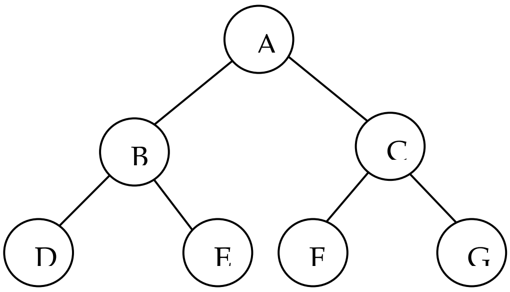
EXPERIMENT NO :

Date :

AIM : Implementation of Depth First Search.

PROBLEM STATEMENT : Write a program to traverse the following graph using DFS method. Consider A is initial state and F is the final state.



THEORY :

Depth First Search (DFS) searches deeper into the problem space. Depth-first search always generates successor of the deepest unexpanded node. It uses Last-In First-Out stack for keeping the unexpanded nodes. More commonly, depth-first search is implemented recursively, with the recursion stack taking the place of an explicit node stack. Depth first search works by taking a node, checking its neighbors, expanding the first node it finds among the neighbors, checking if that expanded node is our destination, and if not, continue exploring more nodes.

Algorithm: Depth First Search

i. Declare two empty lists: Open and Closed.

ii. Add Start node to Open list.

iii. While Open list is not empty, loop the following:

  a. Remove the first node from Open List.

  b. Check to see if the removed node is the destination.

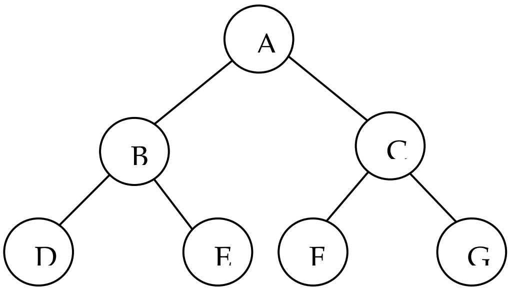
  i. If the removed node is the destination, break out of the loop, add the node to closed list, and return the value of Closed list.

   ii. If the removed node is not the destination, continue the loop (go to Step c).

c. Extract the neighbors of above removed node.

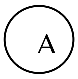
d. Add the neighbors to the *beginning* of Open list, and add the removed node to Closed list. Continue looping.

Example



In depth-first search, we start with the root node and completely explore the descendants of a node before exploring its siblings (and siblings are explored in a left-to-right fashion).

Step 1

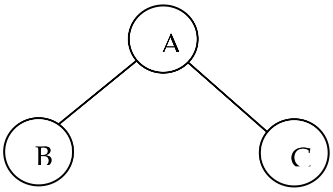


Depth-first traversal:

Open: A

Closed: <empty>

Step 2

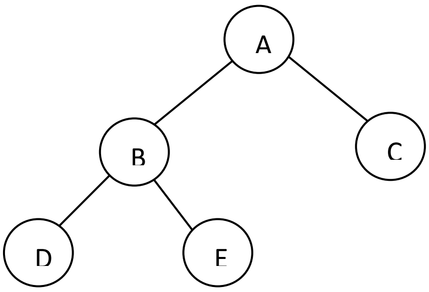


Depth-first traversal:

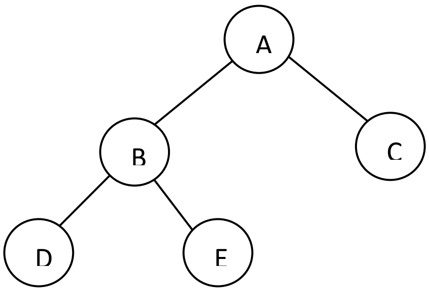
Open: B, C

Closed: A

Step 3



Step 4



Depth-first traversal:

Open: D, E, C

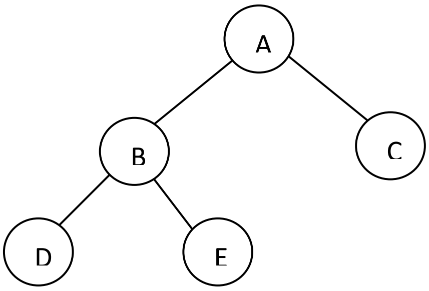
Closed: A,B

Depth-first traversal:

Open: E, C

Closed: A, B, D

Step 5

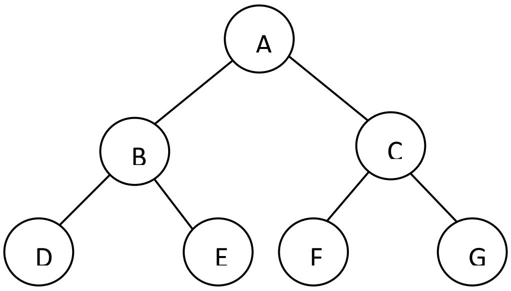


Depth-first traversal:

Open: C

Closed: A, B, D, E

Step 6

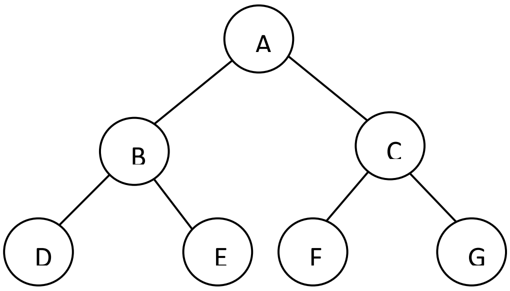


Depth-first traversal:

Open: F, G

Closed: A, B, D, E, C

Step 7

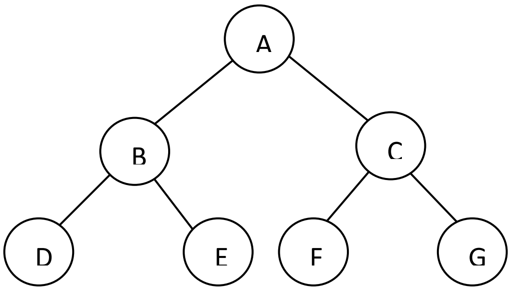


Depth-first traversal:

Open: G

Closed: A, B, D, E, C, F

Step 8



Depth-first traversal:

Open: <empty>

Closed: A, B, D, E, C, F, G

Advantages of Depth-First Search

- The advantage of depth-first Search is that memory requirement is only linear with respect to the search graph. This is in contrast with breadth-first search which requires more space. The reason is that the algorithm only needs to store a stack of nodes on the path from the root to the current node.
- The time complexity of a depth-first Search to depth d is O(b^d) since it generates the same set of nodes as breadth-first search, but simply in a different order. Thus practically depth-first search is time-limited rather than space-limited.
- If depth-first search finds solution without exploring much in a path then the time and space it takes will be very less.

Disadvantages of Depth-First Search

- The disadvantage of Depth-First Search is that there is a possibility that it may go down the left-most path forever. Even a finite graph can generate an infinite tree. One solution to this problem is to impose a cutoff depth on the search. Although the ideal cutoff is the solution depth d and this value is rarely known in advance of actually solving the problem. If the chosen cutoff depth is less than d, the algorithm will fail to find a solution, whereas if the cutoff depth is greater than d, a large price is paid in execution time, and the first solution found may not be an optimal one.
- Depth-First Search is not guaranteed to find the solution.
- And there is no guarantee to find a minimal solution, if more than one solution exists.

Implementing depth-first search in Prolog is very easy, because Prolog itself uses depth-first search during backtracking.

CONCLUSION : Students are advised to write conclusion on separate sheet.

REVIEW QUESTIONS:

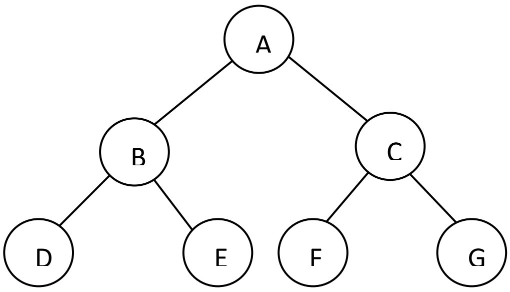
1. What do you mean by traversing?
2. What is the time complexity of DFS?
3. Explain how DFS uses stack?
4. Write a procedure for DFS.
5. Write a function for DFS.

EXPERIMENT NO :

Date :

AIM : Implementation of Breadth First Search.

PROBLEM STATEMENT : Write a program to traverse the following graph using BFS method. Consider A is the starting state and F is goal state.



THEORY :

Breadth First Search is a great algorithm for getting the shortest path to your goal (not applicable to graphs which have weights assigned to edges). Breadth First Search by the name itself suggests that the breadth of the search tree is expanded fully before going to the next step.

Now unlike Depth First Search we don't need a priority queue for this. We use two queues instead, one for expanding and one for temporary storing. Again each element of the queue is a path from the root of the tree.

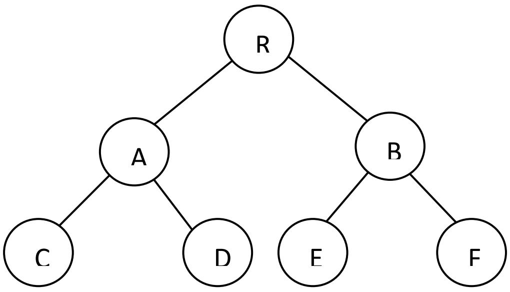
It searches breadth-wise in the problem space. Breadth-First search is like traversing a tree where each node is a state which may be a potential candidate for solution. It expands nodes from the root of the tree and then generates one level of the tree at a time until a solution is found. It is very easily implemented by maintaining a queue of nodes. Initially the queue contains just the root. In each iteration, node at the head of the queue is removed and then expanded. The generated child nodes are then added to the tail of the queue.

Since it never generates a node in the tree until all the nodes at shallower levels have been generated, breadth-first search always finds a shortest path to a goal. Since each node can be generated in constant time, the amount of time used by Breadth first search is proportional to the number of nodes generated, which is a function of the branching factor b and the solution d. Since the number of nodes at level d is b^d, the total number of nodes generated in the worst case is $b + b^2 + b^3 + \ldots + b^d$ i.e. $O(b^d)$, the asymptotic time complexity of breadth first search.

Algorithm: Breadth-First Search

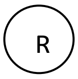
```
i. Declare two empty lists: Open and Closed.
ii. Add Start node to our Open list.
iii. While our Open list is not empty, loop the following:
    a. Remove the first node from our Open List.
    b. Check to see if the removed node is our destination.
        i. If the removed node is our destination, break out of the loop, add the node to our Closed list, and return the value of our Closed list.
        ii. If the removed node is not our destination, continue the loop (go to Step c).
    c. Extract the neighbors of our above removed node.
    d. Add the neighbors to the end of our Open list, and add the removed node to our Closed list.
```

Example



Look at the above tree with nodes starting from root node, R at the first level, A and B at the second level and C, D, E and F at the third level. If we want to search for node E then BFS will search level by level. First it will check if E exists at the root. Then it will check nodes at the second level. Finally it will find E the third level.

Step 1

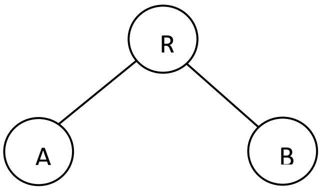


Breadth-first traversal:

Open: R

Closed: <empty>

Step 2

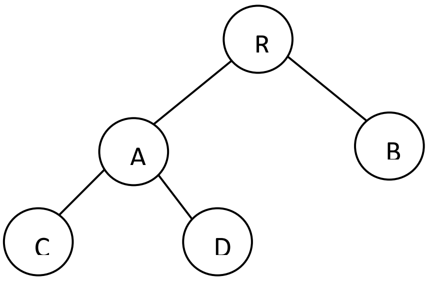


Breadth-first traversal:

Open: A, B

Closed: R

Step 3

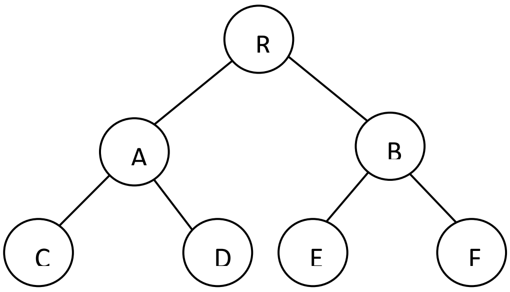


Breadth-first traversal:

Open: B, C, D

Closed: R, A

Step 4

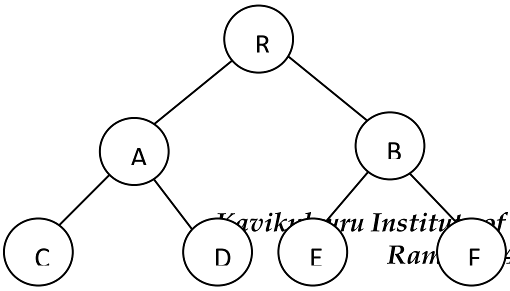


Breadth-first traversal:

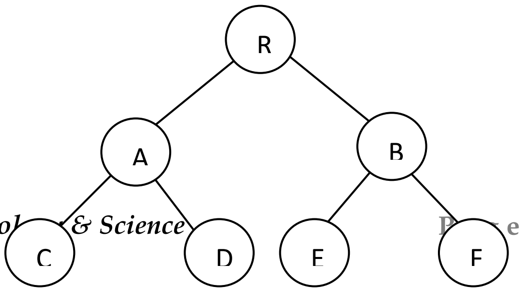
Open: C, D, E, F

Closed: R, A, B

Step 5



Step 6



Breadth-first traversal:

Open: D, E, F

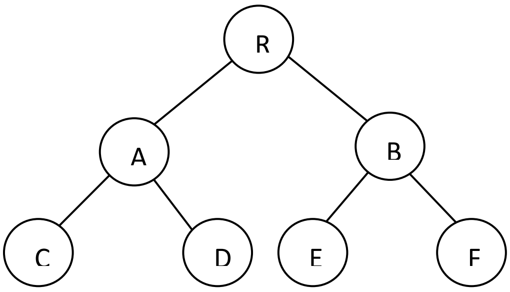
Closed: R, A, B, C

Breadth-first traversal:

Open: E, F

Closed: R, A, B, C, D

Step 7

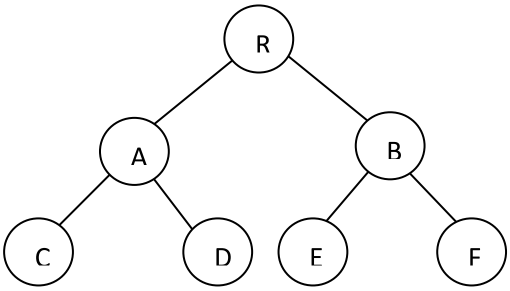


Breadth-first traversal:

Open: F

Closed: R, A, B, C, D, E

Step 8



Breadth-first traversal:

Open: < empty**>**

Closed: R, A, B, C, D, E, F

Advantages of Breadth-First Search

- Breadth first search will never get trapped exploring the useless path forever.
- If there is a solution, BFS will definitely find it out.
- If there is more than one solution then BFS can find the minimal one that requires less number of steps.

Disadvantages of Breadth-First Search

- The main drawback of Breadth first search is its memory requirement. Since each level of the tree must be saved in order to generate the next level, and the amount of memory is proportional to the number of nodes stored, the space complexity of BFS is O(b^d). As a result, BFS is severely space-bound in practice so will exhaust the memory available on typical computers in a matter of minutes. If the solution is farther away from the root, breath first search will consume lot of time.

CONCLUSION : Students are advised to write conclusion on separate sheet.

REVIEW QUESTIONS:

1. What is the time complexity of BFS?
2. Explain how BFS uses queue?
3. Write a procedure for BFS.
4. Difference between DFS and BFS.
5. Write a function for BFS.

EXPERIMENT NO :

Date :

AIM : Implementation of Traveling Salesman Problem.

PROBLEM STATEMENT : A traveling salesman has to travel through a bunch of cities, in such a way that the expenses on traveling are minimized. This means that to find the optimal solution through all possible routes.

THEORY :

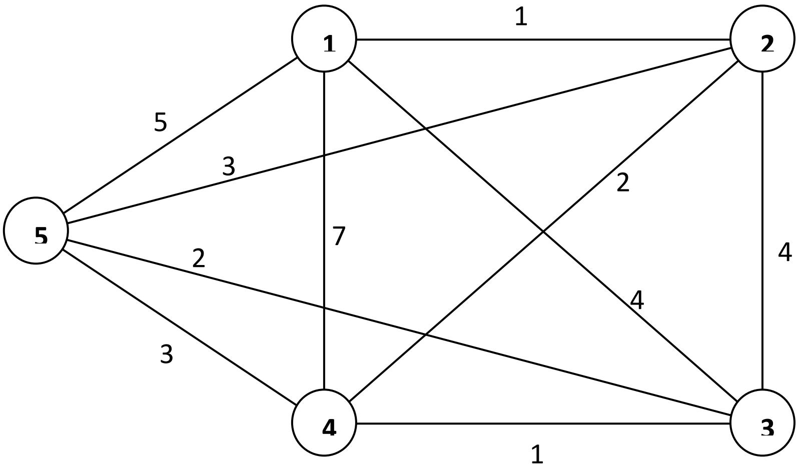
The traveling salesman problem (TSP) is one of the most intensively studied problems in computational mathematics and combinatorial optimization. It is also considered as the class of the NP-complete combinatorial optimization problems. By literatures, many algorithms and approaches have been launched to solve such the TSP. However, no current algorithms that can provide the exactly optimal solution of the TSP problem are available. Three AI search methods, i.e. Genetic Algorithms (GA), Tabu Search (TS), and Adaptive Tabu Search (ATS) can be used for such problems.

The traveling salesman problem consists of a salesman and a set of cities. The salesman has to visit each one of the cities starting from a certain one (e.g. the hometown) and returning to the same city. The challenge of the problem is that the traveling salesman wants to minimize the total length of the trip. The traveling salesman problem can be described as follows:

TSP = { (G, f, t) : G = (V, E) a complete graph, f is a function V x V → Z, t Є Z, G is a graph that contains a traveling salesman tour with cost that does not exceed t }.

Example

A salesman has list of cities, each of which he must visit exactly once(figure shows) there are direct roads between each pair of cities on the list. Find the route the salesman should follow so that he travels the shortest possible on a found trip, starting at any one of the cities and then returning there.



Solution

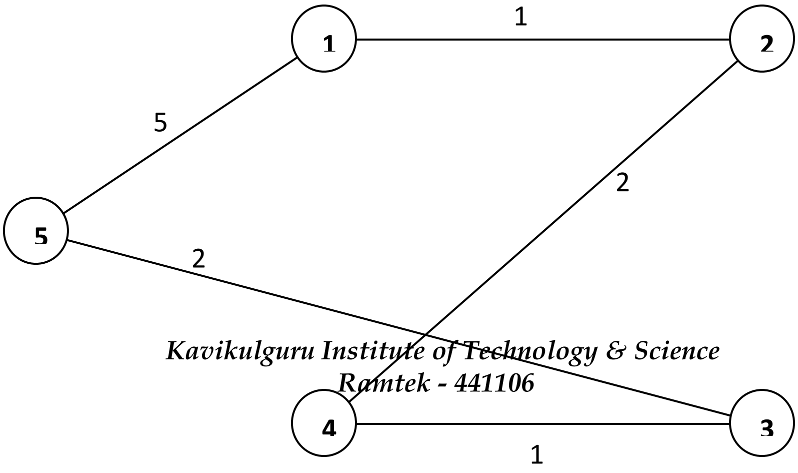
Using the heuristic search we may proceed to solve as follows:

Step 1. Arbitrarily select a starting city.

Step 2. To select the next city, look at all cities not yet visited.

Step 3. Select the one closest to the current city, go to it next.

Repeat Step 3 until all cities have been visited.



Algorithm produces a tour 1—2—4—3—5—1 with cost 1+2+1+2+5 = 11 is the minimum cost to travel cross various cities.

CONCLUSION : Students are advised to write conclusion on separate sheet.

REVIEW QUESTIONS:

1. What is Travelling Salesman Problem?
2. Which technique is used for solving this problem?
3. What do you mean by heuristic search?
4. What do you mean by characteristics “Is a good solution absolute or relative”?
5. What is cognitive exclusion?

EXPERIMENT NO :

Date :

AIM : Implementation of 8 puzzle problem.

PROBLEM STATEMENT : Given 3 x 3 grid with 8 square blocks labeled from 1 to 8 and a blank square. Propose a solution which converts the given initial state to final state.

8	7	6
5	4	1
2	3	
Initial State		

1	2	3
4	5	6
7	8	
Final State		

THEORY :

The 8-puzzle problem is a puzzle invented and popularized by Noyes Palmer Chapman in the 1870s. It is played on a 3-by-3 grid with 8 square blocks labeled 1 through 8 and a blank square. The goal is to rearrange the blocks so that they are in order. It is permitted to slide blocks horizontally or vertically into the blank square.

To solve a problem using a production system, specify the global database the rules, and the control strategy. For the 8 puzzle problem that correspond to these three components. These elements are the problem states, moves and goal. In this problem each tile configuration is a state. Once the problem states have been conceptually identified, construct a computer representation, or description of them. This description is then used as the database of a production system. For the 8-puzzle, a straight forward description is a 3X3 array of matrix of numbers. The initial global database is this description of the initial problem state. Virtually any kind of data structure can be used to describe states.

A move transforms one problem state into another state. The 8-puzzle is conveniently interpreted as having the following for moves. Move empty space (blank) to the left, move blank up, move blank to the right and move blank down,. These moves are modeled by production rules that operate on the state descriptions in the appropriate manner.

The rules each have preconditions that must be satisfied by a state description in order for them to be applicable to that state description. Thus the precondition for the rule associated with "move blank up" is derived from the requirement that the blank space must not already be in the top row.

Example

0	1	3					1		3					1	2	3
4	2	5		⇨			4	2	5		⇨			4		5
7	8	6					7	8	6					7	8	6
															⇩	
							1	2	3					1	2	3
							4	5	6		⇦			4	5	
							7	8						7	8	6

Number of states enqueued = 10

Number of moves = 4

CONCLUSION : Students are advised to write conclusion on separate sheet.

REVIEW QUESTIONS:

1. What is 8-puzzle problem?

8	7	6
5	4	1
2	3	

1	2	3
4	5	6
7	8	

Initial State Final State

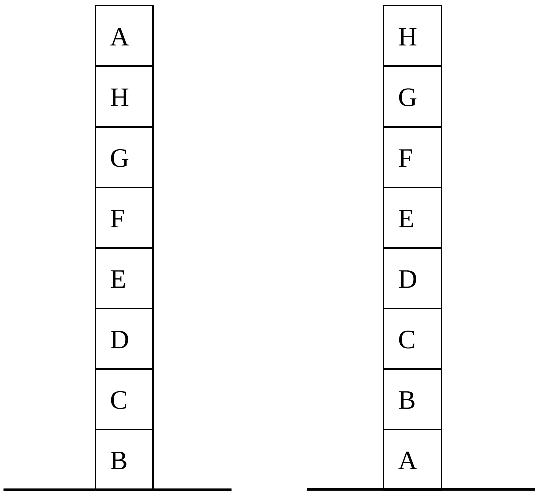
2. Which technique is used for solving 8-puzzle problem?
3. What do you mean by "Recoverable" characteristics?
4. What will be the different possible states to convert the following initial state to final state?
5. What are the different priority functions?

EXPERIMENT NO :

Date :

AIM : Implementation of Hill climbing Algorithm.

PROBLEM STATEMENT : Consider the initial state , obtain goal state by applying Hill Climbing Algorithm.



Initial State Goal State

THEORY :

In computer science, *hill climbing* is a mathematical optimization technique which belongs to the family of local search. It is relatively simple to implement, making it a popular first choice. Although more advanced algorithms may give better results, in some situations hill climbing works just as well.

Hill climbing is used widely in artificial intelligence, for reaching a goal state from a starting node. Choice of next node and starting node can be varied to give a list of related algorithms.

Problems

- ***Local maxima***

A problem with hill climbing is that it will find only local maxima. Unless the heuristic is convex, it may not reach a global maximum. Other local search algorithms try to overcome this problem such as stochastic hill climbing, random walks and simulated annealing. This problem of hill climbing can be solved by using random hill climbing search technique.

- ***Ridges***

A ridge is a curve in the search place that leads to a maximum, but the orientation of the ridge compared to the available moves that are used to climb is such that each move will lead to a smaller point. In other words, each point on a ridge looks to the algorithm like a local maximum, even though the point is part of a curve leading to a better optimum.

- ***Plateau***

Another problem with hill climbing is that of a plateau, which occurs when we get to a "flat" part of the search space, i.e. we have a path where the heuristics are all very close together. This kind of flatness can cause the algorithm to cease progress and wander aimlessly.

To overcome these problems we can

- Back track to some earlier nodes and try a different direction. This is a good way of dealing with local maxim.
- Make a big jump to some direction to a new area in the search. This can be done by applying two more rules of the same rule several times, before testing. This is a good strategy is dealing with plate and ridges.

CONCLUSION : Students are advised to write conclusion on separate sheet.

REVIEW QUESTIONS:

1. What are the different types of Hill Climbing technique?
2. What is the difference between Hill Climbing and Generate & Test Technique?
3. What are the different problems in Hill Climbing Technique?
4. How to overcome the problems of Hill Climbing Technique?
5. Explain where we can use Hill Climbing Technique?

EXPERIMENT NO :

Date :

AIM : Implementation of Water Jug Problem.

PROBLEM STATEMENT :

Given 2 jugs, a 4 liter one and a 3- liter one, neither has any measuring marks on it. There is a pump that can be used to fill the jugs with water. Get exactly 2 liters of water in to the 4-liter jugs?

THEORY :

The state space for this problem can be defined as { (i ,j) i = 0, 1, 2, 3, 4, j = 0, 1, 2, 3 }, 'i' represents the number of liters of water in the 4-liter jug and 'j' represents the number of liters of water in the 3-liter jug. The initial state is (0,0) that is no water on each jug. The goal state is to get (2,n) for any value of 'n'.

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.

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 Predicate: State = (2,y) where $0 \le y \le 3$.

Solution 1

Liters in the 4-liter jug	Liters in the 3-liter jug	Rule applied
0	0	
4	0	1
1	3	8
1	0	6
0	1	10

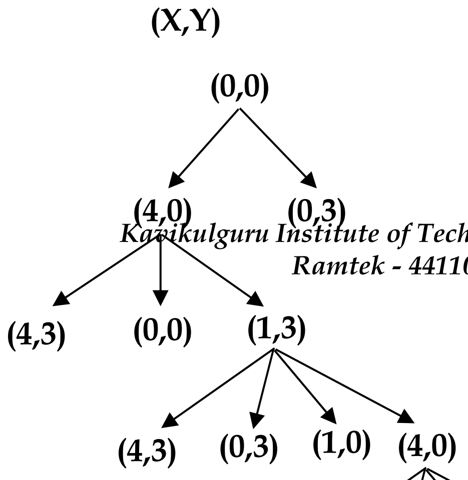
4	1	1
2	3	8

Solution 2

Liters in the 4-liter jug	Liters in the 3-liter jug	Rule applied
0	0	
0	3	2
3	0	9
3	3	2
4	2	7
0	2	5
2	0	9

Solution 3

Liters in the 4-liter jug	Liters in the 3-liter jug	Rule applied
0	0	
4	0	1
1	3	8
0	3	5
3	0	9
3	3	2
4	2	7
0	2	5
2	0	9



CONCLUSION : Students are advised to write conclusion on separate sheet.

REVIEW QUESTIONS:

1. What is Water Jug Problem?
2. What do you mean by Production System?
3. Which type of search algorithm is used?
4. Write the solution for the Water Jug Problem 5-liter and 2-liter jug, at the end 5-liter jug have1-liter of water.
5. What is state space representation?

EXPERIMENT NO :

Date :

AIM : Implementation of Missionaries Cannibal problem.

PROBLEM STATEMENT :

Three missionaries and three cannibals find themselves on one side of a river. They would like to get to the other side. But the missionaries are not sure what else the cannibals agreed to. So the missionaries managed the trip across the river in such a way that the number of missionaries on either side of the river is never less than the number of cannibals who are on the same side. The only boat available holds only two at a time. Propose a solution so that everyone get across the river without the missionaries risking being eaten?

THEORY :

Three missionaries and three cannibals are on one side of a river, along with a boat that can hold one or two people. We have to find a way to get everyone to the other side, without ever leaving a group of missionaries in one place outnumbered by the cannibals in other side.

Solution

The state for this problem can be defined as { (i, j)/ i=0, 1, 2, 3, j=0, 1, 2, 3 }, where i represents the number missionaries in one side of a river. j represents the number of cannibals in the same side of river. The initial state is (3,3), that is three missionaries and three cannibals on one side of a river, (bank 1) and (0,0) on another side of the river (bank 2) . The goal state is to get (3,3) at bank 2 and (0,0) at bank 1.

To solve this problem following assumptions have been made:

1. Number of cannibals should lesser than the missionaries on either side.
2. Only one boat is available to travel.
3. Only one or maximum of two people can go in the boat at a time.
4. All the six have to cross the river from bank.

5. There is no restriction on the number of trips that can be made to reach of the goal.
6. Both the missionaries and cannibals can row the boat.

This kind of problem is often solved by a graph search method. Represent the problem as a set of states which are snapshots of the world and operators which transform one state into another state are mapped to nodes of the graph and operators are the edges of the graph.

Representing the missionaries and cannibals problem.

A State is one "snapshot" in time.

For this problem the only information we need to fully characterize the state is:

The number of missionaries on the left bank,

The number of cannibals on the left bank,

The side the boat is on.

All other information can be deduced from these three items.

In PROLOG, the state can be represented by a 3-arity term, state (Missionaries, Cannibals, State).

Representing the solution

The solution consists of a list of moves, e.g. [move(I.I. right), move 2.0, lift)]. We take this to mean that I missionary and i cannibal moved to the right bank, then 2 missionaries moved to the left bank.

Like the graph search problem, we must avoid returning to state we have visited before.

Overview of solution

We follow a simple graph search procedure:

- Start from an initial state
- Find a neighboring state
- Check that the now state has not been visited before
- Find a path from the neighbor to the goal.

The search terminates when we have found the state: state(0, 0, right).

Production Rules for Missionaries and Cannibals problem.

Rule No	Production Rule and Action.
1	(i, j) : Two missionaries can go only when i-2>=j or i-2=0 in one bank and i+2>=j in the other bank.
2	(i, j) : Two cannibals can cross the river only when j-2<=i or i=0 in one bank and j+2 <= i or i+0 or i=0 in the other.
3	(i, j) : One missionary and one cannibal can go in a boat only when i-1>=j-1 or i=0 in one bank and i+1>=j+1 or i=0 in the other.
4	(i, j) : one missionary can cross the river only when i-1>=j or i=0 in one bank and i+1>=j in the other bank.
5	(i, j) : One cannibal can cross the river only when j-1 < i or i=0 in one bank and j+1<=I or j=0 in the other bank of the river.

Possible Moves

A move is characterized by the number of missionaries and the number of cannibals taken in the boat at one time. Since the boat can carry no more than two people at once, the only feasible combinations are:

Carry (2, 0).

Carry (1, 0).

Carry (1, 1).

Carry (0, 1).

Carry(0, 2).

Where Carry (M, C) means the boat will carry M missionaries and C cannibals on one trip.

Feasible Moves

Once we have found a possible move, we have to confirm that it is feasible. It is not a feasible to move more missionaries or more cannibals than that are present on one bank. When the state is state(M1, C1, left) and we try carry (M,C) then M <= M1 and C<= C1 must be true.

When the state is state(M1, C1, right) and we try carry(M,C) then M + M1 <= 3 and C + C1 <= 3 must be true.

Legal Moves

Once we have found a feasible move, we must check that is legal i.e. no missionaries must be eaten.

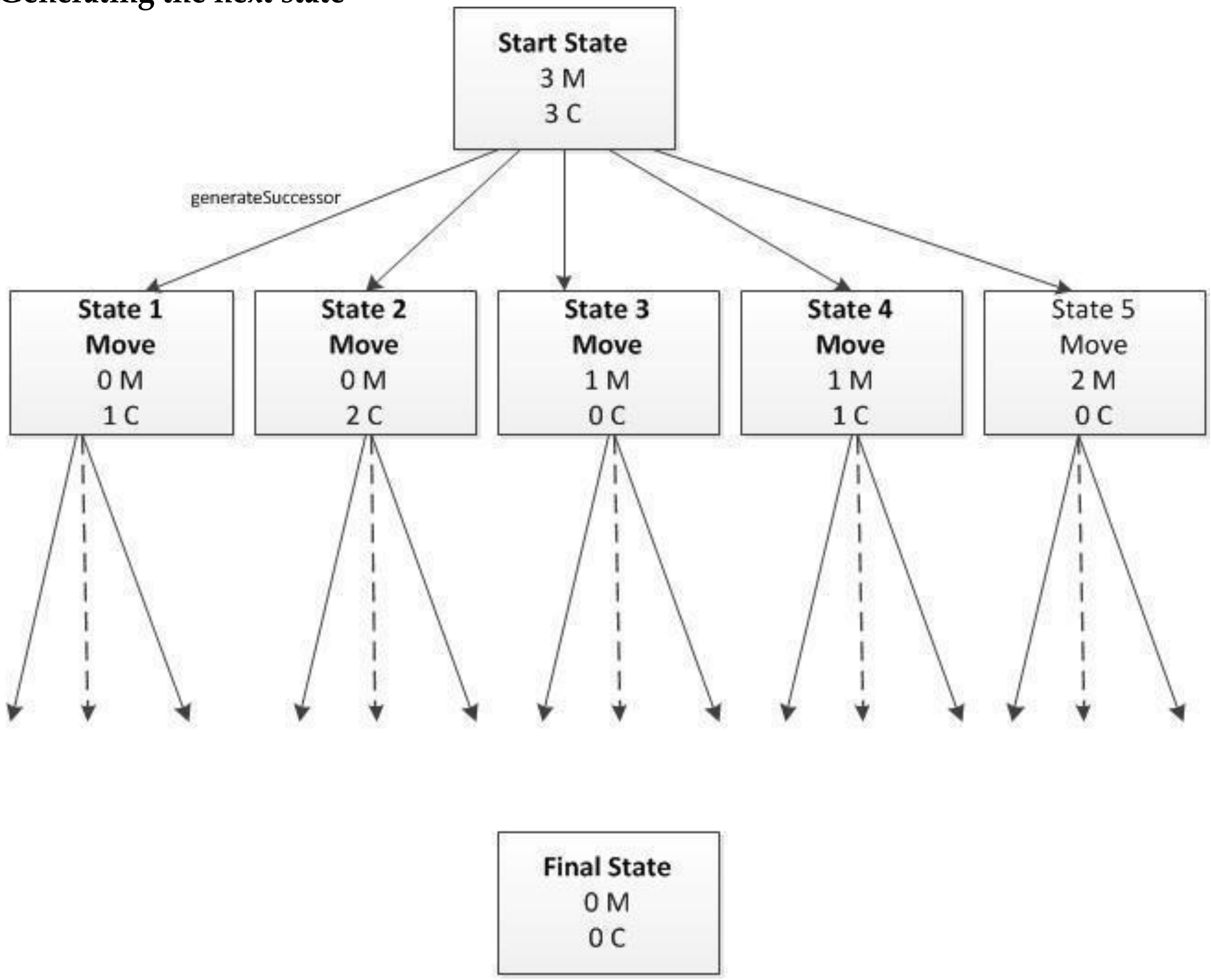
Legal(X, X).

Legal(3, X).

Legal(0, X).

The only safe combinations are when there are equal numbers of missionaries and cannibals or all the missionaries are on one side.

Generating the next state



Breadth First Vs Depth Searches in Missionaries and Cannibals Problem

Breadth first search is guaranteed to find the shortest path from starting state to ending state. In my program it finds the solution at level 12 from the starting node. The state utilization of breadth first, measured in terms of size of the open list, is B^n where B is the branching factor – the average number of descendants per state – and n is the level.

Depth first search is NOT guaranteed to find the shortest path but its gets deeply into the search space. In missionaries and Cannibals problem, depth first search enters into a graph cycle and suffers a infinite loop. That is, it cannot find the goal state. This problem is removed by using a array list which keeps record of the visited nodes. HOWEVER this makes the space utilization of depth first search as poor as Breadth first search. In normal cases, depth first is more efficient for spaces with large branching factor, because it does not have to keep all nodes at a given level on the open. Therefore space utilization of depth first is linear : B * n, since at each level the open list contains the children of a single node.

CONCLUSION : Students are advised to write conclusion on separate sheet.

REVIEW QUESTIONS:

1. What do you mean by missionaries cannibals problem?
2. Which technique is used to solve this problem?
3. What do you mean by graph search algorithm?
4. Describe a heuristic function for this problem.
5. What is back tracking?

EXPERIMENT NO :

Date :

AIM : Implementation of Tower of Hanoi problem.

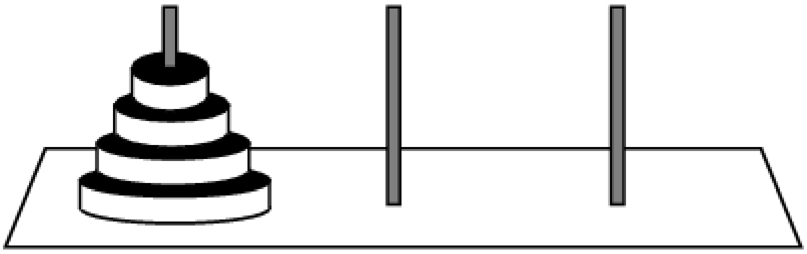
PROBLEM STATEMENT :

There are 3 pegs that can hold disks of 3 different sizes. You start with all disks being on the left peg, as shown below. The goal is to get all disks to the right peg, but you can only move one disk at a time, and you can never put a disk on top of a smaller disk.

THEORY :

The Tower of Hanoi, also called the Tower of Brahma or Towers of Brahma, is a mathematical game or puzzle. It was invented by French mathematician Eduard Lucas in 1883.

It consists of three rods, and a number of disks of different sizes which can slide onto any rod. The puzzle starts with the disks in a neat stack in ascending order of size on one rod, the smallest at the top, thus making a conical shape.



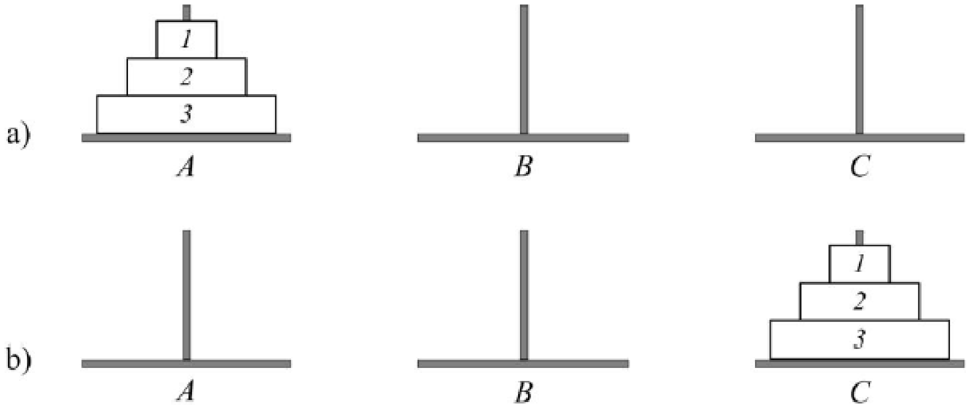
The objective of the puzzle is to move the entire stack to another rod, obeying the following rules:

- ➢ Only one disk may be moved at a time.
- ➢ Each move consists of taking the upper disk from one of the rods and sliding it onto another rod, on top of the other disks that may already be present on that rod.
- ➢ No disk may be placed on top of a smaller disk.

The three-disc Tower-of-Hanoi has three rods A, B, and C, on which can be placed three discs (Labeled D3, D2, D1). Disc D3 is larger than disc D2 which is in turn larger than disc D1. We start with the three discs on rod A and want to move them to one of the other rods.

Let the operators that describe actions be given by the schema *move*(*x; y*), where *x* can be any of the three discs *D*1, *D*2, or *D*3, and *y* can be any rod *A*, *B*, or *C*.

Example



a) Initial State { (3,2,1), ( ), ( ) }
b) Terminal State { ( ), ( ), (3,2,1) }

This is an intelligent task. Intelligence remains in the recursive algorithm below:

Part 1. Move n1 discs from A to B;

Part 2. Move disc n from A to C;

Part 3. Move n1 discs from B to C.

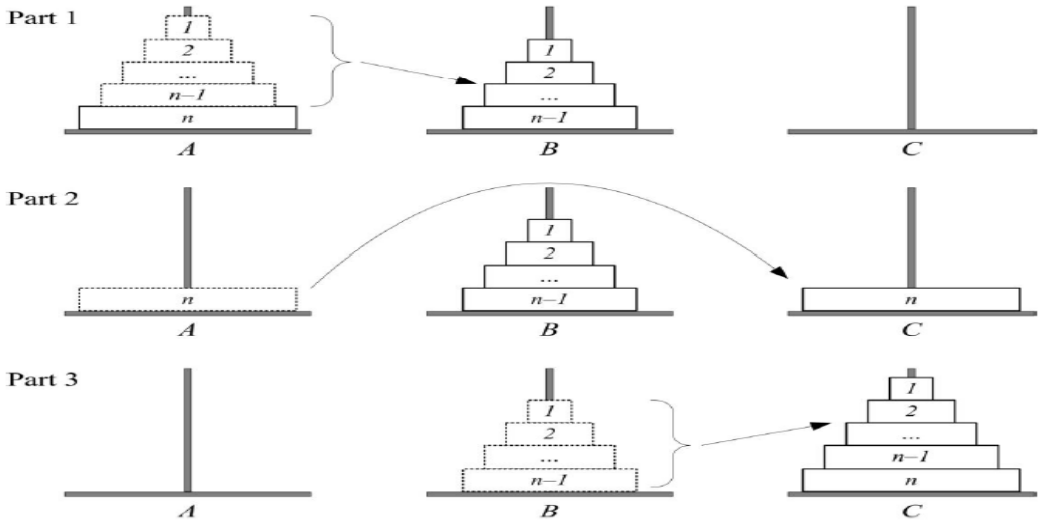


Figure : Three part of recursive algorithm

A sequence of moves for n=3:

{ (3,2,1), ( ), ( ) } initial state;

1. { (3,2), ( ), (1) } move disc 1 from A to C;
2. { (3), (2), (1) } move disc 2 from A to B;
3. { (3), (2,1), ( ) } move disc 1 from C to B;
4. { ( ), (2,1), (3) } move disc 3 from A to C;
5. { (1), (2), (3) } move disc 1 from B to A;
6. { (1), ( ), (3,2) } move disc 2 from B to C;
7. { ( ), ( ), (3,2,1) } move disc 1 from A to C.

A recursive procedure:

```
procedure ht(x, y, z: char; n: integer);
{x, y, z  rod names; n  number of discs.}
{x  from, y  intermediary, z  onto.}
begin
if n > 0 then
begin
ht(x, z, y, n-1); {1. Move n-1 discs onto intermediary.}
```

```
writeln(' Move from ', x,' to ', z);              {2.}
ht(y, x, z, n-1); {3. Move n-1 discs onto target.     }
end
end
```

The number of moves is exponential 2^{n-1}.

State transition in state transition graphs, pateiktas GRAPHSEARCH_DEPTH_FIRST search tree for the Hanoi tower problem n=3.

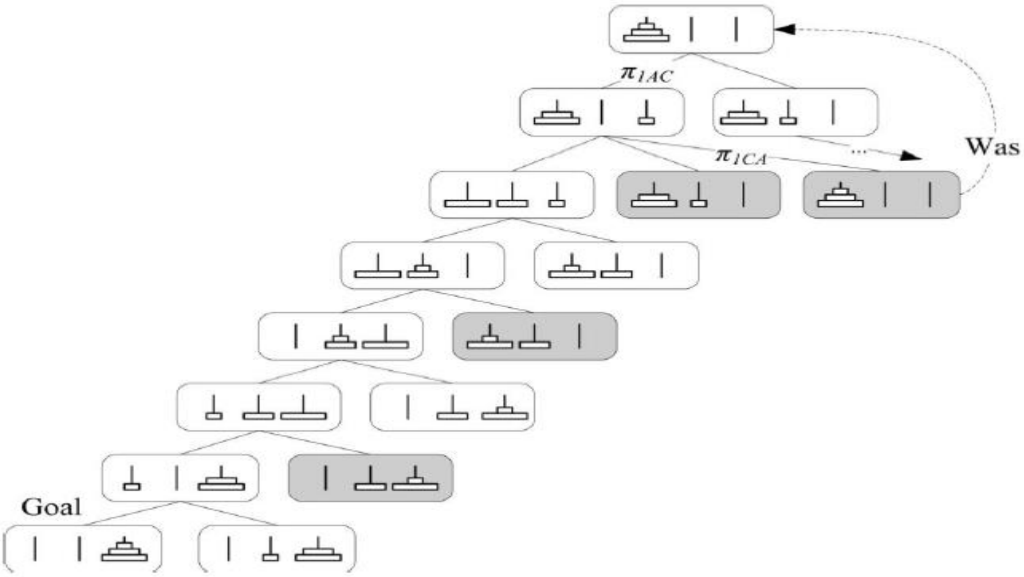


Figure : A search tree of algorithm GRAPHSEARCH_DEPTH_FIRST for the Hanoi tower problem n=3. The gray nodes appear in the list OPEN or CLOSED

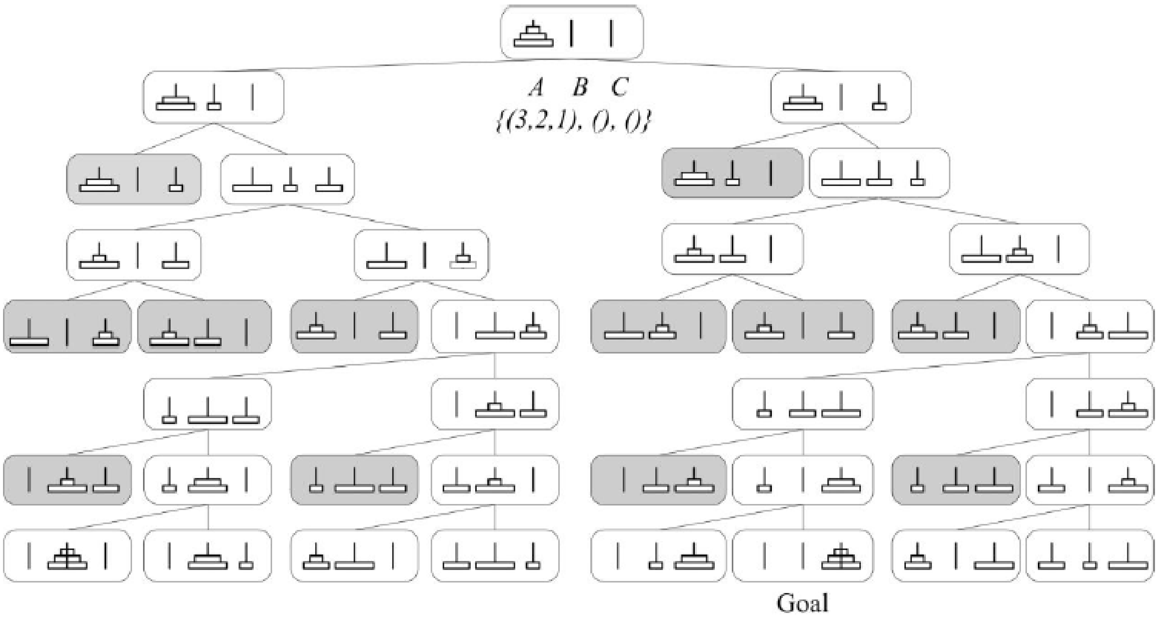


Figure: A search tree of the GRAPHSEARCH_BREADTH_FIRST algorithm for the Hanoi tower problem. The gray nodes appear in the list OPEN or CLOSED

CONCLUSION : Students are advised to write conclusion on separate sheet.

REVIEW QUESTIONS:

1. What is Tower of Hanoi problem?
2. Which technique is used to solve this problem?
3. Name and explain the characteristics in which this problem belongs.
4. Identify a goal state and draw the search space containing all possible states of the puzzle (Consider the example discussed in theory part).
5. What are the steps involved in designing a program to solve an AI-program?

EXPERIMENT NO :

Date :

AIM : Implementation of A* algorithm.

PROBLEM STATEMENT :

Solve 8 puzzle problem using A* algorithm.

THEORY :

The A* algorithm combines features of uniform-cost search and pure heuristic search to efficiently compute optimal solutions. A* algorithm is a best-first search algorithm in which the cost associated with a node is f(n) = g(n) + h(n), where g(n) is the cost of the path from the initial state to node n and h(n) is the heuristic estimate or the cost or a path from node n to a goal. Thus, f(n) estimates the lowest total cost of any solution path going through node n. At each point a node with lowest f value is chosen for expansion. Ties among nodes of equal f value should be broken in favor of nodes with lower h values. The algorithm terminates when a goal is chosen for expansion.

A* algorithm guides an optimal path to a goal if the heuristic function h(n) is admissible, meaning it never overestimates actual cost. For example, since airline distance never overestimates actual highway distance, and manhatten distance never overestimates actual moves in the gliding tile.

A* uses a best-first search and finds a least-cost path from a given initial node to one goal node(out of one or more possible goals). As A* traverses the graph, it follows a path of the lowest known heuristic cost, keeping a sorted priority queue of alternate path segments along the way. It uses a distance-plus-cost heuristic function (usually denoted f(x)) to determine the order in which the search visits nodes in the tree. The distance-plus-cost heuristic is a sum of two functions: the path-cost function, which is the cost from the starting node to the current node (usually denoted g(x))an admissible "heuristic estimate" of the distance to the goal (usually denoted h(x)).

The h(x) part of the f(x) function must be an admissible heuristic; that is, it must not overestimate the distance to the goal. Thus, for an application like routing, h(x) might represent the straight-line distance to the goal, since that is physically the smallest possible distance between any two points or nodes. If the heuristic satisfies the additional condition $h(x)h(x) \leq d(x, y) + h(y)$for every edge$x, y$of the graph (where$d$denotes the length of that edge), then is called monotone, or consistent. In such a case, A* can be implemented more efficiently—roughly speaking, no node needs to be processed more than once—and A* is equivalent to running Dijkstra's algorithm with the reduced cost$d'(x, y) \coloneqq d(x, y) - h(x) + h(y)$.

For Puzzle, A* algorithm, using these evaluation functions, can find optimal solutions to these problems. In addition, A* makes the most efficient use of the given heuristic function in the following sense: among all shortest-path algorithms using the given heuristic function h(n). A* algorithm expands the fewest number of nodes.

The main drawback of A* algorithm and indeed of any best-first search is its memory requirement. Since at least the entire open list must be saved, A* algorithm is severely space-limited in practice, and is no more practical than best-first search algorithm on current machines. For example, while it can be run successfully on the eight puzzle, it exhausts available memory in a matter of minutes on the fifteen puzzle.

Algorithm for A*

1. Start with open containing only the initial node. Set that node's g value to 0, its h′ value to whatever it is, and its f′ value to h′+0, or h′. set closed to the empty list.
2. Until a goal node is fund, repeat the following procedure: If there are no nodes on OPEN, report failure. Otherwise, pick the node on OPEN with the lowest f′ value. Call it BESTNODE. Remove it from OPEN. Place it on closed. See if BESTNODE is a goal node. If so, exit and report a solution (either BESTNODE if all we want is the node or the path that has been created between the initial states or BESTNODE if we are interested in the path). Otherwise, generate the successor of BESTNODE but do not set BESTNODE to point to them yet. (First we need to see if any of them have already been generated.) For each such SUCCESSOR, do the following:

a. Set successor to point back to BESTNODE. These backward links will make it possible to recover the path once a solution is found.

b. Compute g(SUCCESSOR) =g(BESTNODE) + the cost of getting from BESTNODE to SUCCESSOR.

c. See if SUCCESSOR is the same as any node on OPEN (i.e. it has already been generated but not processed). If so, call that node OLD. Since his node already exist in the graph, we can throw SUCCESSOR away and add OLD to the list of BESTNODE's successor. Now we must decide whether OLD's parent link should be reset to point to BESTNODE. It should be if the path we have just found to SUCCESSOR is cheaper that the current best path to OLD . so see whether it is cheaper to get to OLD via its current parent or to SUCCESSOR via BESTNODE by comparing their g values. If OLD is cheaper then we need do nothing. If SUCCESSOR is cheaper, then reset OLD's parent link to point to BESTNODE, record the new cheaper path in g(OLD), and update f'(OLD).

d. If SUCCESSOR was not on OPEN, see it is on CLOSED. If so, call the node on CLOSED OLD and add OLD to the list of BESTNODE's, successor.

If SUCCESSOR was not already on either OPEN or CLOSED, then put it on OPEN, and add it to the list of BESTNODE's successors. Compute f'(SUCCESSOR)= g(SUCCESSOR) + h'(SUCCESSOR).

CONCLUSION : Students are advised to write conclusion on separate sheet.

REVIEW QUESTIONS:

1. What is A* algorithm?
2. What is Dijkstra algorithm?
3. What is worst case space complexity of A* algorithm?
4. What is the application of A* algorithm?
5. What do you mean by branching factor(b)?

EXPERIMENT NO :

Date :

AIM : Implementation of AO* algorithm.

PROBLEM STATEMENT :

Implement AO* algorithm for problem reduction.

THEORY :

When a problem can be divided into a set of sub problems, where each sub problem can be solved separately and a combination of these will be a solution, AND-OR graphs or AND - OR trees are used for representing the solution. The decomposition of the problem or problem reduction generates AND arcs. One AND may point to any number of successor nodes. All these must be solved so that the arc will rise to many arcs, indicating several possible solutions. Hence the graph is known as AND - OR instead of AND. Figure shows an AND - OR graph.

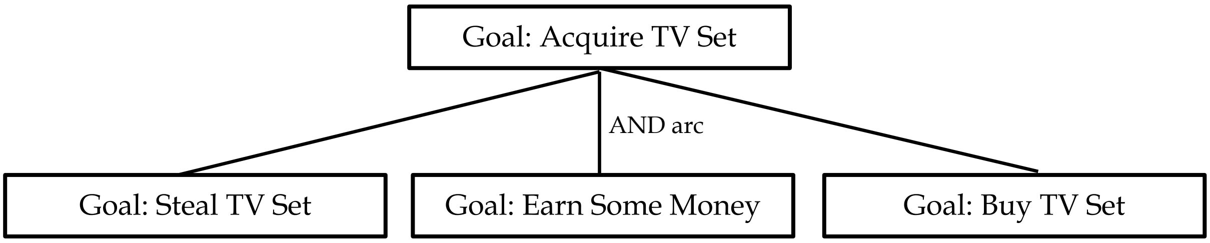


Fig: AND-OR graph an example

An algorithm to find a solution in an AND - OR graph must handle AND area appropriately. A* algorithm cannot search AND - OR graphs efficiently. This can be understand from the given figure.

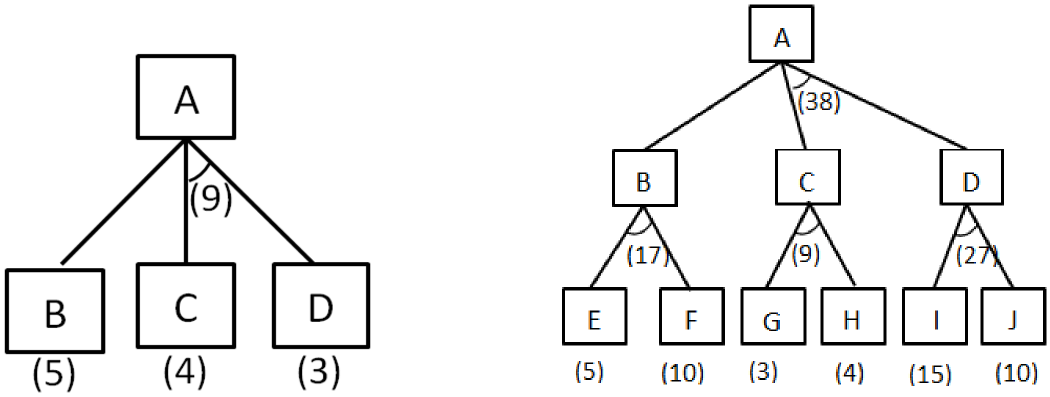


Fig: AND-OR Graph

In figure (a) the top node A has been expanded producing two area one leading to B and leading to C-D. The numbers at each node represent the value of f ' at that node (cost of getting to the goal state from current state). For simplicity, it is assumed that every operation(i.e. applying a rule) has unit cost, i.e., each are with single successor will have a cost of 1 and each of its components. With the available information till now , it appears that C is the most promising node to expand since its f ' = 3 , the lowest but going through B would be better since to use C we must also use D' and the cost would be 9(3+4+1+1). Through B it would be 6(5+1).

Thus the choice of the next node to expand depends not only n a value but also on whether that node is part of the current best path form the initial mode. Figure (b) makes this clearer. In figure the node G appears to be the most promising node, with the least f ' value. But G is not on the current beat path, since to use G we must use GH with a cost of 9 and again this demands that arcs be used (with a cost of 27). The path from A through B, E-F is better with a total cost of (17+1=18). Thus we can see that to search an AND-OR graph, the following three things must be done.

1. Traverse the graph starting at the initial node and following the current best path, and accumulate the set of nodes that are on the path and have not yet been expanded.
2. Pick one of these unexpanded nodes and expand it. Add its successors to the graph and computer f ' (cost of the remaining distance) for each of them.

3. Change the f ' estimate of the newly expanded node to reflect the new information produced by its successors. Propagate this change backward through the graph. Decide which of the current best path.

The propagation of revised cost estimation backward is in the tree is not necessary in A* algorithm. This is because in AO* algorithm expanded nodes are re-examined so that the current best path can be selected. The working of AO* algorithm is illustrated in figure as follows:

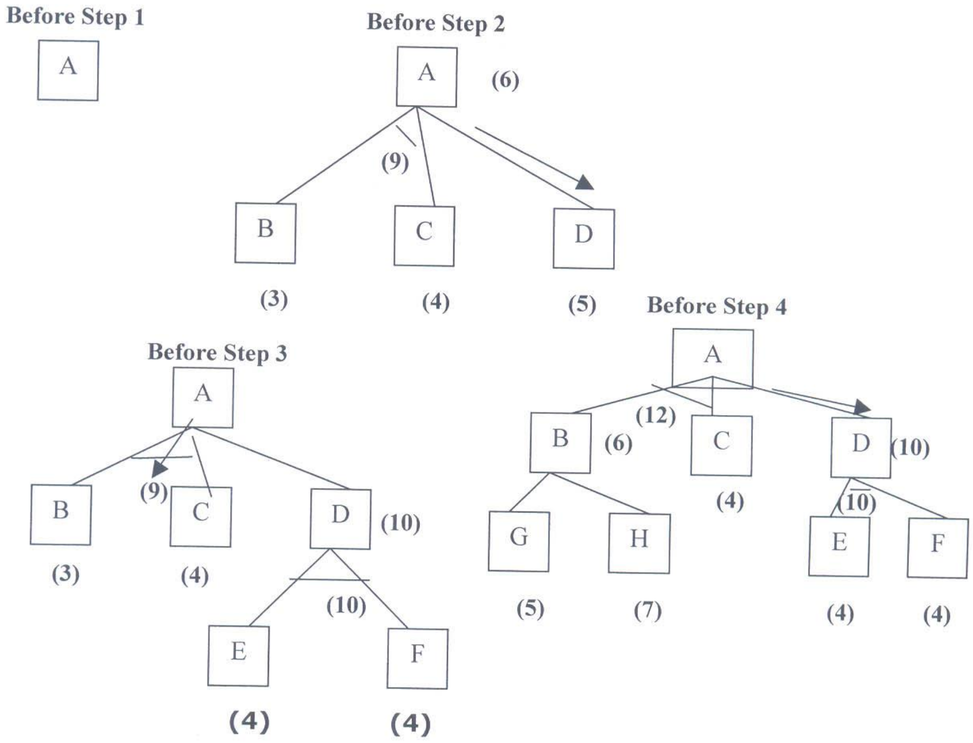


Fig: Working of AO* Algorithm

Referring the figure. The initial node is expanded and D is Marked initially as promising node. D is expanded producing an AND arc E-F. f ' value of D is updated to 10. Going backwards we can see that the AND arc B-C is better. It is now marked as current best path. B and C have to be expanded next. This process continues until a solution is found or all paths have led to dead ends, indicating that there is no solution. An A* algorithm the path from one node to the other is always that of the lowest cost and it is independent of the paths through other nodes.

The algorithm for performing a heuristic search of an AND - OR graph is given below. Unlike A* algorithm which used two lists OPEN and CLOSED, the AO* algorithm uses a single structure G. G represents the part of the search graph generated so far. Each node in G points down to its immediate successors and up to its immediate predecessors, and also has with it the value of h' cost of a path from itself to a set of solution nodes. The cost of getting from the start nodes to the current node "g" is not stored as in the A* algorithm. This is because it is not possible to compute a single such value since there may be many paths to the same state. In AO* algorithm serves as the estimate of goodness of a node. Also a there should value called FUTILITY is used. The estimated cost of a solution is greater than FUTILITY then the search is abandoned as too expansive to be practical.

For representing above graphs AO* algorithm is as follows

AO* ALGORITHM

1. Let G consists only to the node representing the initial state call this node INTT. Compute h' (INIT).
2. Until INIT is labeled SOLVED or hi (INIT) becomes greater than FUTILITY, repeat the following procedure.
   a. Trace the marked arcs from INIT and select an unbounded node NODE.
   b. Generate the successors of NODE. if there are no successors then assign FUTILITY as h' (NODE). This means 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 G
      ii. If successor is not a terminal node, mark it solved and assign zero to its h ' value.
      iii. If successor is not a terminal node, compute it h' value.
   c. Propagate the newly discovered information up the graph by doing the following. Let S be a set of nodes that have been marked SOLVED. Initialize S to NODE. Until S is empty repeat the following procedure;
      i. Select a node from S call if CURRENT and remove it from S.

ii. Compute h' of each of the arcs emerging from CURRENT, Assign minimum h' to CURRENT.

iii. Mark the minimum cost path as the best out of CURRENT.

iv. Mark CURRENT SOLVED if all of the nodes connected to it through the new marked are have been labeled SOLVED.

v. If CURRENT has been marked SOLVED or its h ' has just changed, its new status must be propagate backwards up the graph. Hence all the ancestors of CURRENT are added to S.

AO* Search Procedure

1. Place the start node on open.
2. Using the search tree, compute the most promising solution tree tp .
3. Select node n that is both on open and a part of tp, remove n from open and place it no closed.
4. If n is a goal node, label n as solved. If the start node is solved, exit with success where tp is the solution tree, remove all nodes from open with a solved ancestor.
5. If n is not solvable node, label n as unsolvable. If the start node is labeled as unsolvable, exit with failure. Remove all nodes from open, with unsolvable ancestors.
6. Otherwise, expand node n generating all of its successor compute the cost of for each newly generated node and place all such nodes on open.
7. Go back to step(2).

Note: AO* will always find minimum cost solution.

CONCLUSION : Students are advised to write conclusion on separate sheet.

REVIEW QUESTIONS:

1. What is AO* algorithm?
2. What is worst case space complexity of AO* algorithm?
3. What is the application of AO* algorithm?
4. How is A* algorithm is different from AO* algorithm?
5. What is AND/OR graph?

EXPERIMENT NO :

Date :

AIM : Implementation of Rule Based Expert System.

PROBLEM STATEMENT :

Construct an expert system for medical diagnosis.

THEORY :

Rule based expert system

In computer science, rule-based systems are used as a way to store and manipulate knowledge to interpret information in a useful way. They are often used in artificial intelligence applications and research.

Applications

A classic example of a rule-based system is the domain-specific expert system that uses rules to make deductions or choices. For example, an expert system might help a doctor choose the correct diagnosis based on a cluster of symptoms, or select tactical moves to play a game.

Rule-based systems can be used to perform lexical analysis to compile or interpret computer programs, or in natural language processing.

Rule-based programming attempts to derive execution instructions from a starting set of data and rules. This is a more indirect method than that employed by an imperative programming language, which lists execution steps sequentially.

Construction

A typical rule-based system has four basic components

- A list of rules or rule base, which is a specific type of knowledge base.

- An inference engine or semantic reasoner, which infers information or takes action based on the interaction of input and the rule base. The interpreter executes a production system program by performing the following match-resolve-act cycle.
- Match: In this first phase, the left-hand sides of all productions are matched against the contents of working memory. As a result a conflict set is obtained, which consists of instantiations of all satisfied productions. An instantiation of a production is an ordered list of working memory elements that satisfies the left-hand side of the production.
- Conflict-Resolution: In this second phase, one of the production instantiations in the conflict set is chosen for execution. If no productions are satisfied, the interpreter halts.
- Act: In this third phase, the actions of the production selected in the conflict-resolution phase are executed. These actions may change the contents of working memory. At the end of this phase, execution returns to the first phase.
- Temporary working memory.
- A user interface or other connection to the outside world through which input and output signals are received and sent.

Some of the important advantages of expert systems are as follows:

- Ability to capture and preserve irreplaceable human experience.
- Ability to develop a system more consistent than human experts.
- Minimize human expertise needed at a number of locations at the same time (especially in a hostile environment that is dangerous to human health);
- Solutions can be developed faster than human experts.

The basic components of an expert system are illustrated in Figure 1. The knowledge base stores all relevant information, data, rules, cases, and relationships used by the expert system. A knowledge base can combine the knowledge of multiple human experts. A rule is a conditional statement that links given conditions to actions or outcomes. A frame is another approach used to capture and store knowledge in a knowledge base. It relates an objector item to various facts or values. A frame-based representation is ideally suited for object-oriented programming techniques. Expert systems making use of frames to store knowledge are also called frame-based expert systems.

The purpose of the inference engine is to seek information and relationships from the knowledge base and to provide answers, predictions, and suggestions in the way a human expert would. The inference engine must find the right facts, interpretations, and rules and assemble them correctly. Two types of inference methods are commonly used – Backward chaining is the process of starting with conclusions and working backward to the supporting facts. Forward chaining starts with the facts and works forward to the conclusions.

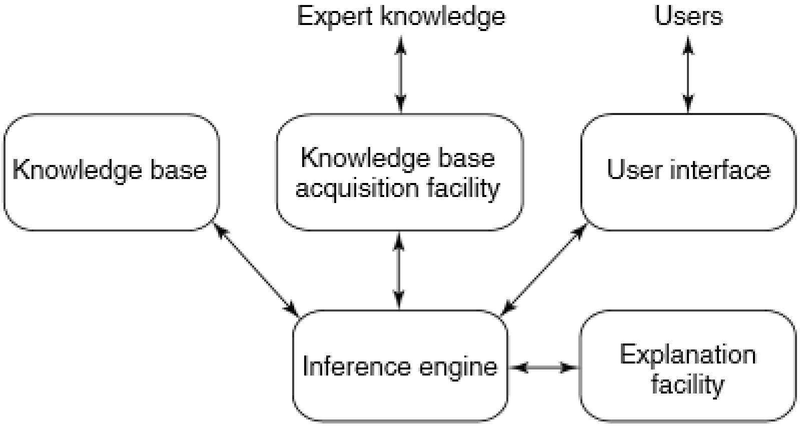


Figure : Architecture of Simple Expert System

The explanation facility allows a user to understand how the expert system arrived at certain results. The overall purpose of the knowledge acquisition facility is to provide a convenient and efficient means for capturing and storing all components of the knowledge base. Very often specialized user interface software is used for designing, updating, and using expert systems. The purpose of the user interface is to ease use of the expert system for developers, users, and administrators.

Inference Engine in Rule-Based Systems

A rule-based system consists of if-then rules, a bunch of facts, and an interpreter controlling the application of the rules, given the facts. These if-then rule statements are used to formulate the conditional statements that comprise the complete knowledge base. A single if-then rule assumes the form if x is A then y is B and the if-part of the rule 'x is A' is called the antecedent or premise, while the then-part of the rule 'y is B' is called the consequent or conclusion. There are two broad kinds of inference engines used in rule-based systems: forward chaining and backward chaining systems. In a forward chaining system, the initial facts are processed first, and keep using the rules to draw new conclusions given those facts. In a backward chaining system, the hypothesis (or solution/goal) we are trying to reach is processed first, and keep looking for rules that would allow to conclude that hypothesis. As the processing progresses, new sub goals are also set for validation. Forward chaining systems are primarily data-driven, while backward chaining systems are goal-driven. Consider an example with the following set of if-then rules

Rule 1: If A and C then Y

Rule 2: If A and X then Z

Rule 3: If B then X

Rule 4: If Z then D If the task is to prove that D is true,

Given A and B are true.

According to forward chaining, start with Rule 1 and go on downward till a rule that fires is found. Rule 3 is the only one that fires in the first iteration. After the first iteration, it can be concluded that A, B, and X are true. The second iteration uses this valuable information. After the second iteration, Rule 2 fires adding Z is true, which in turn helps

Rule 4 to fire, proving that D is true. Forward chaining strategy is especially appropriate in situations where data are expensive to collect, but few in quantity. However, special care is to be taken when these rules are constructed, with the preconditions specifying as precisely as possible when different rules should fire. In the backward chaining method, processing starts with the desired goal, and then attempts to find evidence for proving the goal. Returning to the same example, the task to prove that D is true would be initiated by first finding a rule that proves D. Rule 4 does so, which also provides a sub goal to prove that Z is true. Now Rule 2 comes into play, and as it is already known that A is true, the new sub goal is to show that X is true. Rule 3 provides the next sub goal of proving that B is true. But that B is true is one of the given assertions. Therefore, it could be concluded that X is true, which implies that Z is true, which in turn also implies that D is true. Backward chaining is useful in situations where the quantity of data is potentially very large and where some specific characteristic of the system under consideration is of interest. If there is not much knowledge what the conclusion might be, or there is some specific hypothesis to test, forward chaining systems may be inefficient. In principle, we can use the same set of rules for both forward and backward chaining. In the case of backward chaining, since the main concern is with matching the conclusion of a rule against some goal that is to be proved, the 'then' (consequent) part of the rule is usually not expressed as an action to take but merely as a state, which will be true if the antecedent part(s) are true(Donald, 1986).

CONCLUSION : Students are advised to write conclusion on separate sheet.

REVIEW QUESTIONS:

1. What is an Expert System?
2. Define forward and backward chaining?
3. Draw architecture of simple expert system.
4. What is rule based expert system?
5. What are the different applications of expert system?

EXPERIMENT NO :

Date :

AIM : Industrial relevance of computational intelligence.

PROBLEM STATEMENT : To study the computational intelligence driven robotics applied research with industrial relevance.

THEORY :

Computational Intelligence

Computational intelligence is the study of the design of intelligent agents. An agent is something that acts in an environment—it does something. Agents include worms, dogs, thermostats, airplanes, humans, organizations, and society. An intelligent agent is a system that acts intelligently: What it does is appropriate for its circumstances and its goal, it is flexible to changing environments and changing goals, it learns from experience, and it makes appropriate choices given perceptual limitations and finite computation. The central scientific goal of computational intelligence is to understand the principles that make intelligent behavior possible, in natural or artificial systems. The main hypothesis is that reasoning is computation. The central engineering goal is to specify methods for the design of useful, intelligent artifacts.

Artificial Intelligence

Artificial intelligence (AI) is the established name for the field we have defined as computational intelligence (CI), but the term "artificial intelligence" is a source of much confusion. Is artificial intelligence real intelligence? Perhaps not, just as an artificial pearl is a fake pearl, not a real pearl. "Synthetic intelligence" might be a better name, since, after all, a synthetic pearl may not be a natural pearl but it is a real pearl.

However, since we claimed that the central scientific goal is to understand both natural and artificial (or synthetic) systems, we prefer the name "computational intelligence." It also has the advantage of making the computational hypothesis explicit in the name. The confusion about the field's name can, in part, be attributed to a confounding of the field's purpose with its methodology. The purpose is to understand how intelligent behavior is possible. The methodology is to design, build, and experiment with computational systems that perform tasks commonly viewed as intelligent. Building these artifacts is an essential activity since computational intelligence is, after all, an empirical science; but it shouldn't be confused with the scientific purpose.

Another reason for eschewing the adjective "artificial" is that it connotes simulated intelligence. Contrary to another common misunderstanding, the goal is not to simulate intelligence. The goal is to understand real (natural or synthetic) intelligent systems by synthesizing them. A simulation of an earthquake isn't an earthquake; however, we want to actually create intelligence, as you could imagine creating an earthquake.

The misunderstanding comes about because most simulations are now carried out on computers. However, you shall see that the digital computer, the archetype of an interpreted automatic, formal, symbol-manipulation system, is a tool unlike any other:

It can produce the real thing.

The obvious intelligent agent is the human being. Many of us feel that dogs are intelligent, but we wouldn't say that worms, insects, or bacteria are intelligent. There is a class of intelligent agents that may be more intelligent than humans, and that is the class of organizations. Ant colonies are the prototypical example of organizations. Each individual ant may not be very intelligent, but an ant colony can act more intelligently than any individual ant. The colony can discover food and exploit it very effectively as well as adapt to changing circumstances. Similarly, companies can develop, manufacture, and distribute products where the sum of the skills required is much more than any individual could understand. Modern computers, from the low-level hardware to high-level software, are more complicated than can be understood by any human, yet they are manufactured daily by organizations of humans. Human society viewed as an agent is probably the most intelligent agent known. We take inspiration from both biological and organizational examples of intelligence.

Computational intelligence (CI) is a set of nature-inspired computational methodologies and approaches to address complex real-world problems to which traditional approaches, i.e., first principles modeling or explicit statistical modeling, are ineffective or infeasible. Many such real-life problems are not considered to be well-posed problems mathematically, but nature provides many counterexamples of biological systems exhibiting the required function, practically. For instance, the human body has about 200 joints (degrees of freedom), but humans have little problem in executing a target movement of the hand, specified in just three Cartesian dimensions. Even if the torso were mechanically fixed, there is an excess of 7:3 parameters to be controlled for natural arm movement. Traditional models also often fail to handle uncertainty, noise and the presence of an ever-changing context. Computational Intelligence provides solutions for such and other complicated problems and inverse problems. It primarily includes artificial neural networks, evolutionary computation and fuzzy logic. In addition, CI also embraces biologically inspired algorithms such as swarm intelligence and artificial immune systems, which can be seen as a part of evolutionary computation, and includes broader fields such as image processing, data mining, and natural language processing. Furthermore other formalisms: Dempster–Shafer theory, chaos theory and many-valued logic are used in the construction of computational models.

The characteristic of "intelligence" is usually attributed to humans. More recently, many products and items also claim to be "intelligent". Intelligence is directly linked to the reasoning and decision making. Fuzzy logic was introduced in 1965 as a tool to formalize and represent the reasoning process and fuzzy logic systems which are based on fuzzy logic possess many characteristics attributed to intelligence. Fuzzy logic deals effectively with uncertainty that is common for human reasoning, perception and inference and, contrary to some misconceptions, has a very formal and strict mathematical backbone ('is quite deterministic in itself yet allowing uncertainties to be effectively represented and manipulated by it', so to speak). Neural networks, introduced in 1940s (further developed in 1980s) mimic the human brain and represent a computational mechanism based on a simplified mathematical model of the perceptrons (neurons) and signals that they process. Evolutionary computation, introduced in the 1970s and more popular since the 1990s mimics the population-based sexual evolution through reproduction of generations. It also mimics genetics in so called genetic algorithms.

Agents in the World

There are many interesting philosophical questions about the nature and substance of CI, but the bottom line is that, in order to understand how intelligent behavior might be algorithmic, you must attempt to program a computer to solve actual problems. It isn't enough to merely speculate that some particularly interesting behavior is algorithmic. You must develop a theory that explains how that behavior can be manifest in a machine, and then you must show the feasibility of that theory by constructing an implementation. We are interested in practical reasoning: reasoning in order to do something. Such a coupling of perception, reasoning, and acting comprises an agent.

An agent could be, for example, a coupling of a computational engine with physical actuators and sensors, called a robot. It could be the coupling of an advice-giving computer—an expert system—with a human who provides the perceptual information and who carries out the task. An agent could be a program that acts in a purely computational environment—an infobot.

Figure shows the inputs and outputs of an agent. At any time the agent has:

- Prior knowledge about the world
- Past experience that it can learn from
- Goals that it must try to achieve or values about what is important
- Observations about the current environment and itself and it does some action. For each agent considered, we specify the forms of the inputs and the actions. The goal of this book is to consider what is in the black box so that the action is reasonable given the inputs.

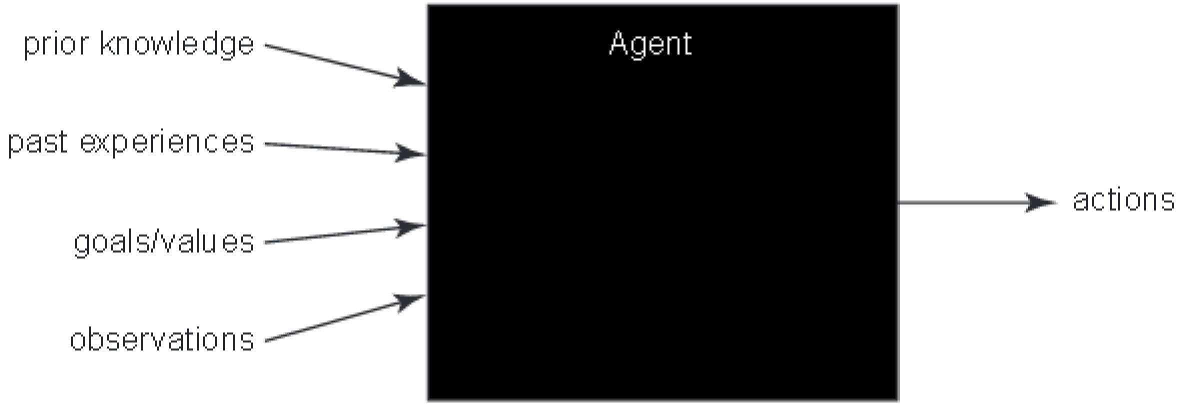


Figure : An agent as a black box

For our purpose, the world consists of an agent in an environment. The agent's environment may well include other agents. Each agent has some internal state that can encode beliefs about its environment and itself. It may have goals to achieve, ways to act in the environment to achieve those goals, and various means to modify its beliefs by reasoning, perception, and learning. This is an all-encompassing view of intelligent systems varying in complexity from a simple thermostat to a team of mobile robots to a diagnostic advising system whose perceptions and actions are mediated by human beings.

Success in building an intelligent agent naturally depends on the problem that one selects to investigate. Some problems are very well-suited to the use of computers, such as sorting a list of numbers. Others seem not to be, such as changing a baby's diaper or devising a good political strategy. We have chosen some problems that are representative of a range of applications of current CI techniques. We seek to demonstrate, by case study, CI's methodology with the goal that the methodology is transferable to various problems in which you may be interested. We establish a framework that places you, the reader, in a position to evaluate the current CI literature and anticipate the future; and, most importantly, we develop the concepts and tools necessary to allow you to build, test, and modify intelligent agents. Finally we must acknowledge there is still a huge gulf between the dream of computational intelligence and the current technology used in the practice of building what we now call intelligent agents. We believe we have many of the tools necessary to build intelligent agents, but we are certain we don't have all of them. We could, of course, be on the wrong track; it is this fallibility that makes CI science and makes the challenge of CI exciting.

Applications

Theories about representation and reasoning are only useful insofar as they provide the tools for the automation of problem solving tasks. CI's applications are diverse, including medical diagnosis, scheduling factory processes, robots for hazardous environments, chess playing, autonomous vehicles, natural language translation systems, and cooperative systems. Rather than treating each application separately, we abstract essential features of such applications to allow us to study principles behind intelligent reasoning and action. This section outlines three application domains that will be developed in examples throughout the book. Although the particular examples presented are simple—for otherwise they wouldn't fit into the book—the application domains are representative of the sorts of domains in which CI techniques can be, and have been, used.

The three application domains are:

- An autonomous delivery robot that can roam around a building delivering packages and coffee to people in the building. This delivery agent needs to be able to, for example, find paths, allocate resources, receive requests from people, make decisions about priorities, and deliver packages without injuring people or itself.
- A diagnostic assistant that helps a human troubleshoot problems and suggests repairs or treatments to rectify the problems. One example is an electrician's assistant that can suggest what may be wrong in a house, such as a fuse blown, a light switch broken, or a light burned out given some symptoms of electrical problems. Another example is of a medical diagnostician that finds potential diseases, possible tests, and appropriate treatments based on knowledge of a particular medical domain and a patient's symptoms and history. This assistant needs to be able to explain its reasoning to the person who is carrying out the tests and repairs, as that person is ultimately responsible for what they do. The diagnostic assistant must add substantial value in order to be worth using.

- An "infobot" that can search for information on a computer system for naïve users such as company managers or people off the street. In order to do this the infobot must find out, using the user's natural language, what information is requested, determine where to find out the information, and access the information from the appropriate sources. It then must report its findings in an appropriate format so that the human can understand the information found, including what they can infer from the lack of information.

CONCLUSION : Students are advised to write conclusion on separate sheet.

REVIEW QUESTIONS:

1. What is computational intelligence?
2. What are the different branches in computational intelligence?
3. Write down different applications of computational intelligence.
4. Explain how computational intelligence is useful in solving simple problems solving?
5. Explain how computational intelligence is useful in Robotics?