the database may be permanent, while other parts of it may pertain only to the solution if the current problem. The information in these databases may be structured in any appropriate way.

• A control strategy that specifies the order in which the rules will be compared to the database and the way if resolving the conflicts that arise when several rules match at once.

• A rule applier.

So far, our definition of a production system has been very general. It encompasses a great many systems, including our descriptions of both a chess player and a water jug problem solver. It also encompasses a family of general production system interpreters, including:

- Basic production system languages, such as OPS5 [Brownston et al., 1985] and ACT*[Anderson, 1983].
- More Complex, often hybrid systems called expert system shells, which provide complete environment for construction of knowledge based expert systems.
- General problem-solving architectures like SOAR [Laird et al., 1987], a system based on a specific set of cognitively motivated hypothesis about the nature of problem solving.

All of these systems provide the overall architecture of a production system and allow the programmer to write rules that define particular problems to be solved.

Now, we got to know that in order to solve a problem, we must first reduce it to one for which a precise statement can be given. This can be done by defining the problem's state space (including the start and goal states) and a set of operators for moving in that space. The problem can then be solved by searching for a path through the space from an initial state to a goal state which can be usefully modelled as a production system.

2.2.1 Control Strategies

Control Strategy is a technique or strategy, tells us about which rule has to be applied next while searching for the solution of a problem within problem space. A good control strategy is always required to decide which rule need to be applied during the process of searching for a solution to a problem. It helps us to decide which rule has to apply next without getting

stuck at any point. These rules decide the way we approach the problem and how quickly it is solved and even whether a problem is finally solved.

There are two requirements for a good control strategy:

Control Strategy should cause Motion

Each rule or strategy applied should cause the motion because if there will be no motion than such control strategy will never lead to a solution. Motion states about the change of state and if a state will not change then there be no movement from an initial state and we would never solve the problem.

Control strategy should be Systematic

Though the strategy applied should create the motion but if do not follow some systematic strategy than we are likely to reach the same state number of times before reaching the solution which increases the number of steps. Taking care of only first strategy we may go through particular useless sequences of operators several times. Control Strategy should be systematic implies a need for global motion (over the course of several steps) as well as for local motion (over the course of single step).

Breadth-First Search:

It searches along the breadth and follows first-in-first-out queue data structure approach.

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 UST 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.

i. If the new state is a goal state, quit and return this state.

ii. Otherwise, add the new state to the end of NODE-LIST.

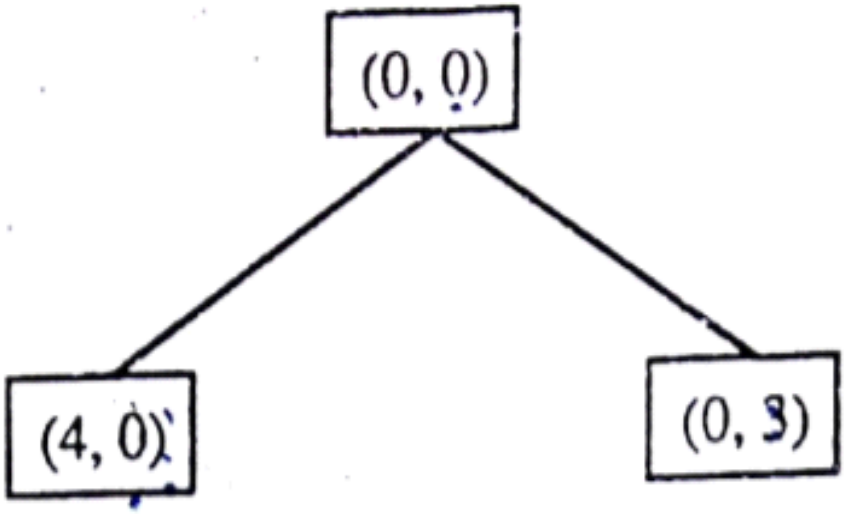


Figure 2.5: One Level of a Breadth-First Search Tree

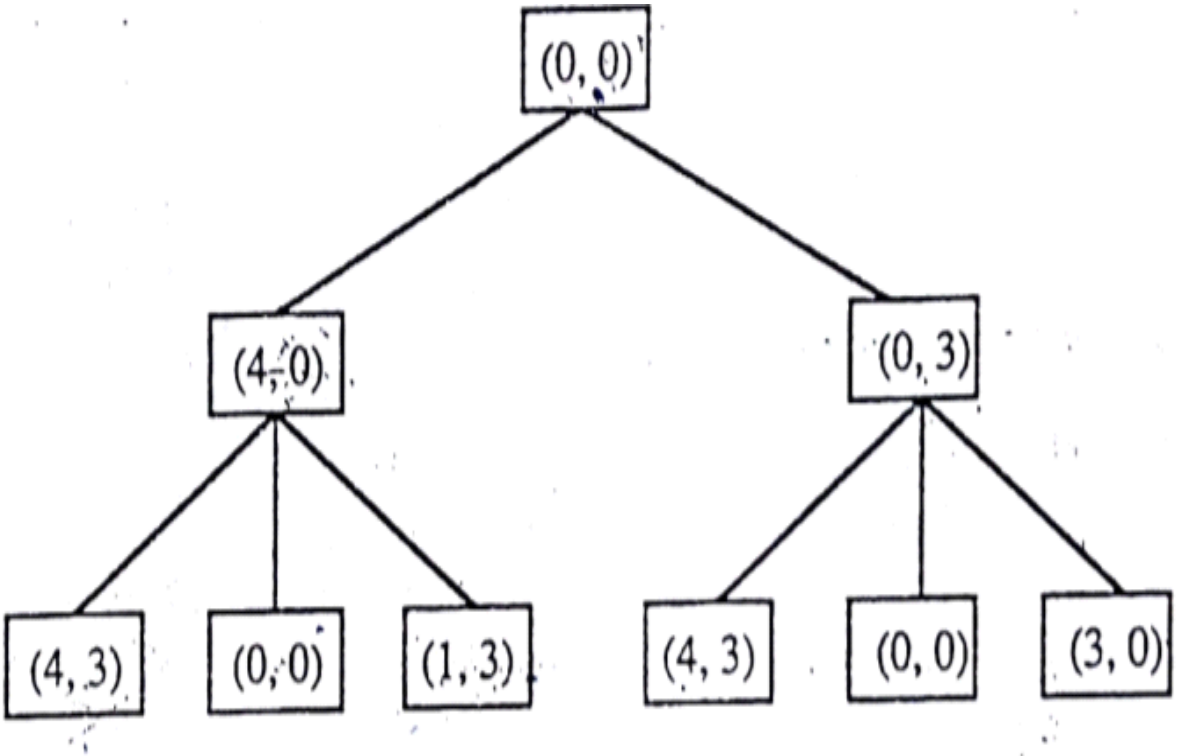


Figure 2.6: Two Levels of a Breadth-First Search Tree

Advantages of BFS

- Breadth-first search will not get trapped exploring a blind alley. This contrasts with depth-first searching, which may follow a single, unfruitful path for a very long time, perhaps forever, before the path actually terminates in a state that has no successors.

- If there is no solution, then breadth-first search is guaranteed to find it. If there are multiple solutions, then a minimal solution will be found.

Other systematic control strategies are also available. For example, 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 be created. This form of backtracking is called chronological backtracking because the order in which steps are undone depends only on the temporal sequence in which the steps were originally made. Specifically, the most recent step is always the first to be undone. This form of backtracking is what is usually meant by the simple term backtracking.

The search procedure we have just described is also called depth-first search. The following algorithm describes this precisely.

Depth-First Search

It searches along the depth and follows the stack approach.

Algorithm: Depth-First Search

1. If the initial state it 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. If there are no more succession signal failure,
   (b) Call Depth-First Search with E as the initial state.
   (c) I success is returned, signal success. Otherwise continue in this loop.

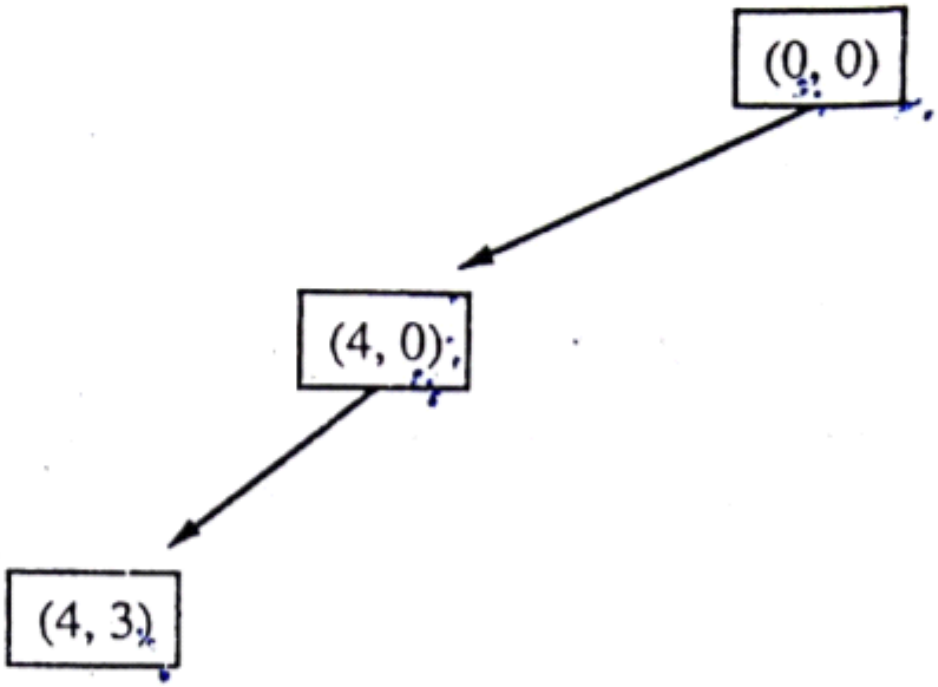


Figure 2.7: A Depth-First Search Tree

The above figure, shows a snapshot of a depth-first search for the water jug problem. A comparison of these two simple methods produces the following observations.

Advantages of DFS

- Depth-First search requires less memory since only the nodes on the current path are stores.
- Depth-First search may find the solution without examining much of the search space. The search stops once a solution is found.

For the water jug problem, most control strategies that cause motion and are systematic will lead to an answer. The problem is simple. But this is not always the case. In order to solve some problems during our lifetime, we must also demand a control structure that is efficient.

Consider the following problem.

The Traveling Salesman Problem:

A salesman has a list of cities, each of which he must visit exactly once. There are direct roads between each pair of cities on the list. Find the route the salesman should follow for the shortest possible round trip that both starts and finishes at any one of the cities.

A simple, motion causing and systematic control structure could be used to solve this problem. It would simply explore all possible paths in the tree and return the one with the shortest length. This approach will even work for very short lists of cities. But it breaks down quickly as the number of cities grows. If there are N cities, then the number of different paths among them is 1, 2…(N- 1), or (N-1)!. The time to examine a single path is proportional to N and the total time required to perform this search is proportional to N!.

Assuming there are only 10 cities, 10! is 3628800 which is a very large number. The salesman could easily have 25 cities to visit. To solve this problem would take more time than he would be willing to spend. This phenomenon is called combinatorial explosion. To do it, we need a new control strategy.

We can beat the simple strategy outlined above using a technique called **branch-and-bound.** Begin generating complete paths, keep track of the shortest path found so far. Give up exploring any path as soon as its partial length becomes greater than the shortest path found so far. Using this technique, we are guaranteed to find the shortest path. Unfortunately, although this algorithm is more efficient than the first one, it still requires exponential time. The exact amount of time it saves for a particular problem depends on the order in which the paths are explored. But it is still inadequate for solving large problems.

2.2.2 Heuristic Search

Heuristic is a technique that improves the efficiency of a search process. It expands nodes in the order of their heuristic values. It creates two lists, a closed list for the already expanded nodes and an open list for the created but unexpanded nodes. In each iteration, a node with a minimum heuristic value is expanded, all its child nodes are created and placed in the closed list. Then, the heuristic function is applied to the child nodes and they are placed in the open

list according to their heuristic value. The shorter paths are saved and the longer ones are disposed.

Using good general purpose or special purpose heuristics, we can get good (though possibly nonoptimal) solutions to hard problems, such as the traveling salesman, in less than exponential time.

One example of good general purpose heuristic that is useful for a variety of combinatorial problems is the nearest neighbour heuristic, which works by selecting the Totally superior alternative at each step. Applying it to the traveling salesman problem, we can procedure executes in time proportional to N^2, an improvement from N! using the following procedure:

- Arbitrarily select a starting city.
- To select the next city, look at all cities not yet visited, and select the one closest to the current city. Go to it next.
- Repeat step 2 until all cities have been visited.

A **heuristic function** is one which maps from problem state descriptions to measures of desirability, which is usually represented in terms of numerical values.

- Well defined heuristic functions can play an important part in efficiently guiding a search process towards a solution.
- It is not a problem the way in which the function is stated.
- The program that uses the values can minimize or maximize it as required.
- The purpose of heuristic function is to guide the process of searching in a most profitable direction.

Some Simple Heuristic Functions:

- Chess - the material advantage of our side over the opponent.
- Travelling Salesman - the sum of the distances so far.
- Tic-Tac-Toe - 1 for each row in which we could win and in which we already have one piece plus 2 for each such row in which we have two pieces.

PROBLEM CHARACTERISTICS

Heuristics cannot be generalized, as they are domain specific. Most problems requiring simulation of intelligence use heuristic search extensively. Some heuristics are used to define the control structure that guides the search process. But heuristics can also be encoded in the rules to represent the domain knowledge. Since most AI problems make use of knowledge and guided search through the knowledge, AI can be described as the study of techniques for solving exponentially hard problems in polynomial time by exploiting knowledge about problem domain.

To use the heuristic search for problem solving, we should focus on the analysis of the problem for the following considerations:

- Is the problem decomposable into set of sub problems?
- Can the solution step be ignored or undone?
- Is the problem universally predictable?
- Is a good solution to the problem obvious without comparison to all the possible solutions?
- Is the desire solution a state of world or a path to a state?
- Is a large amount of knowledge absolutely required to solve the problem?
- Will the solution of the problem required interaction between the computer and the person?

  The general classes of engineering problems such as planning, classification, diagnosis, monitoring and design are generally knowledge intensive and use a large amount of heuristics. Depending on the type of problem, the knowledge representation schemes and control strategies for search are to be adopted.

1.Problem Decomposition

Suppose to solve the expression is: + $(X^3 + X^2 + 2X + 3\sin x)\, dx$

This problem can be solved by breaking it into smaller problems, each of which we can solve by using a small collection of specific rules. Using this technique of problem decomposition, we can solve very large problems very easily. This can be considered as an intelligent behaviour.

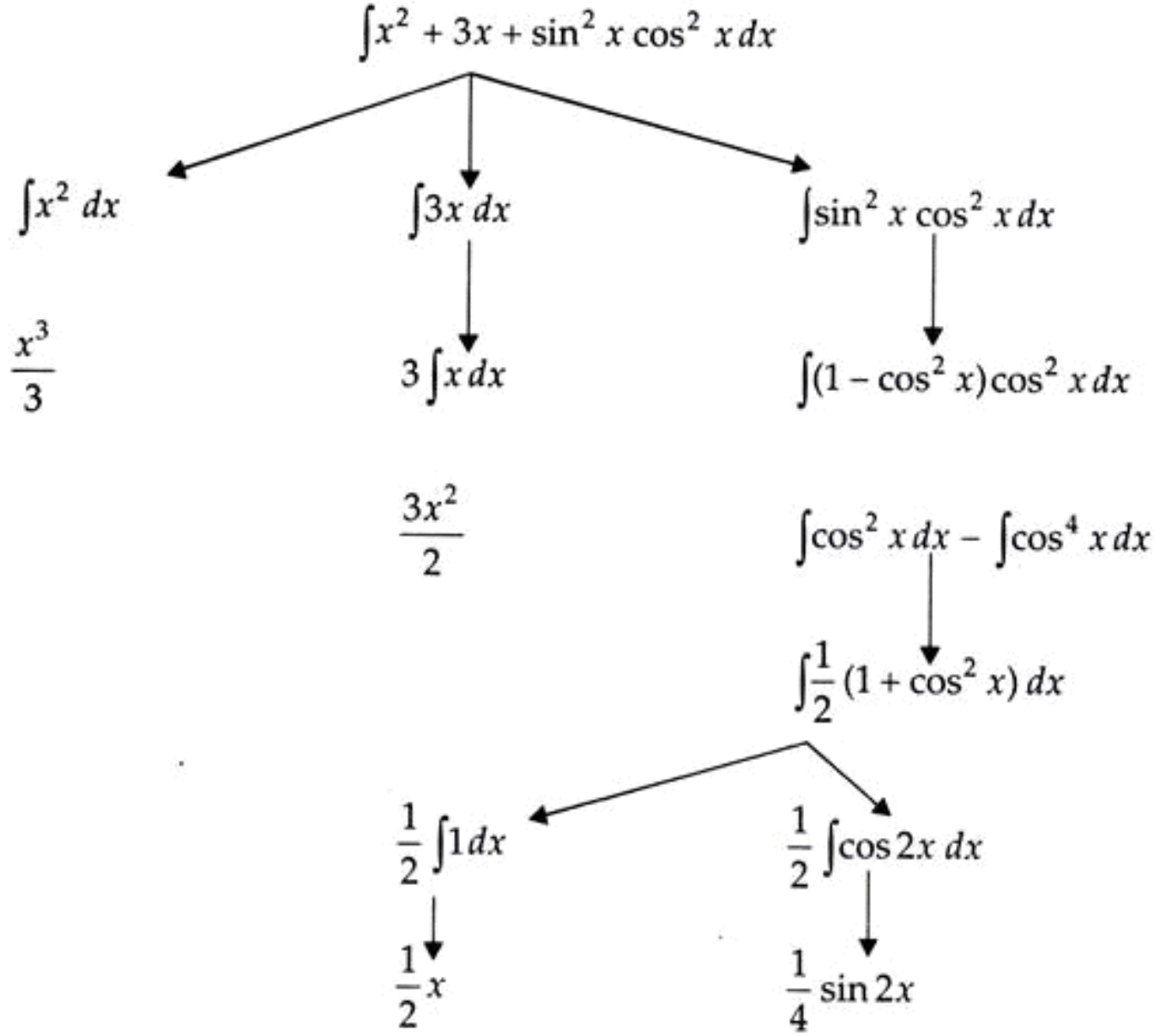


Fig. 3.1.

Proposed solution for a mathematical problem

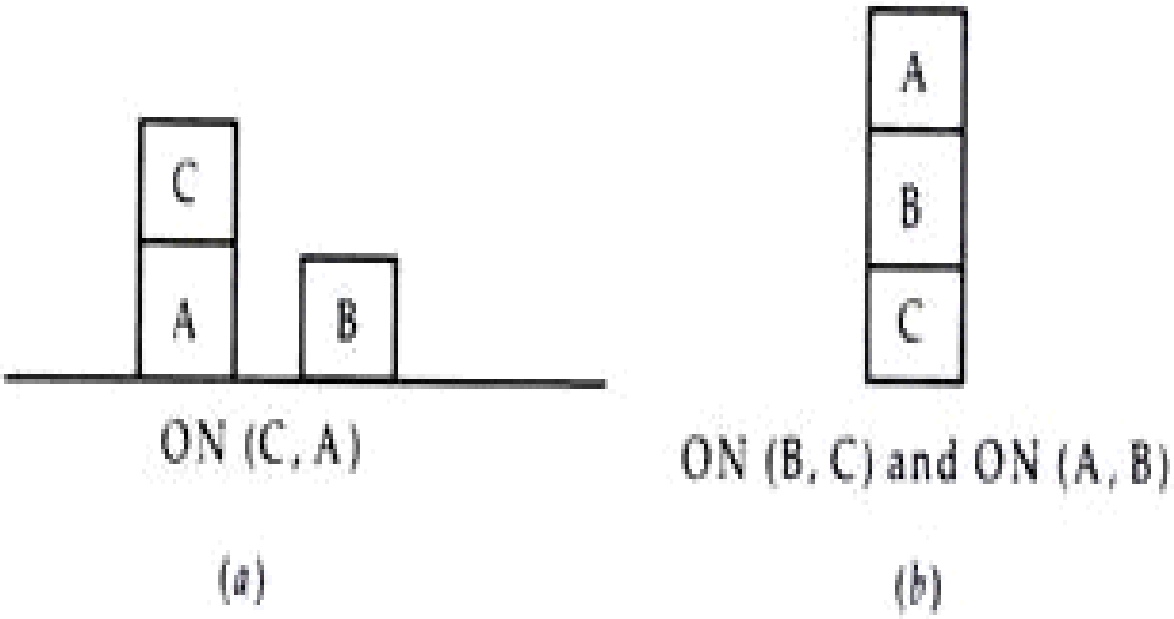


Fig. 3.2. *A block world problem.*

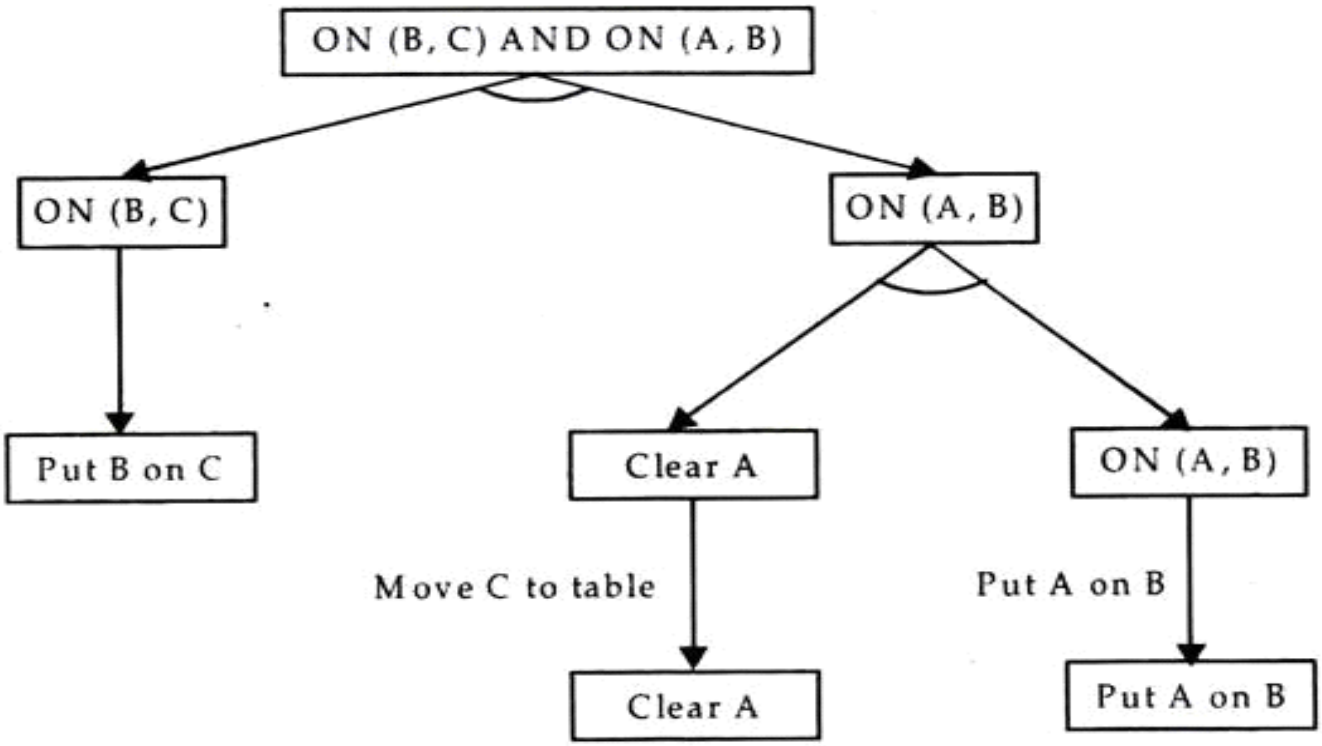


Fig. 3.3. *One Solution for the above problem.*

Proposed solution for the blocks problem

2.Can Solution Steps be Ignored?

Suppose we are trying to prove a mathematical theorem: first we proceed considering that proving a lemma will be useful. Later we realize that it is not at all useful. We start with another one to prove the theorem. Here we simply ignore the first method. Consider the 8-puzzle problem to solve: we make a wrong move and realize that mistake. But here, the control strategy must keep track of all the moves, so that we can backtrack to the initial state and start with some new move. Consider the problem of playing chess. Here, once we make a move we never recover from that step.

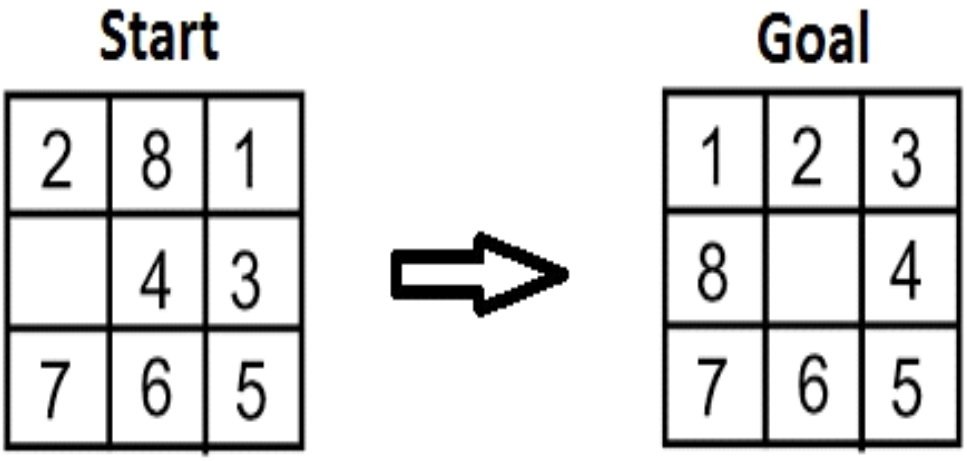


Figure : 8 Puzzle Problem

These problems are illustrated in the three important classes of problems mentioned below:

- Ignorable, in which solution steps can be ignored.

  Eg: Theorem Proving

- Recoverable, in which solution steps can be undone.

  Eg: 8-Puzzle

- Irrecoverable, in which solution steps cannot be undone.

  Eg: Chess

3.Is the Problem Universe Predictable?

Consider the 8-Puzzle problem. 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 what the resulting state will be. We can backtrack to earlier moves if they prove unwise. Suppose we want to play Bridge. We need to plan before the first play, but we cannot play with certainty. So, the outcome of this game is very uncertain. In case of 8-Puzzle, the outcome is very certain. To solve uncertain outcome problems, we follow the process of plan revision as the plan is carried out and the necessary feedback is provided. The disadvantage is that the planning in this case is often very expensive.

4.Is Good Solution Absolute or Relative?

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

- Siva was a man.

2. Siva was a worker in a company.

3. Siva was born in 1905.

4. All men are mortal.

5. All workers in a factory died when there was an accident in 1952.

6. No mortal lives longer than 100 years.

Suppose we ask a question: 'Is Siva alive?' By representing these facts in a formal language, such as predicate logic, and then using formal inference methods we can derive an answer to this question easily. There are two ways to answer the question shown below:

Method I:

1. Siva was a man.

2. Siva was born in 1905.

3. All men are mortal.

4. Now it is 2008, so Siva's age is 103 years.

5. No mortal lives longer than 100 years.

Method II:

1. Siva is a worker in the company.

2. All workers in the company died in 1952.

Answer: So Siva is not alive. It is the answer from the above methods.

If we follow one path successfully to the correct answer, then there is no reason to go back and check another path to lead the solution.

5.Is the Solution a State or a Path?

Consider the problem of finding a consistent interpretation for the lenience

The bank president ate a dish of pasta salad with the fork.

There are several components of this sentence, each of which, may have more than one interpretation. But the components must form whole, and so they constrain each other's interpretations. 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 pairs of nouns. For example, dog food does not normally contain dogs,

* The phrase "with the fork" could modify several parts of the sentence. In this case, it modifies the verb "eat". But if the phrase had been "with vegetables," then the modification structure would be different. And if the phrase had been "with her friends." the structure would be different still.

The two examples, natural language understanding and the water jug problem, illustrate the difference between problems whose solution is a state of the world and problems whose solution is a path to a state. At one level, this difference can be ignored and ail problems can be formulated as ones in which only a state is

required to he reported. If we do this for problems such as the water ug, then we must redescribe our states so that each state represents a partial path to a solution rather than just a single slate of the world.

6. What is the role of Knowledge?

Though one could have unlimited computing power, the size of the knowledge base available for solving the problem does matter in arriving at a good solution. Take for example the game of playing chess, just the rues for determining legal moves and some simple control mechanism is sufficient to arrive at a solution. But additional knowledge

about good strategy and tactics could help to constrain the search and speed up the execution of the program. The solution would then be realistic.

Consider the case of predicting the political trend. This would require an enormous amount of knowledge even to be able to recognize a solution , leave alone the best.

7.Will the solution of the problem required interaction between the computer and the person?

* Solitary, in which the computer is given & problem description and produces an answer with no intermediate communication and with no demand lor an explanation of the reasoning process

* Conversational, in which there is intermediate communication between a person andthe computer, either to provide additional assistance to the computer or to provide additional information to the user, or both.Of course, this distinction is not a strict one describing particular problem domains..

Mathematical theorem proving could be regarded as either. But for a particular application, one or the other of these types of systems will usually be desired and that decision will be important in solving method.

Problem Classification

When actual problems are examined front the point of view of all of these questions, it becomes apparent that there arc several broad classes into which die problems fall. These classes can each be associated with a generic control strategy that is appropriate for solving the problem.

Production system characteristics

The production system is a model of computation that can be applied to implement search algorithms and model human problem solving. Such problem solving knowledge can be packed up in the form of little quanta called productions. A production is a rule consisting of a situation recognition part and an action part. A production is a situation-action pair in which the left side is a list of things to watch for and the right side is a list of things to do so. When productions are used in deductive systems, the situation that trigger productions are specified combination of facts. The actions are restricted to being assertion of new facts deduced directly from the triggering combination. Production systems may be called premise conclusion pairs rather than situation action pair.

A production system consists of following components.

(a) A set of production rules, which are of the form A®B. Each rule consists of left hand side constituent that represent the current problem state and a right hand side that represent an output state. A rule is applicable if its left hand side matches with the current problem state.

(b) A database, which contains all the appropriate information for the particular task. Some part of the database may be permanent while some part of this may pertain only to the solution of the current problem.

(c) A control strategy that specifies order in which the rules will be compared to the database of rules and a way of resolving the conflicts that arise when several rules match simultaneously.

(d) A rule applier, which checks the capability of rule by matching the content state with the left hand

side of the rule and finds the appropriate rule from database of rules.

The important roles played by production systems include a powerful knowledge representation scheme. A production system not only represents knowledge but also action. It acts as a bridge between AI and expert systems. Production system provides a language in which the representation of expert knowledge is very natural. We can represent knowledge in a production system as a set of rules of the form

If (condition) THEN (condition)

along with a control system and a database. The control system serves as a rule interpreter and sequencer. The database acts as a context buffer, which records the conditions evaluated by the rules and information on which the rules act. The production rules are also known as condition – action, antecedent – consequent, pattern – action, situation – response, feedback – result pairs.

For example,

If (you have an exam tomorrow)

THEN (study the whole night)

The production system can be classified as monotonic, non-monotonic, partially commutative and commutative.

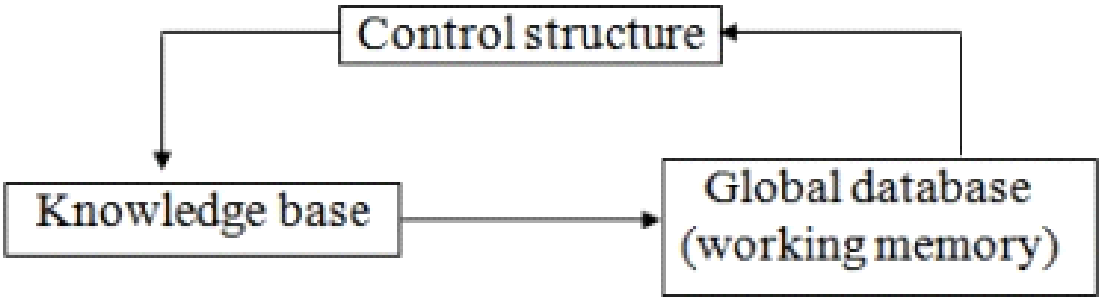


Figure Architecture of Production System

Features of Production System

Some of the main features of production system are:

Expressiveness and intuitiveness: In real world, many times situation comes like "i f this happen-you will do that", "if this is so-then this should happ en" and many more. The production rules essentially tell us what to do in a given situation.

1. **Simplicity:** The structure of each sentence in a production system is unique and uniform as they use "IF-THEN" structure. This structure provides simpli city in knowledge representation. This feature of production system improves the readability of production rules.

2. **Modularity:** This means production rule code the knowledge available in discrete pieces. Information can be treated as a collection of independent facts which may be added or deleted from the system with essentially no deletetious side effects.

3. **Modifiability:** This means the facility of modifying rules. It allows the development of production rules in a skeletal form first and then it is accurate to suit a specific application.

4. **Knowledge intensive:** The knowledge base of production system stores pure knowledge. This part does not contain any type of control or programming information. Each production rule is normally written as an English sentence; the problem of semantics is solved by the very structure of the representation.

Disadvantages of production system

1. **Opacity:** This problem is generated by the combination ofproduction rules. The opacity is generated because of less prioritization of rules. More priority to a rule has the less opacity.

2. **Inefficiency:** During execution of a program several rules may active. A well devised control strategy reduces this problem. As the rules of the production system are large in number and they are hardly written in hierarchical manner, it requires some forms of complex search through all the production rules for each cycle of control program.

3. **Absence of learning:** Rule based production systems do not store the result of the problem for future use. Hence, it does not exhibit any type of learning capabilities. So for each time for a particular problem, some new solutions may come.

4. **Conflict resolution:** The rules in a production system should not have any type of conflict operations. When a new rule is added to a database, it should ensure that it does not have any conflicts with the existing rules.