Problem Solving and Problem Reduction

Problem Solving: State Space Search and Control Strategies:

Problem Solving is a method of deriving solution steps beginning from initial description of the problem to the desired solution. It has been conventionally one of the focus areas of the Artificial Intelligence (AI) and can be characterized as a systematic search using a range of possible steps to achieve some predefined solution.

The two types of problem solving methods that are generally followed include General Purpose and Special Purpose methods General Purpose method is applicable to a wide variety of problems, Special Purpose method is tailor made for a particular problem and often exploits very specific features of the problem.

For generating new state in the search space, an action/operator/rule is applied and tested whether the state is the goal state or not. In case the state is not the goal state, the procedure is repeated. The order of application of the rules to the current state is called control strategy. Since AI programs involve clean separation of computational components of operations, control and data, it is useful to structure of AI programs in such a way that it helps describe search process efficiently that forms the core of many intelligent processes

General Problem Solving: Production Systems and State Space Search method facilitate the modeling of problems and search processes.

Production System:- Production System is one of the formalisms that helps AI programs to do search process more conveniently in state space problems. This system comprises of start (initial) state (s) and goal (final) state (s) of the problem along with one or more databases consisting of suitable and necessary information for the particular task. Production System (PS) consists of number of production rules in which the production rule has left side that determines the applicability of the rule and a right side that describes the action to be performed if the rule is applied. Left side of the rule is current state whereas the right side describes the new state that is obtained from applying the rule. System which is used for expert opinion in a specific domain

1. It is a good way to model the strong state driven nature of intelligent action
2. New rules can be easily added to account for new situations without disturbing the rest of the system.
3. It is quite important in real time environment and applications where new input to the database changes the behaviour of the system.

Water Jug Problem:

Problem Statement: We have two jugs, a 5-gallon (5g) and the other 3-gallon (3g) with no measuring marker on them. There is endless supply of water through tap. Our task is to get 4 gallon of water in the 5-g jug.

Solution: State Space for this problem can be described as the set of ordered pairs of integers (x, y) such that x represents the number of gallons of water in 5-g jug and y for 3-g jug.

1. Start State is $(0, 0)$
2. Goal state is $(4, N)$ for any value of $N \leq 3$

The possible operations that can be used in this problem are listed as follows:

- Fill 5-g jug from the tap and empty the 5-g jug by throwing water down the drain
- Fill 3-g jug from the tap and empty the 3-g jug by throwing water down the drain
- Pour some or 3-g water from 5-g into the 3-g jug to make it full
- Pour some of full 3-g jug water into 5-g jug.

These operations can formally be defined as production rules as given in the below table.

Rule No.	Left of the rule	Right of the rule	Description
1.	$(x, y \mid x < 5)$	$(5, y)$	Fill 5-g jug
2.	$(x, y \mid x > 0)$	$(0, y)$	Empty 5-g jug
3.	$(x, y \mid y < 3)$	$(x, 3)$	Fill 3-g jug
4.	$(x, y \mid y > 0)$	$(x, 0)$	Empty 3-g jug
5.	$(x, y \mid x+y \leq 5 \wedge y > 0)$	$(x+y, 0)$	Empty 3-g jug into 5-g jug
6.	$(x, y \mid x+y \leq 3 \wedge x \neq 0)$	$(0, x+y)$	Empty 5-g jug into 3-g jug
7.	$(x, y \mid x+y \geq 5 \wedge y > 0)$	$(5, y-(5-x))$	Pour water from 3g jug into 5-g jug until 5g jug is full
8.	$(x, y \mid x+y \geq 3 \wedge x > 0)$	$(x-(3-y), 3)$	Pour water from 5g jug into 3g jug until 3-g jug is full.

There are two different solutions for the above water jug problem.

Solution Path 1:

Rule applied	5-g jug	3-g jug	Step No
Start State	0	0	
1	5	0	1
8	2	3	2
4	2	0	3
6	0	2	4
1	5	2	5
8	4	3	6
Goal state	4	0+3 = 3	

Solution Path 2:

Rule Applied	5-g jug	3-g jug	Step NO
Start State	0	0	
3	0	3	1
5	3	0	2
3	3	3	3
7	5	1	4
2	0	1	5
5	1	0	6
3	1	3	7
5	4	0	8
Goal state	4	0	

Missionaries and Cannibals Problem:

Problem statement: Three missionaries and three cannibals want to cross a river. There is a boat on their side of the river that can be used by either one or two persons. How should they use the boat to cross the river in such a way that cannibals never outnumber missionaries on either side of the river? If the cannibals ever outnumber the missionaries then the missionaries will be eaten. How can they all cross over without anyone being eaten?

Solution: State space for this problem can be described as the set of ordered pairs of left and right banks of the river as (L, R) where each bank is represented as a list [nM, mC, B]. Here n is number of missionaries M, m is number of cannibals C and B represents Boat.

1. Start state : ([3M, 3C, 1B], [0M, 0C, 0B])

2. Any state : $([n_1M, m_1C,], [n_2M, m_2C,])$ at any state

$$n_1(\neq 0) \geq m_1 ; \; n_2(\neq 0) \geq m_2 ; \; n_1 + n_2 = 3, \; m_1 + m_2 = 3$$

3. Goal state : ([0M, 0C, 0B], [3M, 3C, 1B])

N	Left side of the rule		Right side of the rule
	Rules for boat going from left bank to right bank of river		
1	$([n_1M, m_1C, 1B], [n_2M, m_2C, 0B])$	$\rightarrow$	$([(n_1-2)M, m_1C, 0B], [(n_2+2)M, m_2C, 1B])$
2	$([n_1M, m_1C, 1B], [n_2M, m_2C, 0B])$	$\rightarrow$	$([(n_1-1)M, (m_1-1)C, 0B], [(n_2+1)M, (m_2+1)C, 1B])$
3	$([n_1M, m_1C, 1B], [n_2M, m_2C, 0B])$	$\rightarrow$	$([n_1M, (m_1-2)C, 0B], [n_2M, (m_2+2)C, 1B])$
4	$([n_1M, m_1C, 1B], [n_2M, m_2C, 0B])$	$\rightarrow$	$([(n_1-1)M, m_1C, 0B], [(n_2+1)M, m_2C, 1B])$
5	$([n_1M, m_1C, 1B], [n_2M, m_2C, 0B])$	$\rightarrow$	$([n_1M, (m_1-1)C, 0B], [n_2M, (m_2-1)C, 1B])$
	Rules for boat coming from right bank to left bank of river		
1	$([n_1M, m_1C, 0B], [n_2M, m_2C, 1B])$	$\rightarrow$	$([(n_1+2)M, m_1C, 1B], [(n_2-2)M, m_2C, 0B])$
2	$([n_1M, m_1C, 0B], [n_2M, m_2C, 1B])$	$\rightarrow$	$([(n_1+1)M, (m_1+1)C, 1B], [(n_2-1)M, (m_2-1)C, 0B])$
3	$([n_1M, m_1C, 0B], [n_2M, m_2C, 1B])$	$\rightarrow$	$([n_1M, (m_1+2)C, 1B], [n_2M, (m_2-2)C, 0B])$
4	$([n_1M, m_1C, 0B], [n_2M, m_2C, 1B])$	$\rightarrow$	$([(n_1+1)M, m_1C, 0B], [(n_2-1)M, m_2C, 0B])$
5	$([n_1M, m_1C, 0B], [n_2M, m_2C, 1B])$	$\rightarrow$	$([n_1M, (m_1+1)C, 0B], [n_2M, (m_2-1)C, 0B])$

Solution Path:

Rule Number	([3M, 3C, 1B], [0M, 0C, 0B]) ← start State
L2	([2M, 2C, 0B], [1M, 1C, 1B])
R4	([3M, 2C, 1B], [0M, 1C, 0B])
L3	([3M, 0C, 0B], [0M, 3C, 1B])
R5	([3M, 1C, 1B], [0M, 2C, 0B])
L1	([1M, 1C, 0B], [2M, 2C, 1B])
R2	([2M, 2C, 1B], [1M, 1C, 0B])
L1	([0M, 2C, 0B], [3M, 1C, 0B])
R5	([0M, 3C, 1B], [3M, 0C, 0B])
L3	([0M, 1C, 0B], [3M, 2C, 1B])
R5	([0M, 2C, 1B], [3M, 1C, 0B])
L3	([0M, 0C, 0B], [3M, 3C, 1B]) ← Goal state

State Space Search:- Similar to production system, state space is another method of problem representation that facilitates easy search. Using this method, one can also find a path from start state to goal state while solving a problem.

A state space basically consists of four components:

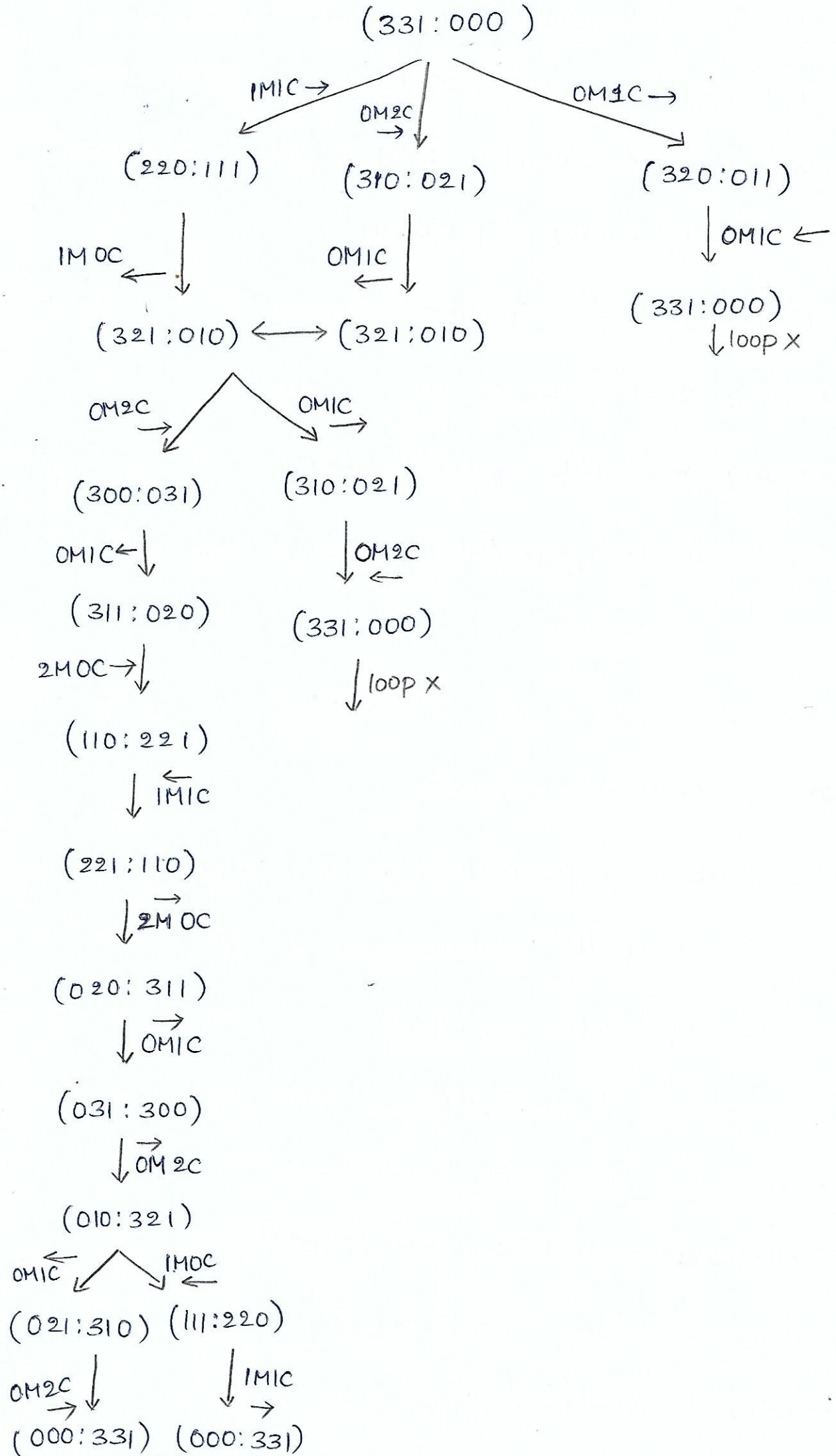
1. A set s containing start states of the problem.
2. A set G containing goal states of the problem
3. Set of nodes (states) in the graph / tree. Each node represents the state in problem solving process
4. Set of arcs connecting nodes. Each arc corresponds to operator that is a steps in problem solving process.

A solution path is a path through the graph from a node in s to a node in G. The main objective of search algorithm is to determine a solution path in the graph. There may be many solution paths. Commonly used approach is to apply appropriate operator to transfer from one state to another.

Let us consider Missionaries and Cannibals problem

The possible operators are {2MOC, 1MIC, OM, 2C, 1MOC, OMIC}

1. Start state : (3M3C1B : OMOCOB) simply (331 : 000)
2. Goal state : (OMOCOB : 3M3C1B) simply (000 : 331)


(331:000)
1M1C
0M2C
0M1C
(220:111)
(310:021)
(320:011)
1M0C
0M1C
0M1C
(321:010)
(321:010)
(331:000)
loop x
0M2C
0M1C
(300:031)
(310:021)
0M1C
0M2C
(311:020)
(331:000)
2M0C
loop x
(110:221)
1M1C
(221:110)
2M0C
(020:311)
0M1C
(031:300)
0M2C
(010:321)
0M1C
1M0C
(021:310)
(111:220)
0M2C
1M1C
(000:331)
(000:331)

For example,

- An invalid state like (1M2C1B : 2M1C0B) is not possible state, as it leads to one missionary and two cannibals on the left bank.

In case of a valid state like (2M2C1B : 1M1C0B), the operator 0M1C or 0M2C would be illegal. Hence the operators when applied should satisfy some conditions that shouldn't lead to invalid states.

Two Solution paths are:

Solution Path 1	Solution Path 2
1M1C →	1M1C →
1M0C ←	1M0C ←
0M2C →	0M2C →
0M1C ←	0M1C ←
2M0C →	2M0C →
1M1C ←	1M1C ←
2M0C →	2M0C →
0M1C ←	0M1C ←
0M2C →	0M2C →
0M1C ←	1M0C ←
0M2C →	1M1C →

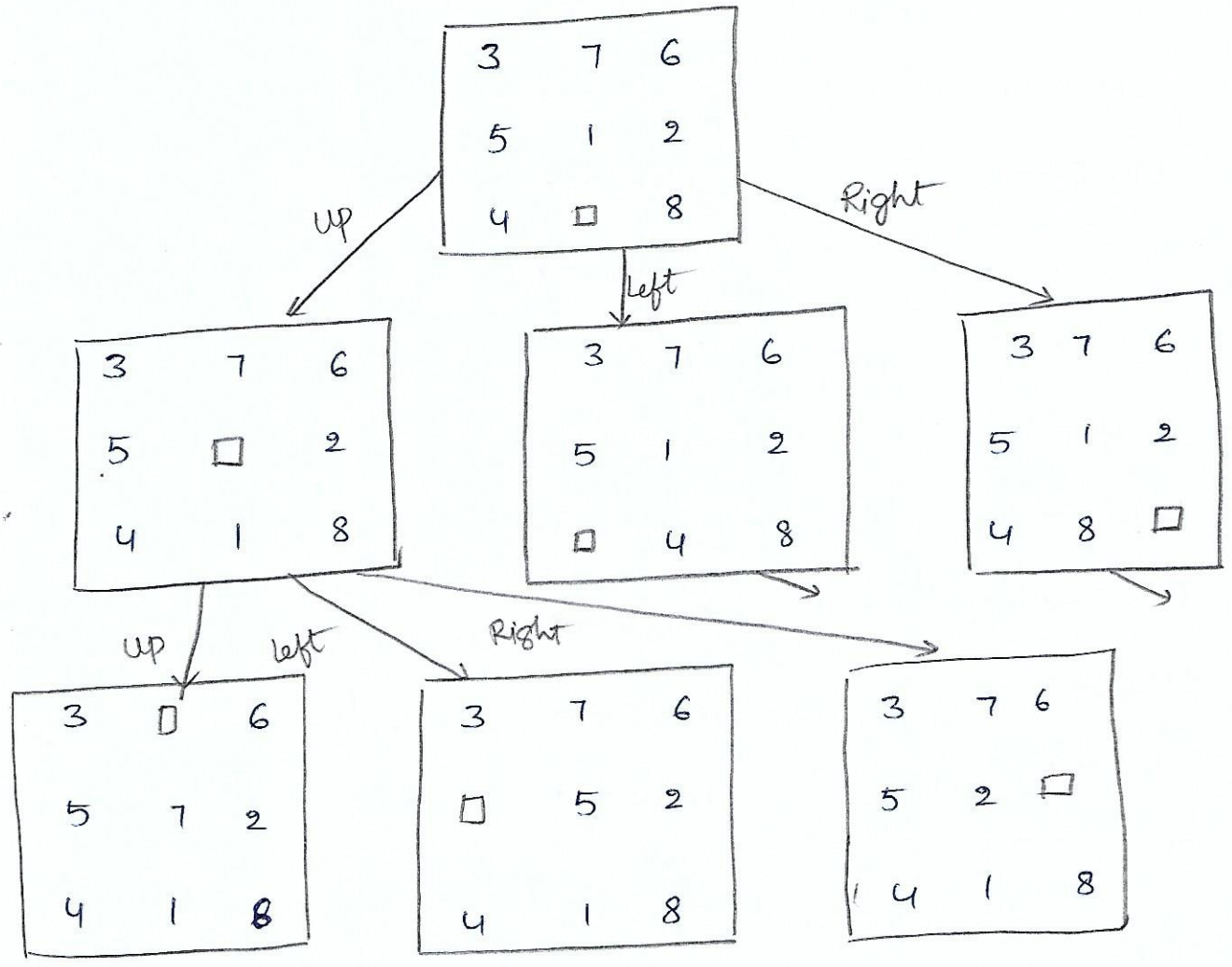
Eight Puzzle Problem:-

Problem Statement: The Eight puzzle problem has 3x3 grid with 8 randomly numbered tiles arranged on it with one empty cell. At any point the adjacent tile can move to the empty cell creating a new empty cell. Solving this problem involves arranging tiles such that we get the state from the start state.

1. Start State : [[3, 7, 6], [5, 1, 2], [4, 0, 8]]

2. Goal State : [[5, 3, 6], [7, 0, 2], [4, 1, 8]]

3. The operators can be thought of moving {up, Down, left, right}, the direction in which blank space effectively moves



Partial Search Tree for Eight Puzzle Problem

Control Strategies:- Control strategy is one of the most important components of problem solving that describes the order of application of the rules to the current state. Control Strategy should be such that it causes motion towards a solution.

For example, in water jug problem if we apply a simple control strategy of starting each time from the top of rule list and select the first applicable one, then we will never move towards solution. The second requirement of control strategy is that it should explore the solution in a systematic manner. This is because control strategy is not systematic.

Depth First Search and Breadth First Search are the systematic control strategies but these are Blind searches. In DFS, we follow a single branch of tree until it yields a solution In BFS, a search space is generated level wise until we find a solution. These strategies are Exhaustive, Uninformed and Blind searches in nature. There are two directions in which such a search should proceed. They are :- 1. Forward Chaining

2. Backward Chaining

Forward chaining:- The process of forward chaining begins with known facts and works towards a conclusion. This is called Data Driven Approach. For example, in eight puzzle problem, we start from the start state and work to the conclusion i.e goal state. This process is continued until a configuration matches the goal state is generated. Language OPS 5 is used for forward chaining.

Backward chaining: It is a goal directed strategy that begins with the goal state and continues working backward, generating more sub goals that must be satisfied to satisfy main goal until we reach to start state. Prolog language uses this strategy

The Computational effort in both strategies is the same. In bot the case, the same state space is searched but in different order. It efficient to use Forward chaining.

We know that from small set of axioms can prove large number of theorems. On the other hand, large number of theorems ca go back to the small set of axioms. Therefore, proving theorem using backward strategy is useful.

Characteristics of a Problem: Before starting modelling the search and trying to find the solutions for the problem, one must analyse it along several key characteristics:

Type of Problem: There are three types of problem in real life i.e Ignorable, Recoverable, Irrecoverable.

Ignorable:- These are the problems where we can ignore the solution steps. For example, in proving theorem, if some lemma is proved to prove a theorem and later we realize it is not useful.

Recoverable:- These are the problems where solution steps can be undone. For example, in water jug problem, if we have filled the jug, we can empty it also. Any state can reached again by undoing the steps.

Irrecoverable:- The problems where solution steps cannot be undone. For example two player games are in this category.

Decomposability of a Problem:- Divide the problem into a set of independent smaller sub problems, solve them and combine the solutions to get final solution. The process of dividing sub problem continues till we get set of the smallest sub problem.

Divide and Conquer technique is the commonly used method for

solving such problems.

Role of knowledge :- Knowledge plays an important role in solving any problem. knowledge could be in the form of rules and facts which help generating search space for finding the solution.

Consistency of knowledge Base used in solving problem : Make sure that knowledge base used to solve problem is consistent. Inconsistent knowledge base will lead to wrong solutions.

For example, If it is humid, it will rain

If it is sunny, then it is day time.

This knowledge is not consistent.

Requirement of Solution : We should analyse the problem whether solution required is absolute or relative. We call solution to be absolute if we have to find exact solution, whereas it is relative if we have reasonably good and approximate solution. For example, in water jug problem there are many ways to solve a problem, therefore we follow one path successfully. The best path problems are computationally harder compared with any path problems.

Exhaustive Searches:- Some systematic uninformed searches are Breadth First Search, Depth First Search, Depth First Iterative Deepening and Bidirectional Searches.

Breadth First Search:- Breadth First search expands all the states one step away from the start state and expands all states two steps from start state then three steps until a goal state is reached. All successor states are examined at the same depth before going deeper. The BFS always gives an optimal path or solution.

Algorithm: (BFS)

Input: START and GOAL states

Local Variables: OPEN, CLOSED, STATE_X, SUCCS, FOUND

Output: Yes or No

Method:

Initialize OPEN list with START and CLOSED = ϕ

FOUND = false

while (OPEN $\neq \phi$ and FOUND = false) do

{

remove the first state from OPEN and call it STATE_X

put STATE_X in the front of CLOSED list

if STATE_X = GOAL then FOUND = true else

{

perform EXPAND operation ON STATE-X, producing succs

remove from successors those states, if any, that are in closed list.

append succs at the end of OPEN list

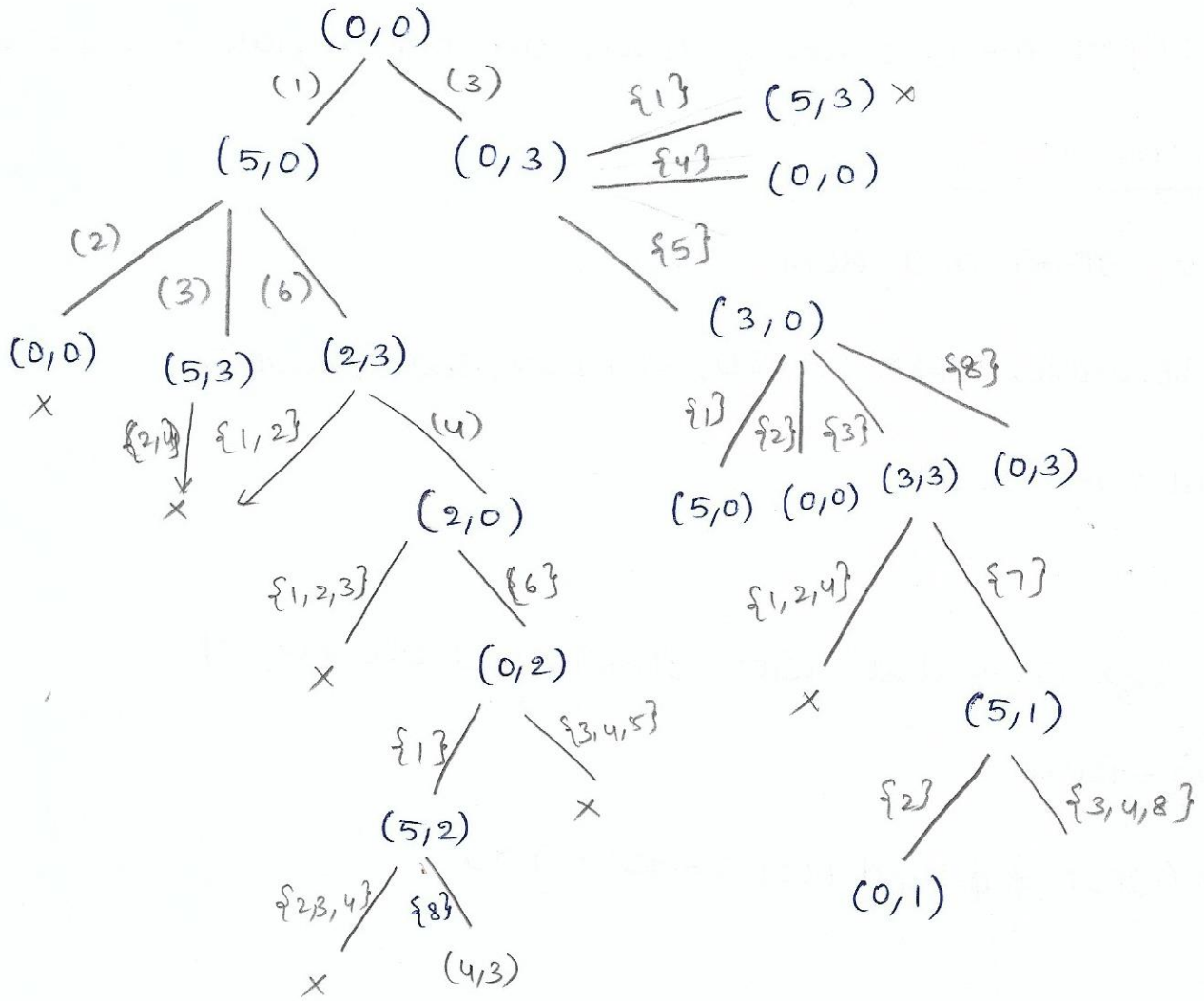
}

}

if FOUND = true then return Yes else return No

Stop.

Example: Water Jug Problem



Depth First Search: In Depth First Search (DFS), we go as far down as possible into the search tree/graph before backing up and trying alternatives. It works by always generating a descendant of the most recently expanded node until some depth cut off is reached and then backtracks to next most recently expanded node and generates one of its descendants. DFS is memory efficient as it only stores a single path from the root to leaf node along with the remaining unexpanded siblings for each node on the path.

Algorithm (DFS):

Input: START and GOAL states

Local Variables: OPEN, CLOSED, RECORD_X, SUCCESSORS, FOUND

Output: A path sequence from START to GOAL state, if one exists
otherwise return NO

```
Begin
  Initialise OPEN list with (START, nil) and set CLOSED = ϕ
  FOUND = false
  while (OPEN ≠ ϕ and FOUND = false) do
  {
    remove the first record from OPEN list and call it RECORD_X
    put RECORD_X in the front of CLOSED list
    if (STATE_X of RECORD_X = GOAL) then FOUND = true else
```

{

perform EXPAND operation on STATE_X producing a list of records called SUCCESSORS : create each record by associating parent link with its state

remove from SUCCESSORS any record that is already in the CLOSED

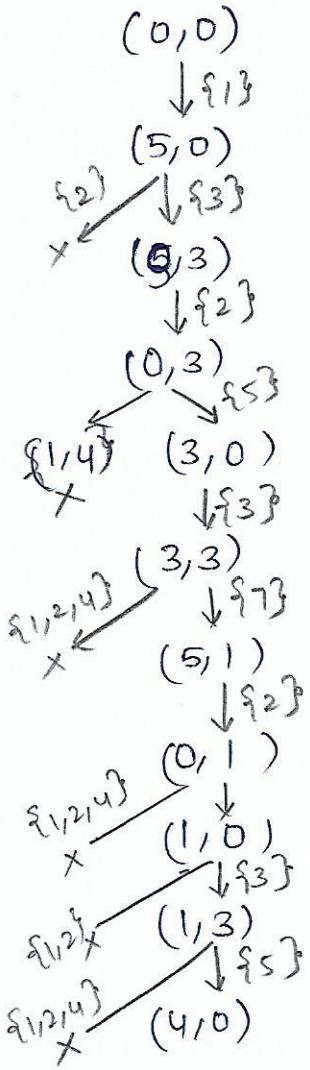
insert SUCCESSORS in the front of the OPEN list

}

}

if FOUND = true then return the path by tracing through the pointers to the parents on the CLOSED list else return NO

stop



Since these are unguided, blind and exhaustive searches, we cannot say much about them but can make some observations

- BFS is effective when the search tree has a low branching factor
- BFS can work even in trees that are infinitely deep
- BFS requires a lot of memory as number of nodes in level of the tree increases exponentially
- BFS is superior when the GOAL exists in the upper right portion of a search tree
- BFS gives optimal solution
- DFS is effective when there are few subtrees in the searchtree that have only one connection point to the rest of the states
- DFS is best when the GOAL exists in the lower left portion of the search tree
- DFS can be dangerous when the path closer to the START and farther from the GOAL has been chosen
- DFS is memory efficient as the path from start to current node is stored. Each node should contain state and its parent
- DFS may not give optimal sdution.

Depth First Iterative Deepening: Depth First Iterative Deepening takes advantages of both BFS and DFS searches on trees

Algorithm (DFID):

Input : START and GOAL states

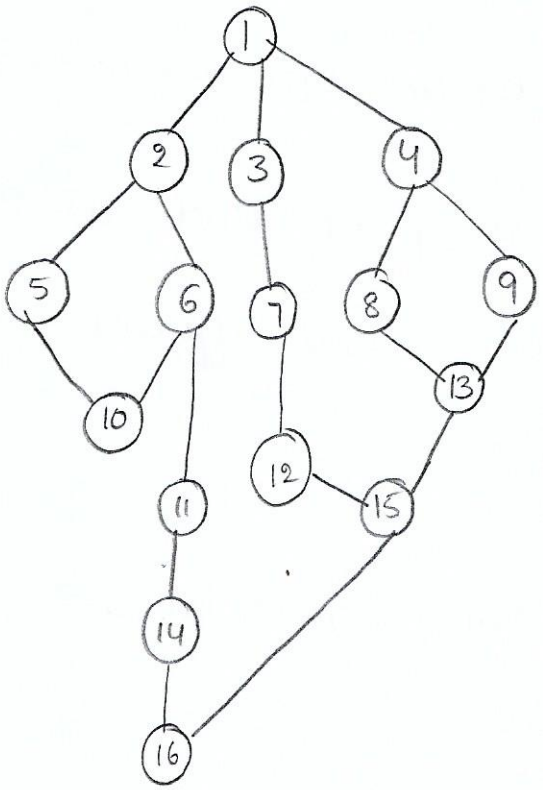
Local Variables : FOUND

Output : Yes or No

```
Begin
 Initialise  d=1   FOUND = false
 while ( FOUND = false) do
 {
   perform a depth first search from start to depth d
   if goal state is obtained then FOUND = true else discard nodes
   generated in the search of depth d
   d = d + 1
 }
 if FOUND = True return Yes otherwise return No
End
```

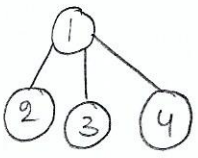
Disadvantage of DFID is that it performs wasted computation before reaching the goal depth

Bidirectional Search :- Bidirectional Search is a graph search algorithm that runs two simultaneous searches. One search moves forward state and other moves backward from the goal and stops when two meet in the middle. It is useful for those problems which have a single start state and single goal state

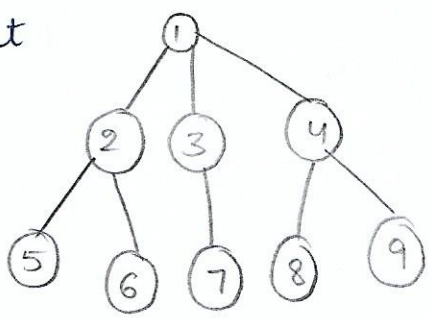


K = 0 Start (1)

K = 1 Start

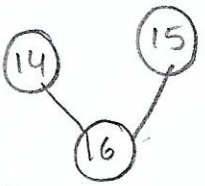


K = 2 Start

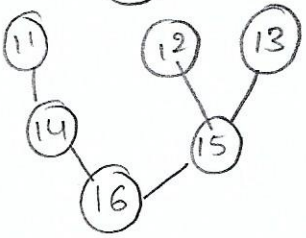


K = 0 Goal (16)

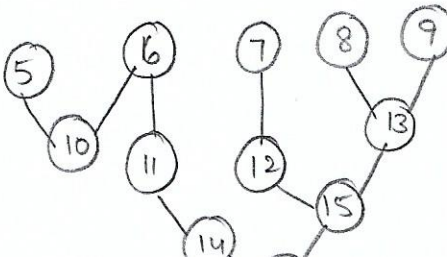
K = 1 Goal



K = 2 Goal



K = 3 Goal



Analysis of Search Methods:- Effectiveness of any search strategy in problem solving is measured in terms of:

Completeness:- Completeness means that an algorithm guarantees a solution if it exists

Time Complexity: Time required by an algorithm to find a solution

Space Complexity: Space required by an algorithm to find a solution

Optimality: The algorithm is optimal if it finds the highest quality solution when there are several different solutions for the problem.

Let us assume 'b' to be the branching factor and 'd' to be the depth of the tree in the worst case.

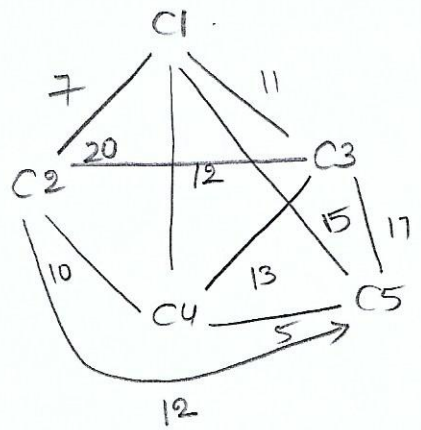
	BFS	DFS	DFID	Bidirectional
Completeness:	✓	X	✓	✓
Complexity Time:	$O(b^d)$	$O(b^d)$	$O(b^d)$	$O(b^{d/2})$
Space Complexity	$O(b^d)$	$O(d)$	$O(d)$	$O(d/2)$
Optimality	Yes	No	Yes	No

Heuristic Searches:-

Travelling Sales Man Problem:

Statement: In Travelling Salesman Problem, one is required to find the shortest route visiting all the cities once and returning back to starting point. Assume that there are 'n' cities and the distance between each pair of cities is given.

The problem seems to be simple, but deceptive. TSP is one of the most intensely studied problems in computational mathematics



Some of the partial paths are pruned if the distance computed is less than minimum computed distance so far between any pair of cities

Some paths

C1 → C2 → C3 → C4 → C5 → C1

7 20 13 5 15

27 40 45 60

C1 → C2 → C4 → C5 → C3 → C1

7 10 5 17 11 50 Minimum distance

17 22 39 50

Heuristic Search Techniques:- Heuristic Search Technique is a criteria for determining which among several alternatives will be most effecti to achieve some goal.

This technique improves the efficiency of a search process possibly by sacrificing claims of systematic and completeness.

It is no longer guarantees to find the best solution but almost always finds a very good solution using boot heuristics we hope to get solutions to hard problems (such as TSP) in less than exponential time.

There are two types of Heuristics namely

1. General purpose Heuristics that are useful in various problem domain
2. Special purpose Heuristics that are useful in Domain Specific

General Purpose Heuristics:- A general purpose heuristic for combinatio -al problems is nearest neighbour algorithm that work by selecting locally superior alternative. For such algorithms, it is often possible to prove an upper bound in the error.

→ In many AI problems, it is often difficult to measure precisely the goodness of a particular solution.

→ For real world problem, it is often useful to introduce heuristic on the basis of relatively unstructural knowledge

→ It is impossible to define this knowledge in such a way that mathem-
-atical analysis can be performed

→ However, it is important to keep performance in mind while designing algorithms one of the best/most important analysis of the search process is to find number of nodes in a complete search tree of depth 'd' and branching factor 'b' i.e $b \times d$

→ This simple analysis motivates to look for improvements on the exhaustive searches and to find upper bound on search tree

→ The searches which use some domain knowledge are called Informal Search strategies

a) Branch and Bound Search (Uniform Cost Search): In this search method, cost function ($g(x)$) is designed that assigns cummulative response to the path from start node to current node 'x' by applying the sequence of operators. while generating a search space, a least cost path obtained so far is expanded at each iteration till we reach to goal state. Since branch and bound search expands the least cost partial path it is sometimes also called Uniform Cost Search

Algorithm (Branch and Bound):

Input: START and GOAL states

Local Variables: OPEN, CLOSED, NODE, SUCCs, FOUND

Output: Yes or No

```
Begin
   initially store the start node with g(root) = 0 in OPEN list.
   CLOSED = ϕ  FOUND = False
   while (OPEN ≠ ϕ and FOUND = False) do
   {
     remove the top element from OPEN list and call it NODE
     if NODE is goal node then FOUND = true else
     {
       put NODE in CLOSED list
       find SUCCs of NODE, if any compute their 'g' values and
       store them in OPEN list
       sort all the nodes in the OPEN list based on their cost-function
       values
     }
   }
  if FOUND = True then return Yes otherwise return No;
End
```

In Branch and Bound if $g(x) = 1$ for all operators then it generates a simple BFS. From AI point of view, it is as bad as DFS and BFS. Th can be improved if we augment it by Dynamic programming i'e delete those paths which are redundant. We notice that algorithm gen -ally requires generate solution and test it for its goodness. Solutio can be generated using any method and testing might be based on some heuristics

Algorithm (Generate and Test):

Start

Generate a possible solution

Test if it is a goal

if not go to start else quit

Stop

Hill Climbing: Quality Measurement turns DFS into Hill Climbing (A variant of generate and test strategy). It is an optimisation techniq that belong to family of local searches. It is relatively simple technique to implement as a popular first choice is explored. Hill Climbing can be used to solve problems that have many solutions but where some solutions are better than others

Algorithm (Hill Climbing):

Input: START and GOAL states

Local Variables: OPEN, NODE, SUCCs, FOUND

Output: Yes or No

Begin

store initially the start node in OPEN list, FOUND = false

while(OPEN ≠ empty and FOUND = false) do

{

remove the top element from OPEN list and call it NODE

if NODE is the goal node then FOUND = true else

find SUCCs of NODE if any

sort SUCCs by estimated cost from NODE to goal state and

add them to the front of OPEN list

}

if FOUND = true then return Yes otherwise return No

End

Problems with Hill Climbing: In the search process, we may reach to the position that is a solution but from there no move improves the situation. This will happen if we reached local maximum, a plateau or a ridge

Local maximum: It is a state that is better than all its neighbours but not better than some of the states which are far away. From this states all moves looks to be worse. In such situation, backtrack to some earlier state and try going in different direction to find a solution.

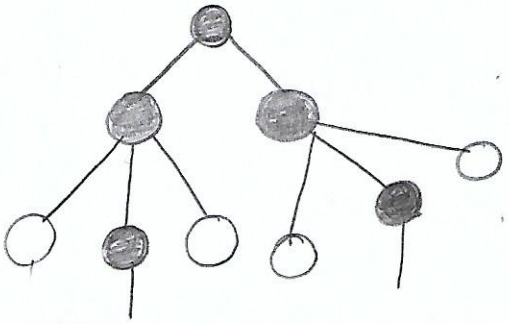
Plateau :- It is a flat area of the search space where all neighbouring states that has some value. It is not possible to determine the best direction. In such a situation, make a big jump to some direction and try to get to new section of the search space.

Ridge: It is an area of search space that is higher than surrounding area but that cannot be traversed by single moves in any direction. It is a special kind of local maxima. We move two or more rules before doing test. i.e moving in several directions at once.

Beam Search:- It is a heuristic search algorithm in which 'w' number of best nodes at each level always expanded. It progresses level by level and moves down only from the best 'w' nodes at each level.

Beam search uses BFS to build its search tree. At each level of the tree it generate all successors of the states at the current level and sorts them in order of increasing heuristic values. However,

we only considers a 'w' number of states at each level. Here 'w' is called width of Beam Search. If w=1 then it becomes Hill climbing search. If beam width is infinite then no states are pruned and beam search is identical to BFS.



Algorithm (Beam Search):

Input: START and GOAL states

Local Variables: OPEN, NODE, SUCCs, W_OPEN, FOUND

Output: Yes or No

Begin

NODE = Root_node, FOUND = False

if NODE is the goal node then FOUND = true else find SUCCs of NODE if any with its estimated cost and store in OPEN list

while (FOUND = False and not able to proceed further) do

{

SORT OPEN list

Select 'w' top element from OPEN list and put it in W_OPEN list and Empty OPEN list

for each NODE from W_OPEN list

{

if NODE = Goal state then FOUND = true else find SUCCs of NODE, if any with its estimated cost and store in OPEN list

} }

if FOUND = true then return Yes otherwise return No

End

Best First Search:- Best First search is based on expanding the best partial path from current node to goal node. Here forward motion is from the best open node so far in the partial developed tree. The cost of ~~popular~~ partial paths is calculated using some heuristic. If the state has been generated earlier and the new path is better than the previous one then change the parent and update the cost. It should be noted that in Hill Climbing sorting is done on the successor nodes whereas in the Best First Search, sorting is done on the entire list. It is not guaranteed to find an optimal solution but generally finds some solution faster than solution obtained from any other method.

Algorithm (Best First Search):

Input: START and GOAL states

Local Variables: OPEN, CLOSED, NODE, FOUND

Output: Yes or No

```
Begin
  Initialise OPEN list by root node and CLOSED = ϕ, FOUND = False
  while (OPEN ≠ ϕ and FOUND = False) do
  {
    if the first element is GOAL Node then FOUND = true else
```

remove it from OPEN list and put it in CLOSED list

Add it successor if any in OPEN list

Sort the entire list by the value of some heuristic function to assign to each node. The estimate to reach to the goal node

}

if FOUND = True then return Yes otherwise return No

End

A * Algorithm:- It is combination of Branch and Bound and Best First Search methods combined with dynamic programming principles. It use Heuristic evaluation function usually denoted by $f(x)$ to determine the order in which the search the nodes in the tree. The Heuristic function for a node x is defined as follows: $f(x) = g(x) + h(x)$

The function 'g' is the measure of the cost of getting from start node to current node 'x'

The function 'h' is the estimate of additional cost of getting from current node 'x' to goal node. This is a place where about the problem domain is exploited. Generally A * algorithm is called

OR Graph/Tree

Algorithm (A*):

Input: START and GOAL states

Local Variables: OPEN, CLOSED, Best_Node, SUCCs, OLD, FOUND

Output: Yes or No

```
Begin
  Initialise OPEN list with START node, CLOSE = ϕ , g = 0, f = h, FOUND = Fa
  while (OPEN ≠ ϕ and FOUND = False) do
  {
    remove the node with the lowest value of f from OPEN list and store
    it in CLOSED list. Call it as a Best_Node
    if (Best_Node = Goal_state) then FOUND = true else
    {
      generate the SUCCs of the Best_Node
      for each succ do
      {
        establish parent link of succ
        Compute g(succ) = g(Best_Node) + cost of getting from Best_Node to
                                                                    succ

        if (succ ∈ OPEN) then
        {
          call the matched node as OLD and add it in the successor
          list of the Best_Node

          ignore the succ node and change the parent of OLD if
```

required as follows:

if g(succ) < g(OLD) then make parent of OLD to be Best_Node and change the value of g and f for OLD else ignore;

}

if (SUCC ∈ CLOSED) then

{

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

Ignore the succ node and change the parent of OLD, if required as follows:

if g(succ) < g(OLD) then make parent of OLD to be Best_Node and change the values of g and f for OLD and propagate the change to OLD's children using DFS else ignore

}

if SUCC ≠ OPEN or CLOSED

{

Add it to the list of Best Node's successors;

compute f(succ) = g(succ) + h(succ)

Put succ on OPEN list with its f value

} }

} }

}

if FOUND = true then return Yes otherwise return No

End

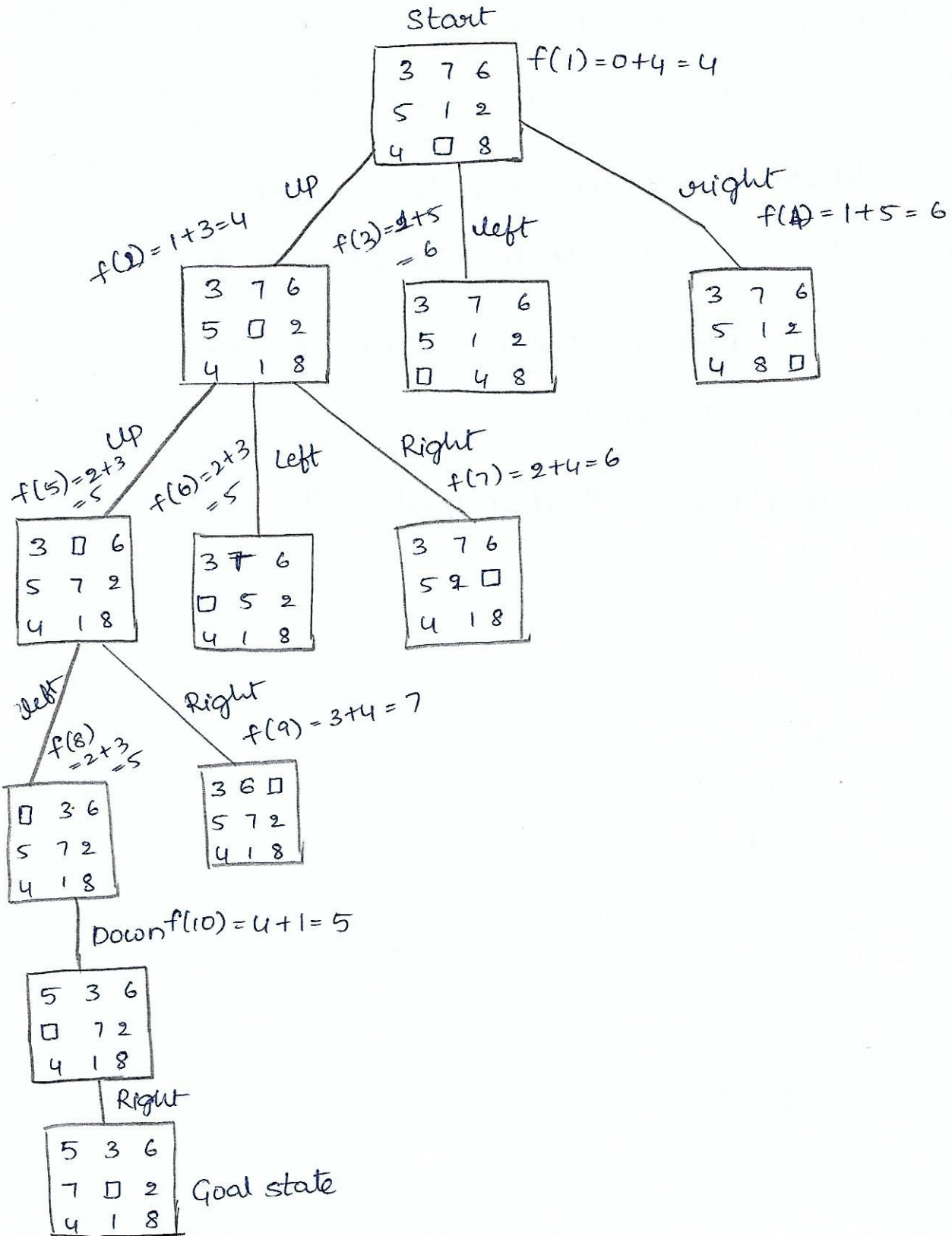
Let us consider an example of Eight puzzle and solve it using A* Algor

The Evaluation function $f(x)$ is defined as follows:

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

where $h(x)$ = number of tiles not in their goal position in given state

$g(x)$ = depth of node x in search tree



Iterative Deepening A*:- Iterative Deepening A* is a combination of Depth First Iterative Deepening and A* algorithm. Here successive iterations are corresponding to increasing values of the total cost of the path rather than increasing the depth of the search. This algorithm works as follows:

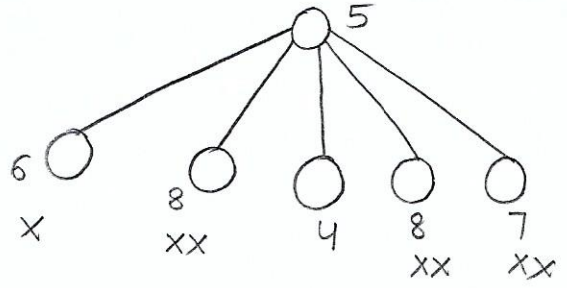
1. For each iteration, perform a DFS pruning off a branch when its total cost (g+h) exceeds a given threshold

2. The initial threshold starts at the estimate cost of the start state and increases for each iteration of the algorithm.

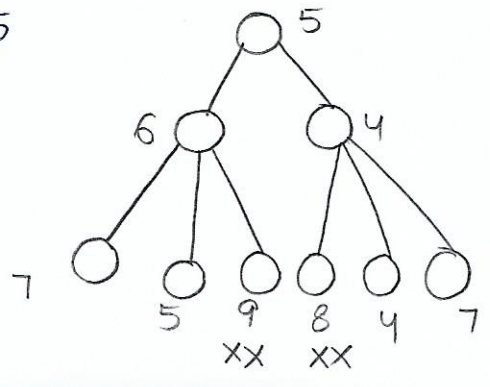
3. The threshold used for the next iteration is the minimum cost of all values exceeded the current threshold

4. These steps are repeated till we find a goal state

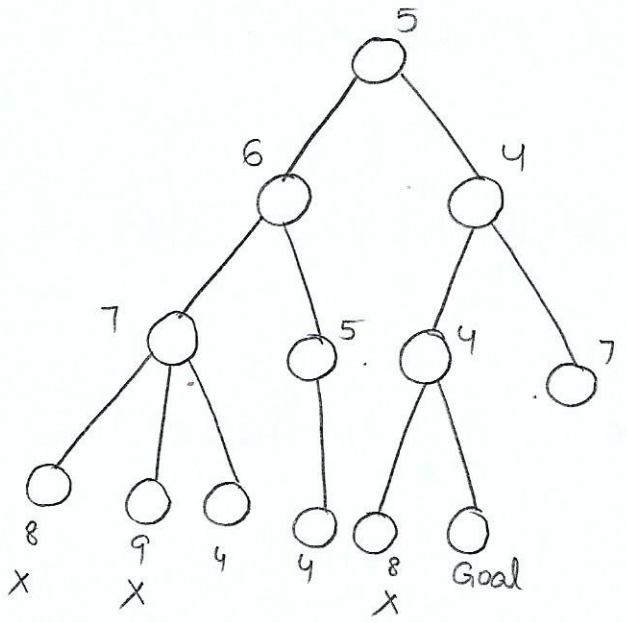
1st threshold : 5



2nd Threshold : 5



3rd Threshold : 7



Constraint Satisfaction Problem (CSP) :- Many AI problems can be viewed as problems of constraint satisfaction in which the goal is to solve some problem state that satisfies a given set of constraints instead of finding optimal path to the solution. Such problems are called Constraint Satisfaction Problems.

For example, some of the CSP are

1. Cryptography
2. N-Queens
3. Map Coloring
4. Crossword Puzzle.

Constraint satisfaction is defined as follows:

i, $\{x_1, x_2 \ldots x_n\}$ $x_i \in D_i$ where each $x_i \in D_i$ with possible values.

A set of constraints i.e relations that are assumed to hold between the values of the variables. A CS problem is

usually represented as an undirected graph called Constraint graph in which the nodes are the variables and the edges are binary constraints.

Crypt Arithmetic: The CSP search problem is formulated as follows:

Start state: The empty assignment i.e all variables are unassigned

Goal state: All variables are assigned values which satisfy constraints

Operator: Assigns values to any unassigned variable, provided that it doesn't conflict with previously assigned variables.

Algorithm (CSP):

Begin

until complete solution is found or all paths have lead to values

{

select an unexpanded node of the search graph

Apply the constraint inference rules to the selected nodes that generates all possible new constraints

if set of constraints contain contradiction then report that path is dead end

if set of constraints describes a complete solution then

if neither a contradiction nor a complete solution has been found then apply problem space rules to generate new path solutions that are consistent with the current set of constraint

Insert these partial solutions into search graph

}

End.

Crypt Arithmetic Puzzle:

Problem: Solve the following puzzle by assigning numeral (0 to 9) in such a way that each letter is assigned unique digit which satisfy the following addition

```
    B A S E
+   B A L L
-------------
  G A M E S
```

```
   C4 C3 C2 C1
    B A S E
+   B A L L
-------------
  G A M E S
```

1. G = C4 = 1

2. 2B + C3 = A

   2B + C3 > 9 [B possible values 5 to 9]

$B = 5$

- $C3 = 0 \Rightarrow A = 0$
  - $M = 0$ if $C2 = 0$
  - $M = 1$ if $C2 = 1$
- $C3 = 1 \Rightarrow A = 1$ x ($\because G = 1$)

$B = 6$

- $C3 = 0 \Rightarrow A = 2$
  - $M = 4$ if $C2 = 0$
  - $M = 5$ if $C2 = 1$
- $C3 = 1 \Rightarrow A = 3$ x

$B = 7$

- $C3 = 0 \Rightarrow \boxed{A = 4}$
  - $M = 8$ if $C2 = 0$
  - $\boxed{M = 9}$ if $C = 1$
- $C3 = 1 \Rightarrow A = 5$
  - $M = 0$ if $C2 = 0$
  - $M = 1$ if $C2 = 1$

3. $S + L + C1 = E \longrightarrow (a)$

$E + L = S \longrightarrow (b)$

$E + L + L + C1 = E$

$2L + C1 = 0$

$C1 = 0$

$L = 5$

$A = 4$

$M = 9$

$B = 7$

$E + L = S$

$E + 5 = S$

$E = 2 \Rightarrow S = 7$

$E = 3 \Rightarrow S = 8$

$$\begin{array}{cccccc} & 1 & 0 & 1 & 0 & \\ & 7 & 4 & 8 & 3 & \\ & 7 & 4 & 5 & 5 & \\ \hline 1 & 4 & 9 & 3 & 8 & \\ \hline G & A & M & E & S & \end{array}$$