Introduction to search - Artificial Intelligent UNIT-2ND

PRASHANT TOMAR

Problem solving and search

In intelligent agents knowledge base corresponds to environment, operators correspond to sensors and the search techniques are actuators. Hence, three parameters of an intelligent are….

1. Knowledge base
2. Operators
3. Control strategy(search technique)

The knowledge base describes the current task domain and the goal. In other words ,goal is nothing but the state. Operator are manipulate the knowledge base .

The control strategy decides what operators to apply and where. The aim of any search technique is the application of an appropriate sequence of operators to initial state to achieve the goal .

The objective to get the goal state : The objective can be achieved in two ways.

Forward reasoning: It refer to the application of operators to those structure in the knowledge base that describe the task domain in order to produce modified state . Such a method is also referred to as bottom up and data driven reasoning.

Backward reasoning: It break down the goal(problem) statement into sub goals which are easier to solve and whose solution are sufficient to solve original problem.

A problem solving agent or system uses either forward or backward reasoning. Each of its operators works to produce a new state in the knowledge base which is which is said to represent problem in a state space.

Problem Formulation: The problem can be defined formally by four component:

i. **The Initial state :**The start state

ii. **State space**: It involve a description of all the possible action available , i.e. , the set of all state s reachable from the initial state . the state space forms a graph in which the nodes are states and the arcs b/t node are actions .

iii. **Goal Test:** It determines whether a given state is goal state .

iv. **Path cost**: It is a function that assign a numeric cost to each path.

A solution to a problem is a path from the initial state to a goal path

Any problem can be solved by the following series of step:

1. Define a state space which contain all the possible configuration of the relevant object.

2. Specify one or more states within that space which would describe possible situation from which find out the initial states.

3. Specify one or more states which would be acceptable as solution to the problem .the state are called goal states.

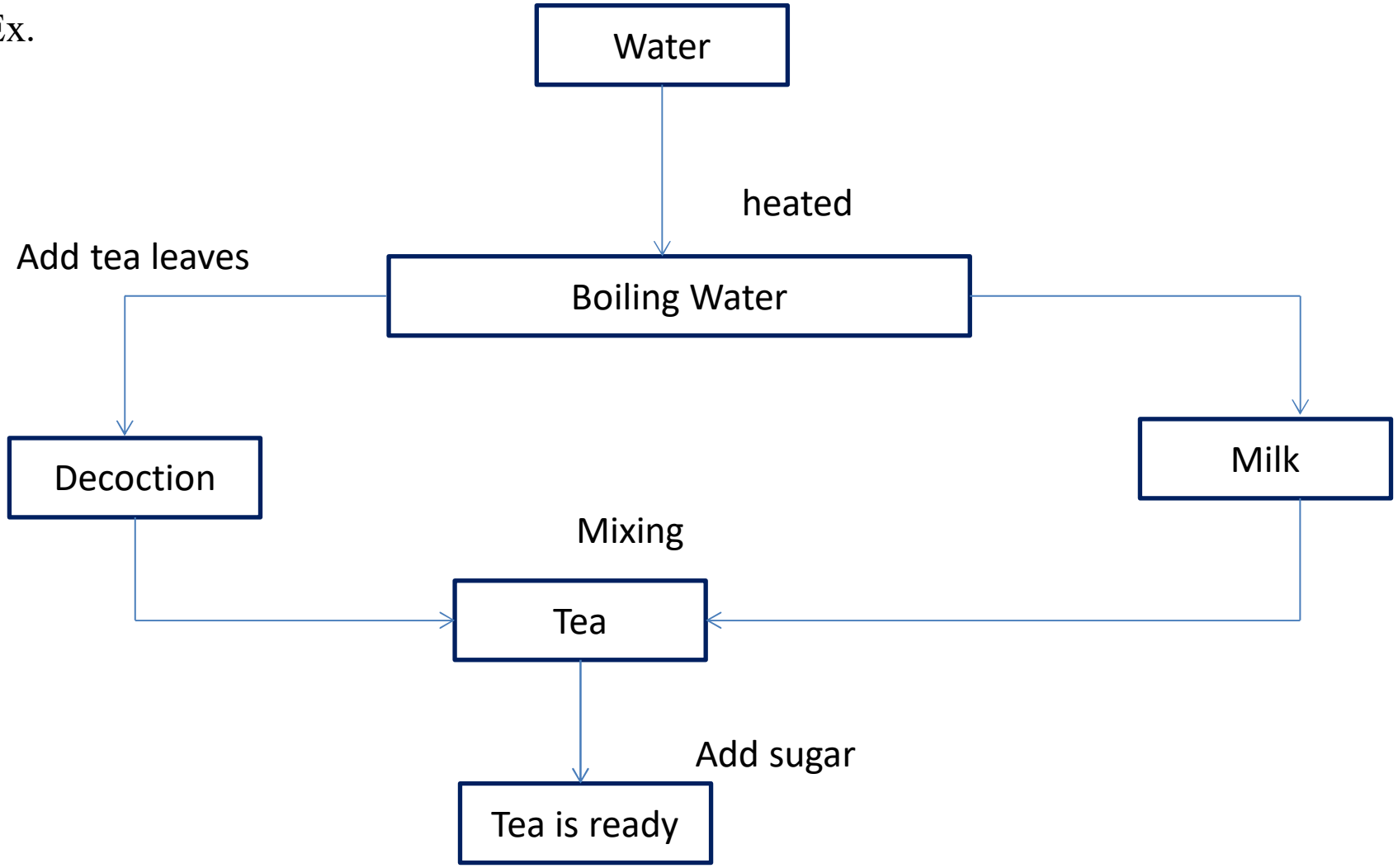
4. Specify a set of rules which describe the actions (operators) available and a control strategy to decide the order of application of these rules.

The most common methods of problem solving representation in AI..

1. State space representation
2. Problem reduction

State space Representation: Set of all possible state is known as the state space of the problem.

Ex.



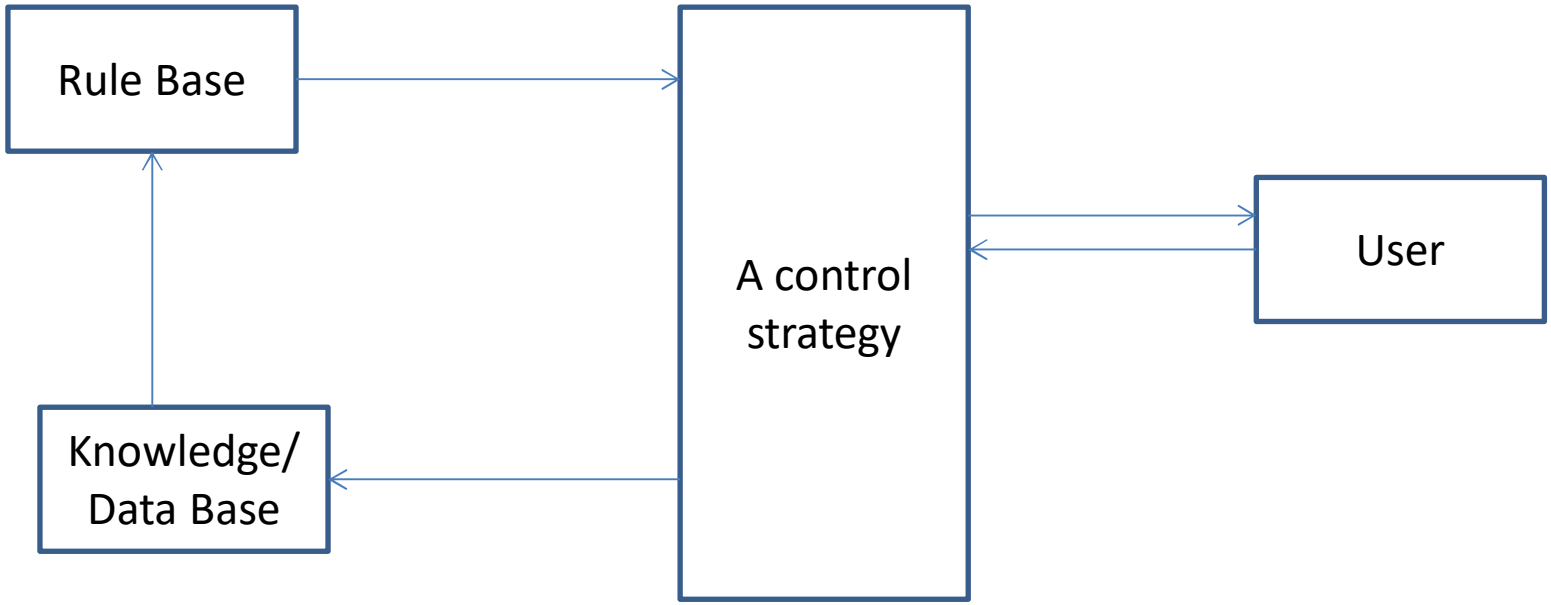
State space representation for tea making

Production System: The main function of the production system is to provide a useful tool for problem solving in A.I. Production System are Frequently referred to as inferential system or rule base –based system or production system.

It is useful to structure AI problems in a way that facilitates describing & performing the search process. Production systems provide these structures. A production system consists of following components----

1. **A set of rules** each consisting of a left side that determines the acceptability of the rule and a right that describes the operation to be performed if the rule is applied.
2. One or more **knowledge bases** that contain whatever information are appropriate for a particular task.
3. **A control strategy** that specifies the order in which in which the rules will be compared to the database and a way of resolving the conflicts that arise when several rules match at once.
4. **A rule applier(user**).

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



Block diagram of production system

Advantages of production systems:-

1. Production systems provide an excellent tool for structuring AI programs.

2. Production Systems are highly modular because the individual rules can be added, removed or modified independently.

3. The production rules are expressed in a natural form, so the statements contained in the knowledge base should the a recording of an expert thinking out loud.

Disadvantages of Production Systems:-

One important disadvantage is the fact that it may be very difficult analyze the flow of control within a production system because the individual rules don’t call each other.

AI Problem:

1. **Water Jug Problem**
2. **Playing chess**
3. **8-puzzle problem**
4. **Tic-tac –toe problem**
5. **8-queen problem**
6. **The Tower of Hanoi problem**
7. **The missionaries and cannibals problem**

Water Jug Problem: you are two jugs ,a 4-gallon and a 3- gallon one . Neither has any measuring markers on it .there is pump that can be used to fill the jugs with water. How can you exactly 2 gallons of water into the 4-gallon jug .Solve the problem using production Rules.

To solve the problem, we define rules whose left sides are matched against the current state and whose right sides describe the new state that results from applying the rule. The assumptions are:

1. We can fill a jug from pump.
2. We can pour water out of the jug onto the ground.
3. We can pour water from one jug to another.
4. There are no measuring devices available.

To solve the water jug problem, we need a control structure that loops through a simple cycle in which some rule whose left side matches the current state is chosen, and the resulting state is checked if it corresponds to the goal state. As long it does not, the cycle continues.

Initial state: (0,0) start state.

State Space: The state space for this problem can be described as the set of ordered pair of integer(X,Y) such that X=0,1,2,3,0r 4 and Y=0,1,2,0r 3. X represent the number of gallons of water in the 4-gallon jug and Y represent the quantity water in the 3-gallon jug.

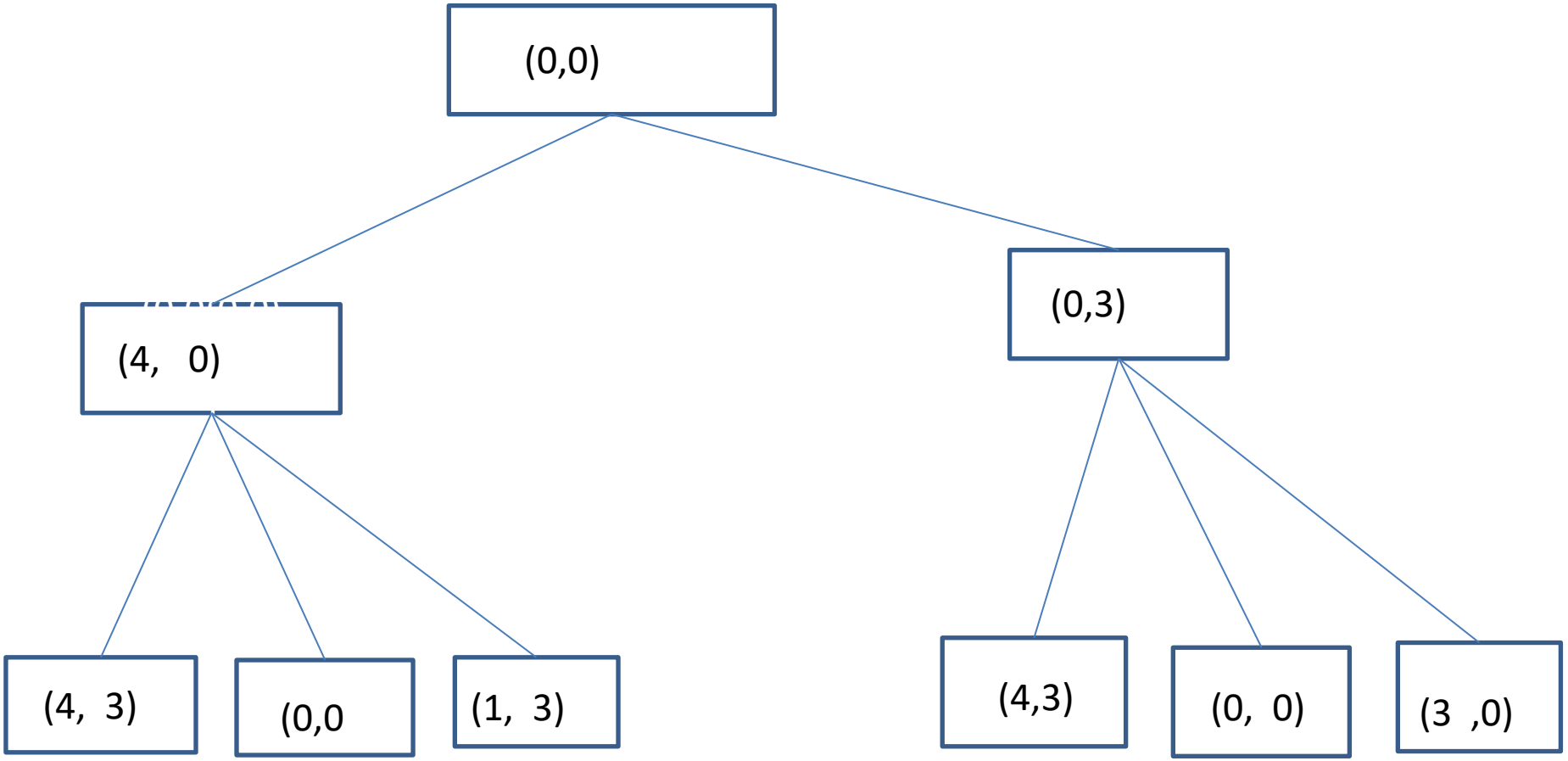
Production rules for water jug problem:

1.(X, Y) If(x<4)	-	(4, Y)	Fill the 4-gallon jug
2. (X, Y) if y<3	-	(X, 3)	Fill the 3-gallon jug
3.(X, Y) if x>0	-	(X-d, Y)	Pour some water out of 4-gallon jug
4.(X, Y) if y>0	-	(X, Y-d)	Pour some water out of 3-gallon jug
5.(X, Y) if x>0	-	(0,Y)	Empty the 4-gallon jug on the ground
6. (X, Y) if y>0	-	(X, 0)	Empty the 3-gallon jug on the ground

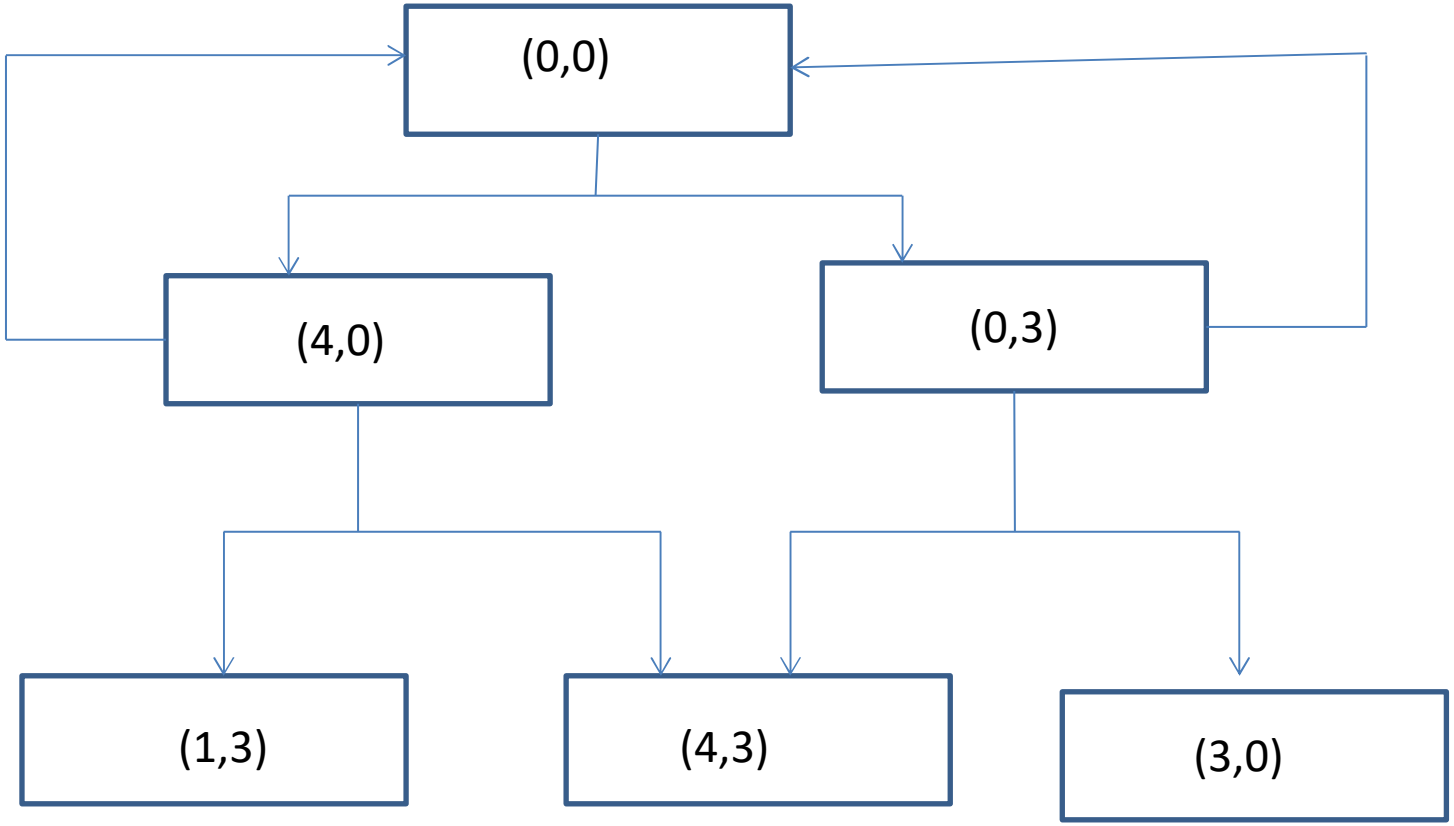
7.(X, Y) if x+y >=4 and y>0	- (4, y- (4-x))	Pour water from the 3 gallon jug into 4 gallon jug until 4-gallon jug is full
8. (X, Y) x+y>=3 and x>0	- (X- (3-y), 3)	Pour water from 4-gallon jug into 3- if gallon jug until 3-gallon jug is full
9.(X, Y) if x+y<=4 and y>0	- (X+Y, 0)	Pour all the water from the 3-gallon jug into the 4-gallon jug
10. (X, Y) if x+y<=3 and x>0	- (0, x+y)	Pour all the water from the 4 gallon jug into the 3-gallon jug
11.(0, 2)	-(2, 0)	Pour the 2-gallons from the 3-jug gallon into 4-gallon jug
12. (2, y)	- (0,y)	Empty the 2-gallons in the 4-gallon jug on the ground

Solution for water jug problem:

4-gallon jug	3-gallon jug	Rule applied
0	0	-
0	3	2
3	0	9
3	3	2
4	2	7
0	2	5 or 12
2	0	9 or 12



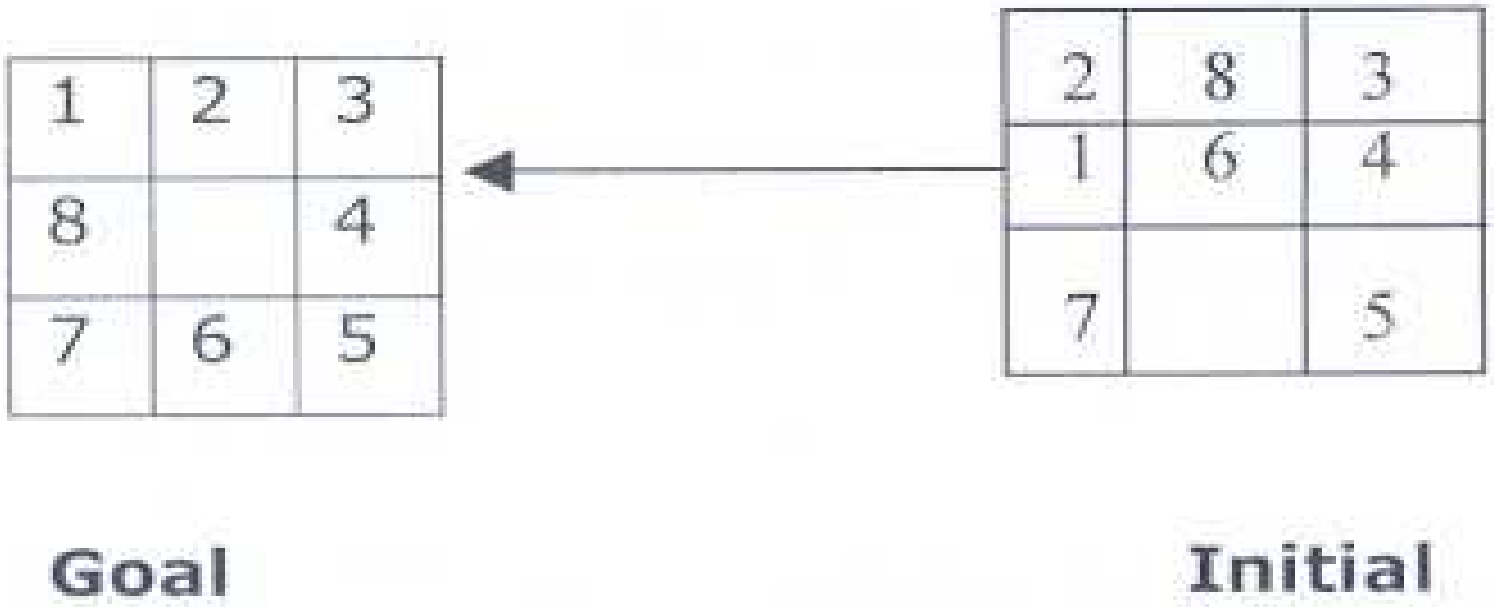
A search tree of a water jug problem



A search graph for the water Jug problem

8 Puzzle Problem.

The 8 puzzle consists of eight numbered, movable tiles set in a 3x3 frame. One cell of the frame is always empty thus making it possible to move an adjacent numbered tile into the empty cell.



The state space representation:

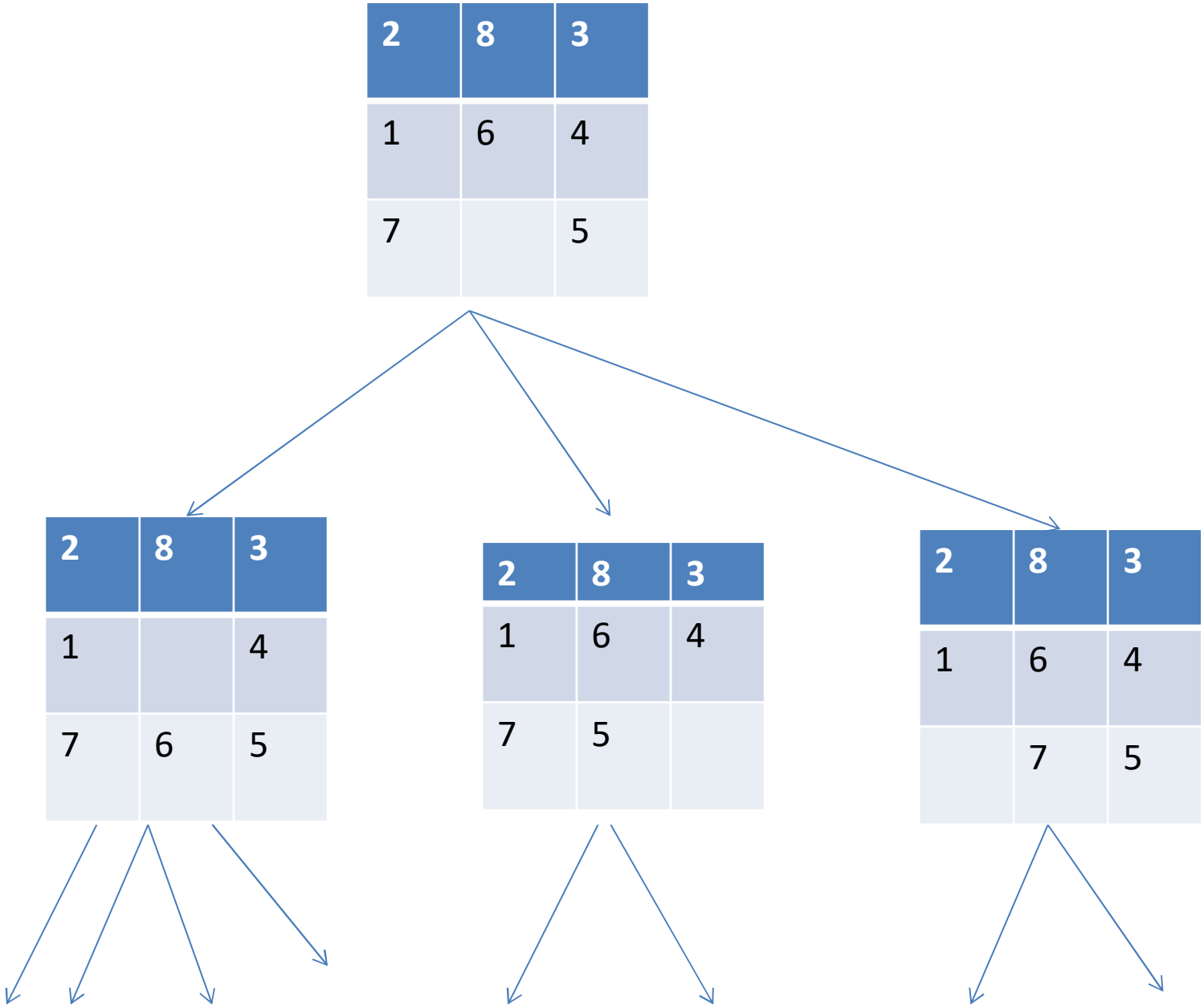
States space: A state is a description of each of the eight tiles in each location that it can occupy.

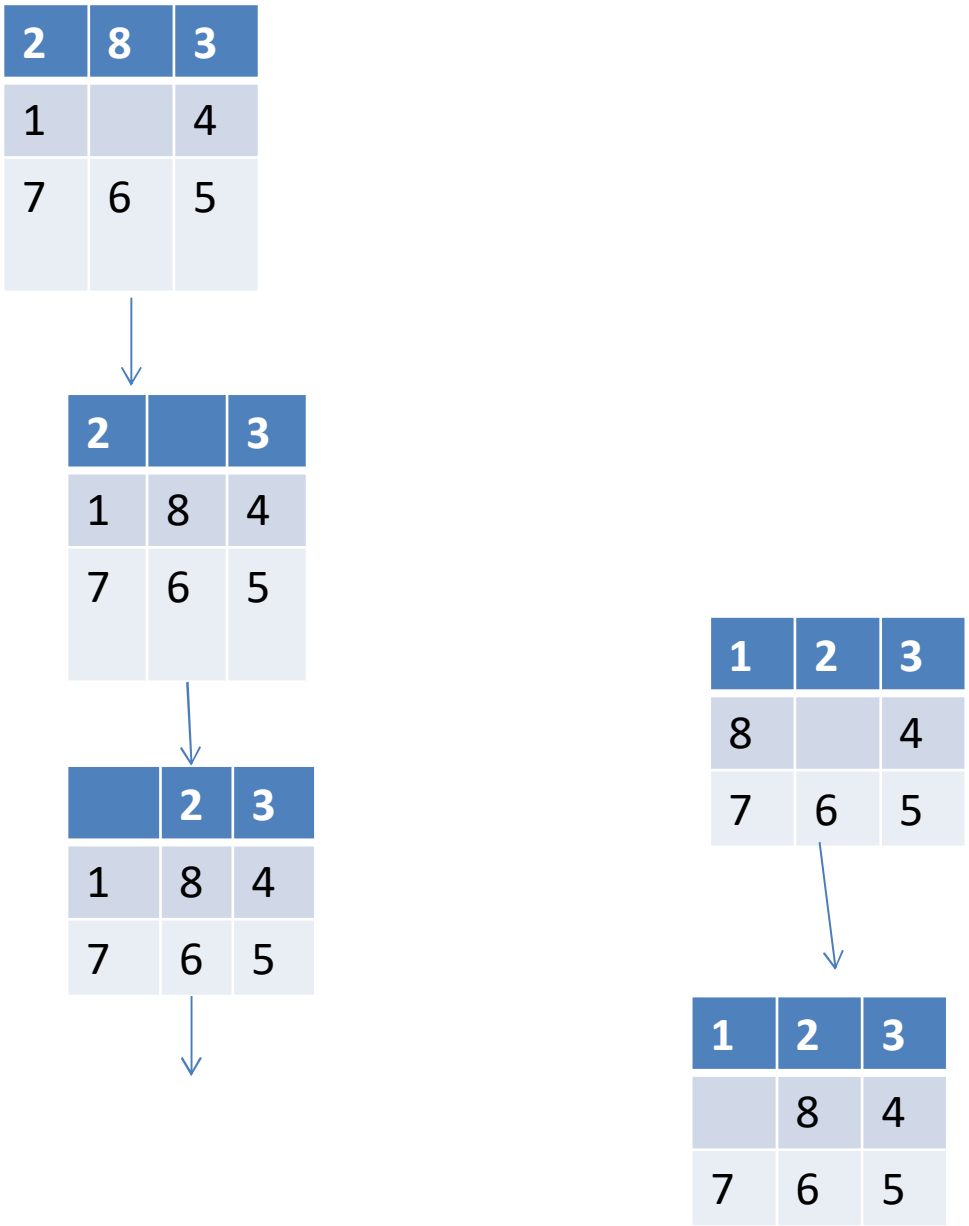
Operators/Action: The blank moves left, right, up or down

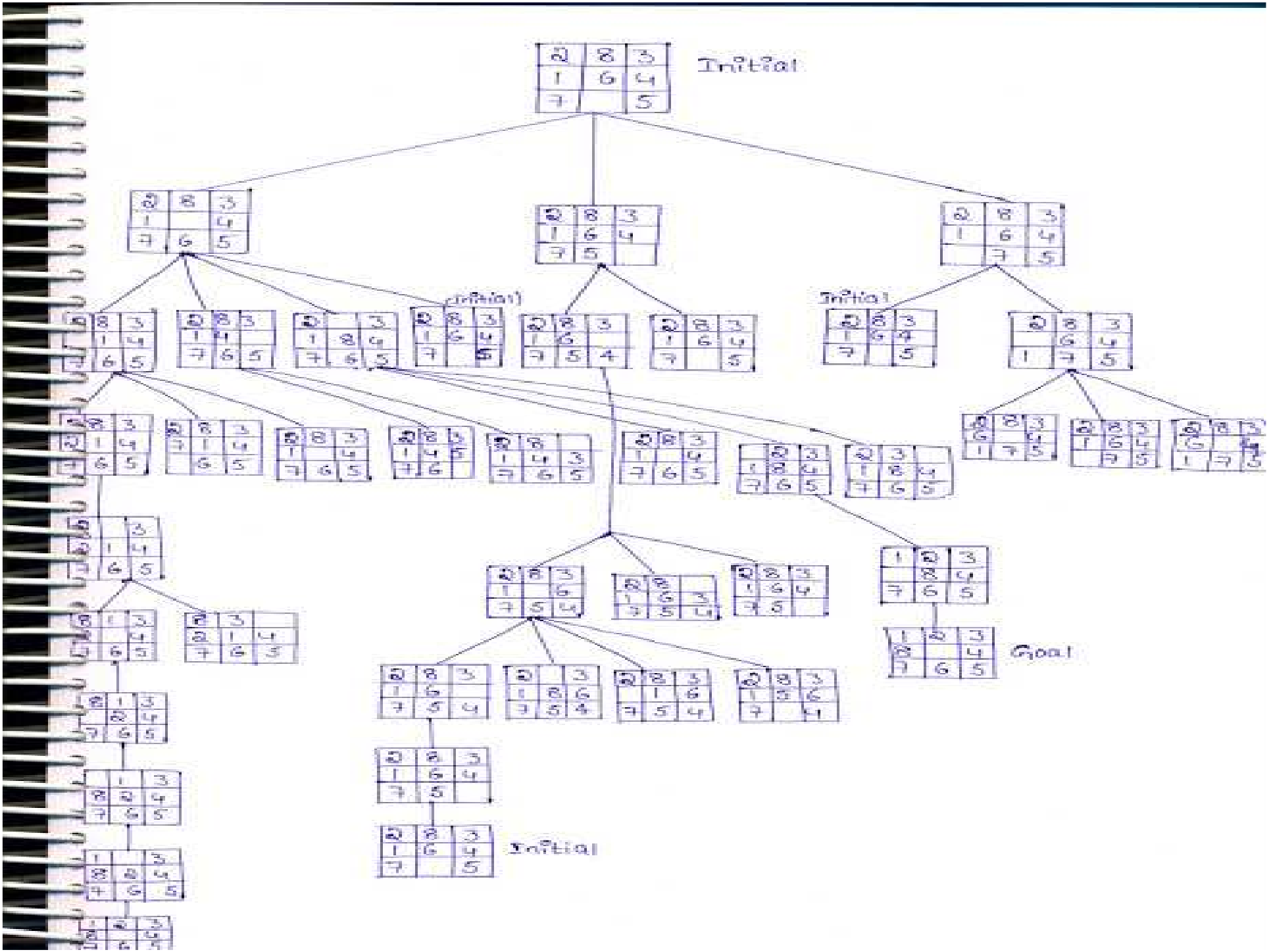
Goal Test: The current state matches a certain state

Path Cost: Each move of the blank costs 1

Note that we do not need to generate all the states before the search begins. The states can be generated when required.


2 8 3
1 6 4
7 5
2 8 3
1 4
7 6 5
2 8 3
1 6 4
7 5
2 8 3
1 6 4
7 5


2 8 3
1 4
7 6 5
2 3
1 8 4
7 6 5
2 3
1 8 4
7 6 5
1 2 3
8 4
7 6 5
1 2 3
8 4
7 6 5

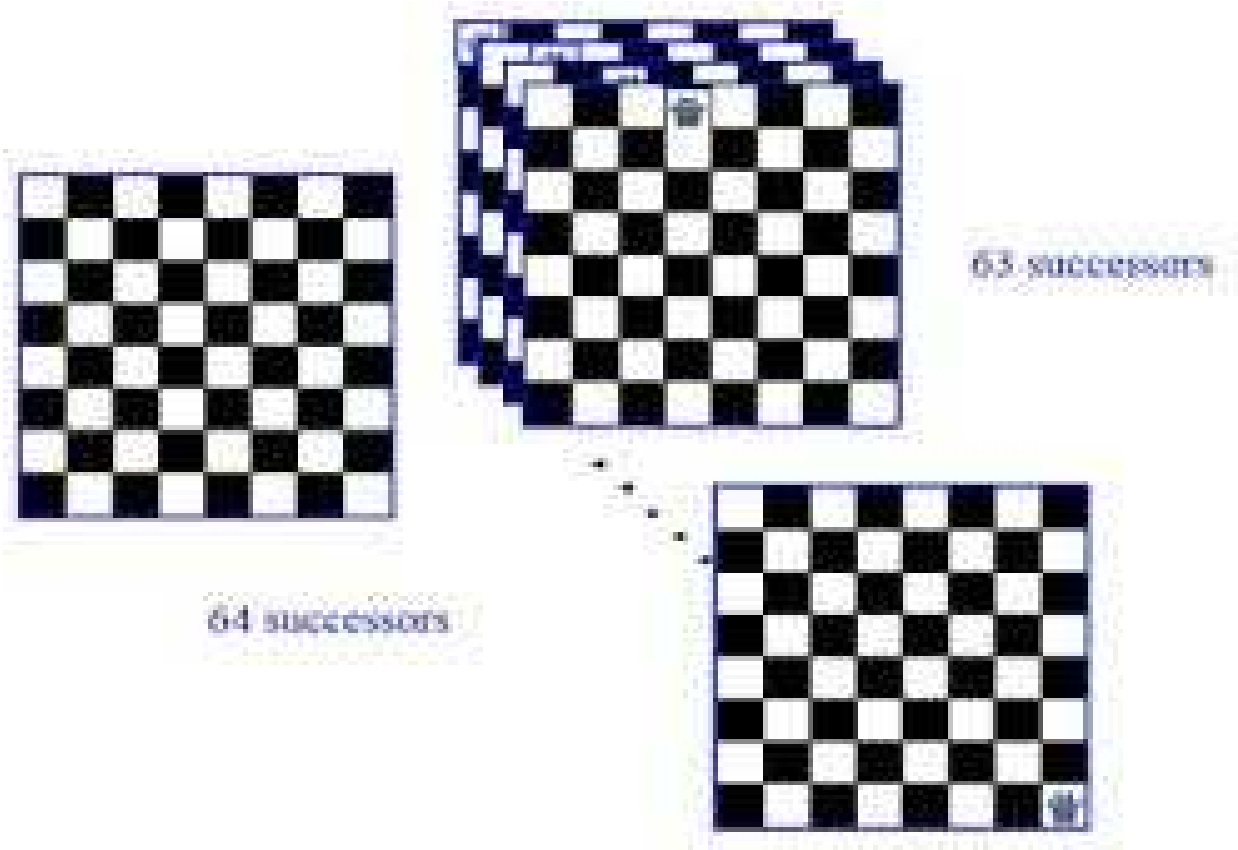

Initial
Initial
Initial
Goal
Initial

8-**queens problem :**The problem is to place 8 queens on a chessboard so that no two queens are in the same row, column or diagonal .How do we formulate this in terms of a state space search problem? The problem formulation involves deciding the representation of the states, selecting the initial state representation, the description of the operators, and the successor states. We will now show that we can formulate the search problem in several different ways for this problem.

N queens problem formulation 1

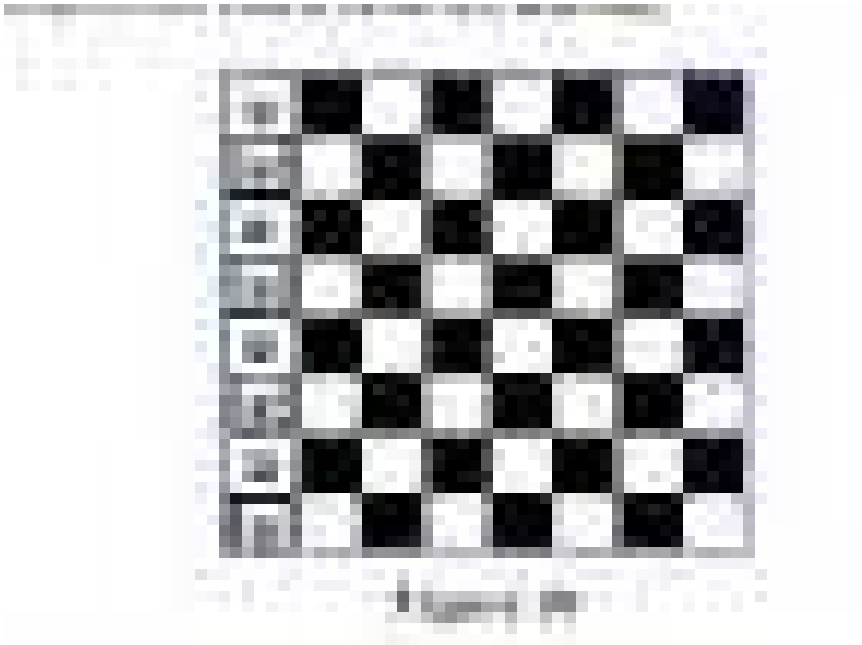
- States: Any arrangement of 0 to 8 queens on the board
- **Initial state**: 0 queens on the board
- **Successor function**: Add a queen in any square
- **Goal test**: 8 queens on the board, none are attacked

The initial state has 64 successors. Each of the states at the next level have 63 successors, and so on. We can restrict the search tree somewhat by considering only those successors where no queen is attacking each other. To do that we have to check the new queen against all existing queens on the board. The solutions are found at a depth of 8.



N queens problem formulation 2

- **States**: Any arrangement of 8 queens on the board
- **Initial state**: All queens are at column 1
- **Successor function**: Change the position of any one queen
- **Goal test**: 8 queens on the board, none are attacked



If we consider moving the queen at column 1, it may move to any of the seven remaining columns.

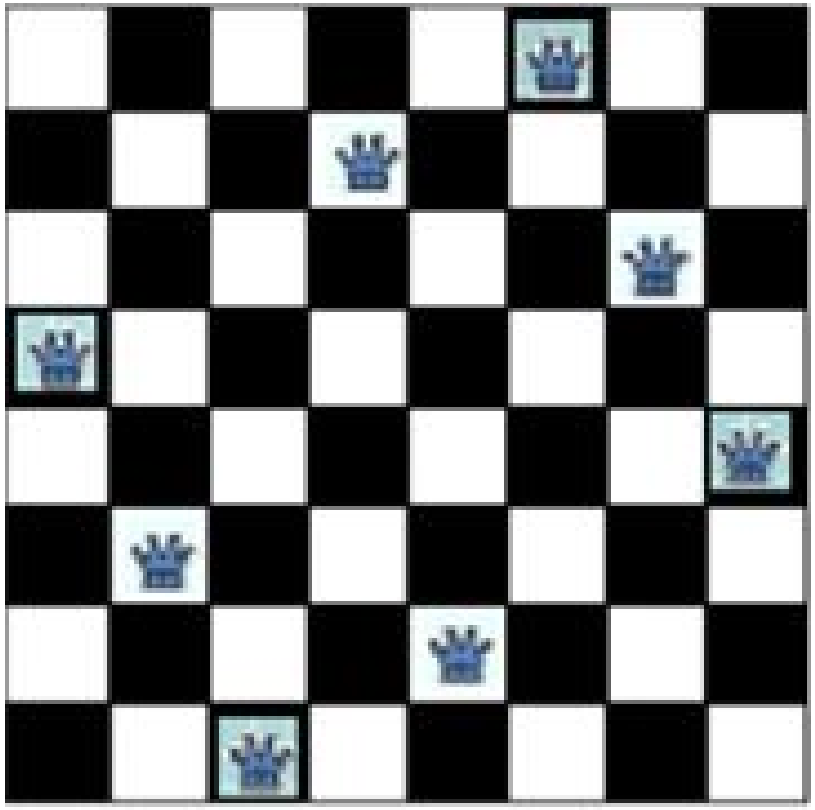
N queens problem formulation 3

- **States**: Any arrangement of k queens in the first k rows such that none are attacked
- **Initial state**: 0 queens on the board
- **Successor function**: Add a queen to the (k+1)th row so that none are attacked.

Keep on shuffling the queen until the goal is reached.

- **Goal test** : 8 queens on the board, none are attacked

This formulation is more systematic hence it is called as iterative formulation.



Goal state

Playing Chess:

State Space Search: Playing Chess

Each position can be described by an 8-by-8 array.

Initial position is the game opening position.

Goal position is any position in which the opponent does not have a legal move and his or her king is under attack.

Legal moves can be described by a set of rules:

- Left sides are matched against the current state.

- Right sides describe the new resulting state.

- **State space** is a set of legal positions.
- Starting at the initial state.
- Using the set of rules to move from one state to another.
- Attempting to end up in a goal state

i . There are roughly 10^120 board position. So it is difficult problem.

ii. No program can easily handle all rules

In order to handle ,to minimize such problem we introduce some convenient notation.

White pawn at

Square(file e , rank 2) Move pawn from

AND

Square(file e , rank 3)->Square(file e , rank 2) to Square(file e , rank 4);

AND

Square(file e , rank 4) is empty

Searching For Solutions: Solution to an AI problem involves performing an action to go to one proper state among possible numerous possible state of agent.

Thus the process of finding solution can be boiled down to searching of that best state among all the possible state.

Search Strategy: We will evaluate strategy in term of four criteria……..

1. Completeness: Is the strategy, guaranteed to find a solution when there is one ?

2. Time Complexity: How long does it take to find solution ?

3.Space Complexity: How much memory does it need to perform the search?

4. optimality: Does the strategy find the highest quality solution when there are several different solution ?

Search Strategies:

1. **Uninformed search or blind search:**
   - Uninformed search means , searching through the state space without using any extra information or domain specific information

   - Having no information about the number of steps from the current state to the goal.

2. **Informed search or heuristic search:**
   - More efficient than uninformed search.

3.Constraint Satisfaction Search

4. Adversary Search

Uninformed search or blind search:

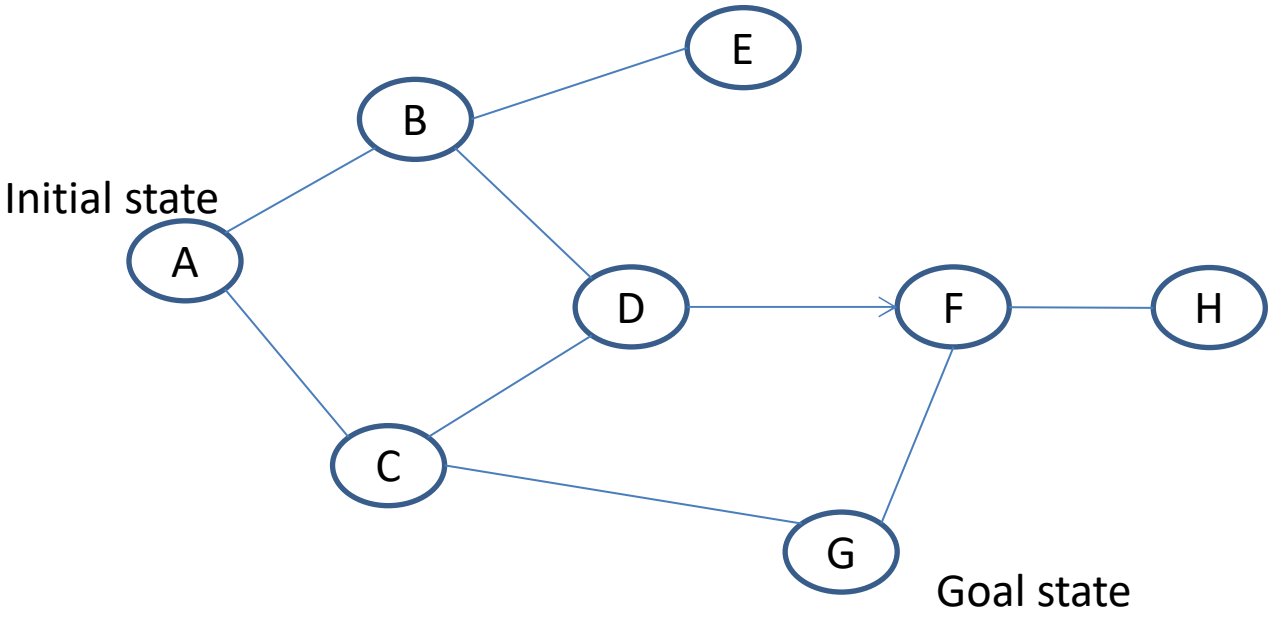
1. Depth first search
2. Breadth first search
3. Iterative deepening search
4. Bidirectional Search
5. Uniform cost search

Blind Search: Blind search or uninformed search that does not use any extra information about the problem domain. The two common methods of blind search are:

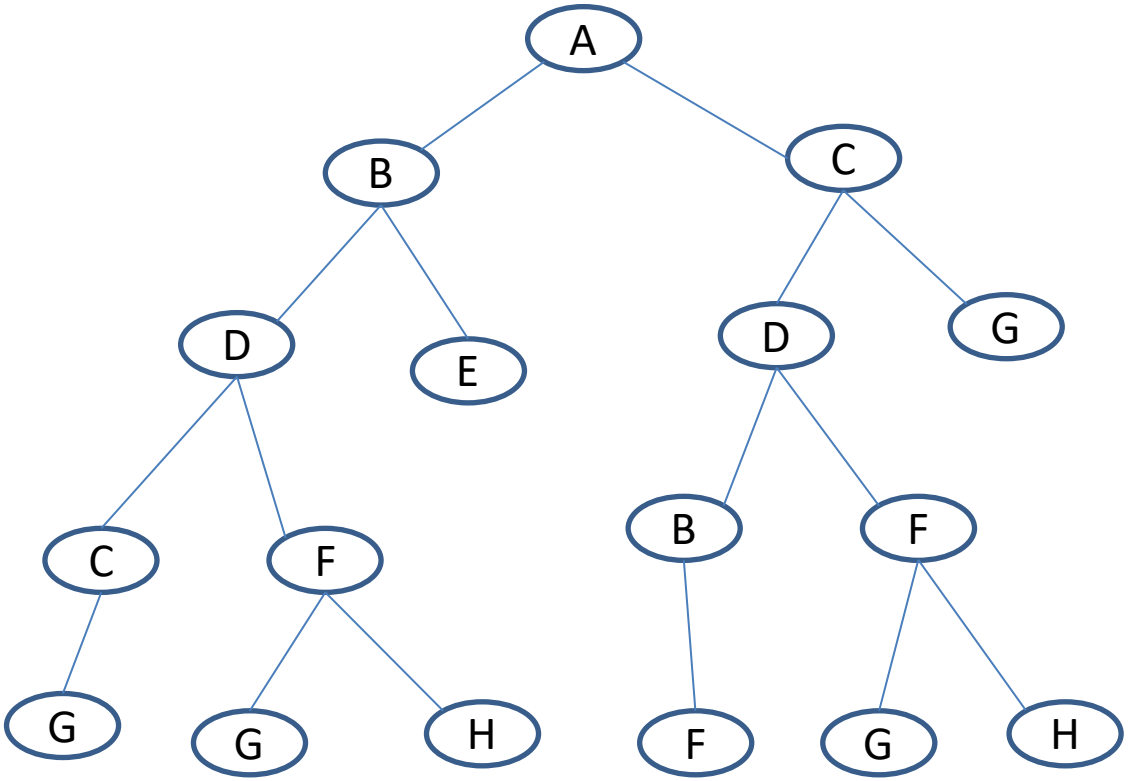
- BFS or Breadth First Search
- DFS or Depth First Search

Search Tree

A search tree is a data structure containing a root node, from where the search starts. Every node may have 0 or more children. If a node X is a child of node Y, node Y is said to be a parent of node X.



State space graph



A Search tree for state space graph

Search Tree – Terminology

- **Root Node**: The node from which the search starts.
- **Leaf Node**: A node in the search tree having no children.
- **Ancestor/Descendant**: X is an ancestor of Y is either X is Y's parent or X is an ancestor of the parent of Y. If S is an ancestor of Y, Y is said to be a descendant of X.
- **Branching factor**: the maximum number of children of a non-leaf node in the search tree
- **Path:** A path in the search tree is a complete path if it begins with the start node and ends with a goal node. Otherwise it is a partial path.

We also need to introduce some data structures that will be used in the search algorithms.

Node data structure

1. **A state description**
2. **A pointer to the parent of the node**
3. **Depth of the node**
4. **The operator that generated this node**
5. **Cost of this path (sum of operator costs) from the start state**

The nodes that the algorithm has generated are kept in a data structure called OPEN or **fringe**. Initially only the start node is in OPEN.

The search starts with the root node. The algorithm picks a node from OPEN for expanding and generates all the children of the node. Expanding a node from OPEN results in a closed node. Some search algorithms keep track of the closed nodes in a data structure called CLOSED.

A solution to the search problem is a sequence of operators that is associated with a path from a start node to a goal node. The cost of a solution is the sum of the arc costs on the solution path. For large state spaces, it is not practical to represent the whole space.

The search process constructs a search tree, where

- **root** is the initial state and
- **leaf nodes** are nodes
- not yet expanded (i.e., in fringe) or
- having no successors (i.e., “dead-ends”)

Search tree may be infinite because of loops even if state space is small

Breadth First Search: First Search is a great algorithm for getting the shortest path to your goal. Breadth First Search by the name itself suggests that the breadth of the search tree is expanded fully before going to the next step.

Two list:

Open List/fringe: initial node

Closed List: <empty>

Algorithm:

Let OPEN be a list containing the initial state and a CLOSED list that store the remove Node

Loop

if OPEN is empty return failure

Node ← remove-first (OPEN) , and

Add to CLOSED List

if Node is a goal

then

return the path from initial state to Node

else

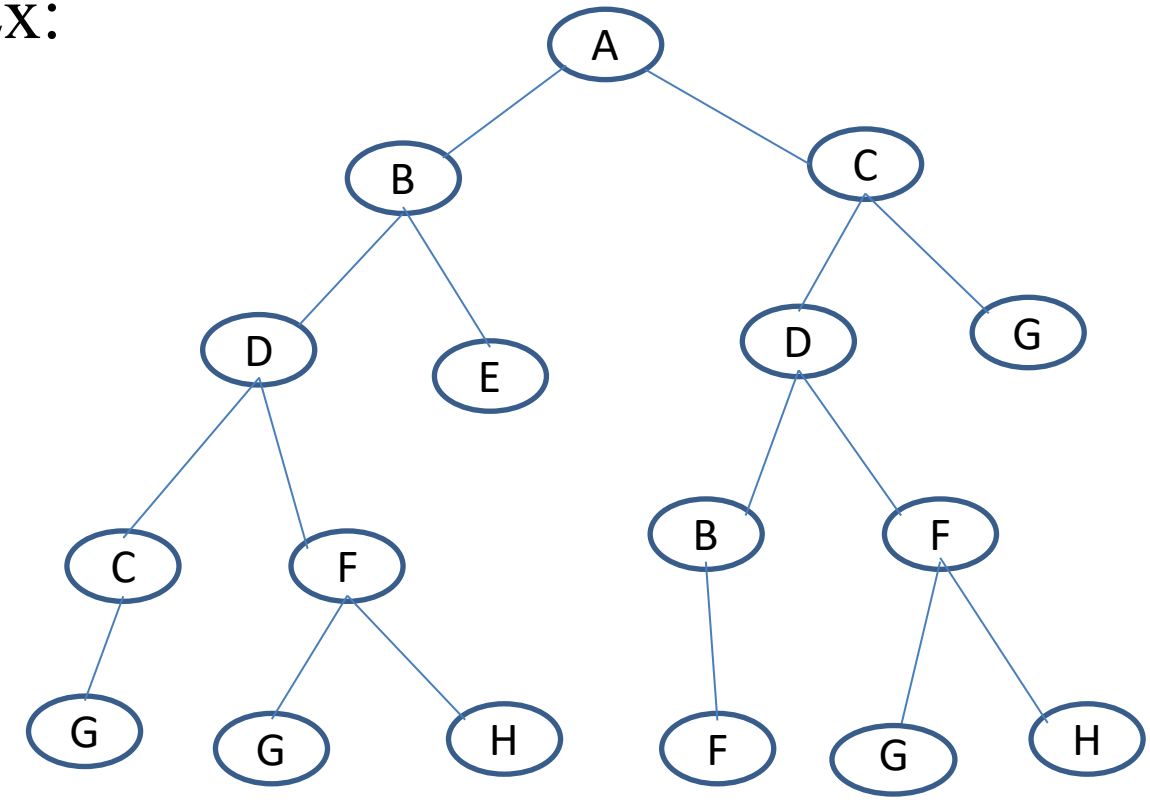
generate all successors of Node, and

add generated nodes to the back of fringe(add in end of OPEN list)

End Loop

In breadth first search the newly generated nodes are put at the back of fringe or the OPEN list , will be expanded in a FIFO (First In First Out) order. The node that enters OPEN earlier will be expanded earlier.

Ex:



Properties of BFS:

We assume that every non-leaf node has b children. Suppose that d is the depth o the shallowest goal node, and m is the depth of the node found first.

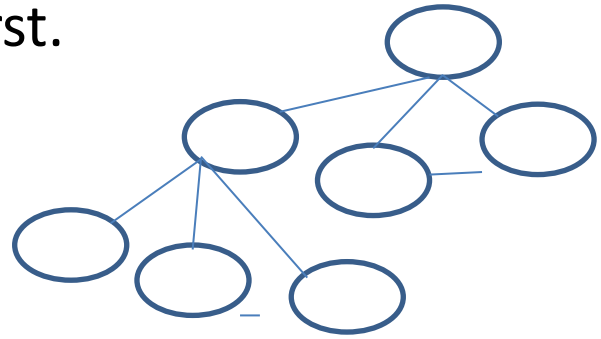


Figure 9: Model of a search tree with uniform branching factor b

Breadth first search is:

Complete- Yes (if *b* is finite)

Time $1+b+b^2+b^3+\ldots+b^d + (b^{d+1}-b)$ total nodes = $O(b^{d+1})$

Space- $O(b^{d+1})$ (keeps every node in memory)where **d** is the depth of the solution and **b** is the branching factor (i.e., number of children) at each node.

Optimal- Yes (if cost = 1 per step)

- The algorithm has exponential time and space complexity.

Advantage:

1. If there is solution ,BFS guaranteed to find it.
2. If there are multiple solution, a minimal solution will be found.

Disadvantage:

1.BFS requires more memory space because all the node of tree must be stored on that level

2.It take more time if the goal state occurs in depth.

Depth first Search: DFS always expands one of the node of the deepest level of the tree .DFS uses LIFO queue for keeping the unexpanded nodes . DFS has very modest memory requirement. It need to store only single path from root to a leaf node, along with the remaining unexpanded sibling node for each node on the path.

Algo:

Let fringe be a list containing the initial state

Loop

If fringe is empty return failure

Node ←remove-first (fringe)

if Node is a goal

then return the path from initial state to Node

else generate all successors of Node, and

merge the newly generated nodes into fringe

add generated nodes to the front of fringe

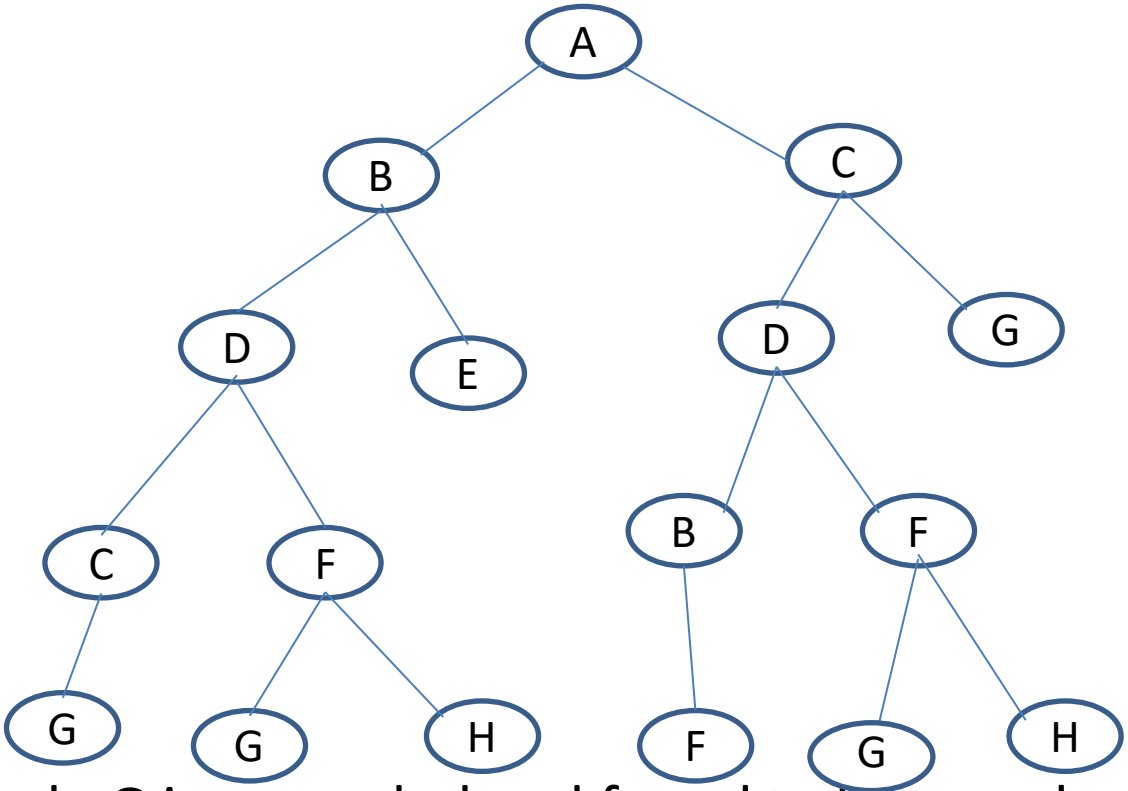
End Loop

Note: *fringe* = LIFO queue, i.e., put successors at front.

Ex.

Fringe: A

- A is expanded and its children B and C are put in front of fringe.



B,C

D ,E ,C

C,F,E,C

G,F,E,C

Node G is expanded and found to be a goal node. The solution path A-B-D-C-G is returned and the algorithm terminates.

Properties of depth-first search:

Complete: No: fails in infinite-depth spaces.

→ complete in finite spaces

Time: $O(b^m)$: terrible if m is much larger than d

but if solutions are dense, may be much faster than breadth-first.

b=no. of expanded node

m=depth of the shallowest goal

Space: For a state space with branching factor **b** and maximum depth-**m**, then DFS require storage of only b*m=bm node, then space complexity $O(bm)$, i.e., linear space.

Optimal: No

Advantage:

1. DFS require less memory since only the node on the current path are stored.

2. If DFS finds solution without exploring much in a path then the time and space will be very less.

***Disadvantage**:*

1. If stop after one solution is found . So minimal solution may not be found.
2. *In DFS there is a possibility that may go down the left-most path forever , even a finite graph can generate a infinite tree*

Depth Limited Search :

A solution of Depth First Search problem . Define a limit in DFS and Nodes are only expanded if they have depth less than the limit. This algorithm is known as depth-limited search.

Let fringe be a list containing the initial state
Loop if fringe is empty return failure
Node← remove-first (fringe)
if Node is a goal
then return the path from initial state to Node
else if depth of Node = limit return cutoff
else add generated nodes to the front of fringe
End Loop

Problem: If we choose limit at which depth solution is not foud. Then search is not complete.

Depth-First Iterative Deepening (DFID): It is the solution of Depth limited search. The Idea is that if do not find the solution up to limit , increase the limit by one.

First do DFS to depth 0 (i.e., treat start node as having no successors), then, if no solution found, do DFS to depth 1, etc.

Algo:

until solution found do

DFS with depth cutoff c

c = c+1

Advantage :

- Linear memory requirements of depth-first search
- Guarantee for goal node of minimal depth

Properties: For large d the ratio of the number of nodes expanded by DFID compared to that of DFS is given by b/(b-1). For a branching factor of 10 and deep goals, 11% more nodes expansion in iterative- deepening search than breadth-first search

The algorithm is

- Complete
- Optimal/Admissible if all operators have the same cost. Otherwise, not optimal but guarantees finding solution of shortest length (like BFS).
- Time complexity is a little worse than BFS or DFS because nodes near the top of the search tree are generated multiple times, but because almost all of the nodes are near the bottom of a tree, the worst case time complexity is still exponential, $O(b^d)$.

If branching factor is b and solution is at depth d, then nodes at depth d are generated once, nodes at depth d-1 are generated twice, etc.

Hence $b^d + 2b^{(d-1)} + \ldots + b^d <= b^d / (1-1/b)^2 = O(b^d)$.

• Linear space complexity, O(b^d), like DFS

Depth First Iterative Deepening combines the advantage of BFS (i.e., completeness) with the advantages of DFS (i.e., limited space and finds longer paths more quickly) This algorithm is generally preferred for large state spaces where the solution depth is unknown.

Informed Search

***Informed Search or heuristic* search**: We know that uninformed search methods systematically explore the state space and find the goal. They are inefficient in most cases. Informed search methods use problem specific knowledge, and may be more efficient. They often depend on the use of heuristic function.

1. Generate and test
2. Hill climbing – i) Simple hill climbing ii) Steepest –ascent hill climbing
3. Best first search
   A* Search and AO* search(AND –OR graph searching)
4. Problem reduction
5. Means-ends analysis
6. Branch and bound

Heuristic function: Heuristic means "rule of thumb" **or** judgmental technique that leads to solution. Heuristics are criteria , method or principles that might not always find the best solution but it is guaranteed to find good solution in reasonable time. In heuristic search or informed search, heuristics are used to identify the most promising search path.

A heuristics function at a node **n** is an estimate of the optimum cost from the current node to goal node. It is denoted by **h(n).**

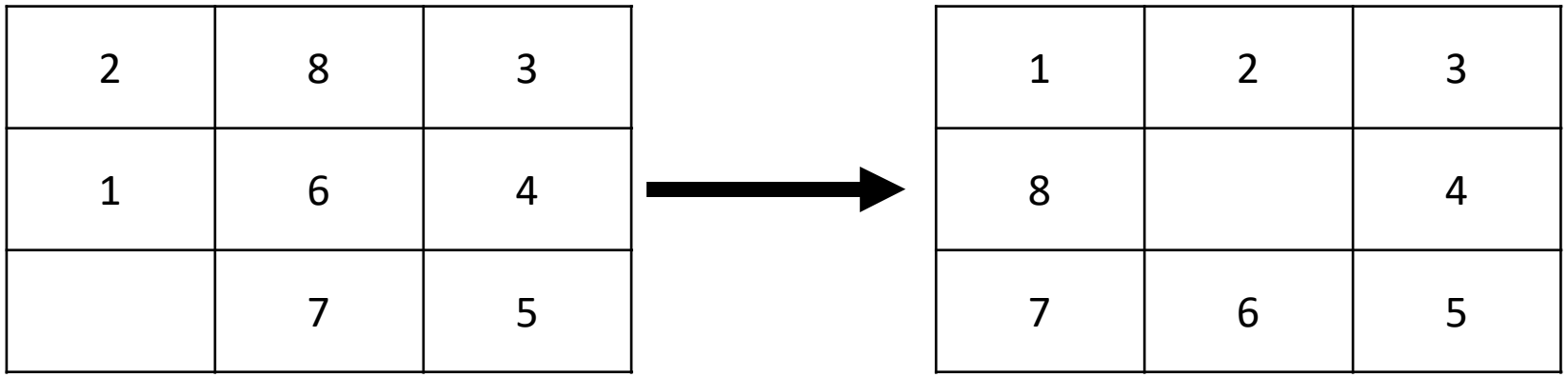
h(n)=estimated cost of the cheapest path from the **n** node to a goal node

Ex . We want a path from mzn to delhi
heuristic for delhi may be straight –line distance b/w mzn to delhi then heuristics function is------

h(mzn)= distance(mzn,delhi)

There are heuristics function of some problem.

8-puzzle- total no. of the misplaced tiles.
TSP- The sum of distance traveled so far.



total no. of the misplaced tiles = 5, because the tiles 2,8,1,6,7 are out of place, then heuristic function is h(n)=4

Manhattan Distance heuristics: Another heuristics for 8-puzzle is MDH. This heuristics sums the distance that the tiles are out of place. The distance of a titles is measured by the sum of the differences in the x-position and y-position.
h(n)=1+1+0+0+0+1+0+2=5

Generate and Test: The Generate and Test strategy is the simplest of all approaches.

Algorithm:

1. Generate a possible solution. For some problems, this means generating a particular point in the problem space.

2. Test to see if this is a solution by comparing the chosen point or the end-point of the chosen path to the set of acceptable goal states.

3. If a solution has been found, quit. Otherwise, return to step 1.

If the generation of possible solutions is done systematically, then the process will find a solution, if it exists. This is called *"Exhaustive search"*.

If the generation of possible solutions is done randomly, there is no guarantee that a solution is found. This type of search is called *"British Museum Algorithm"*

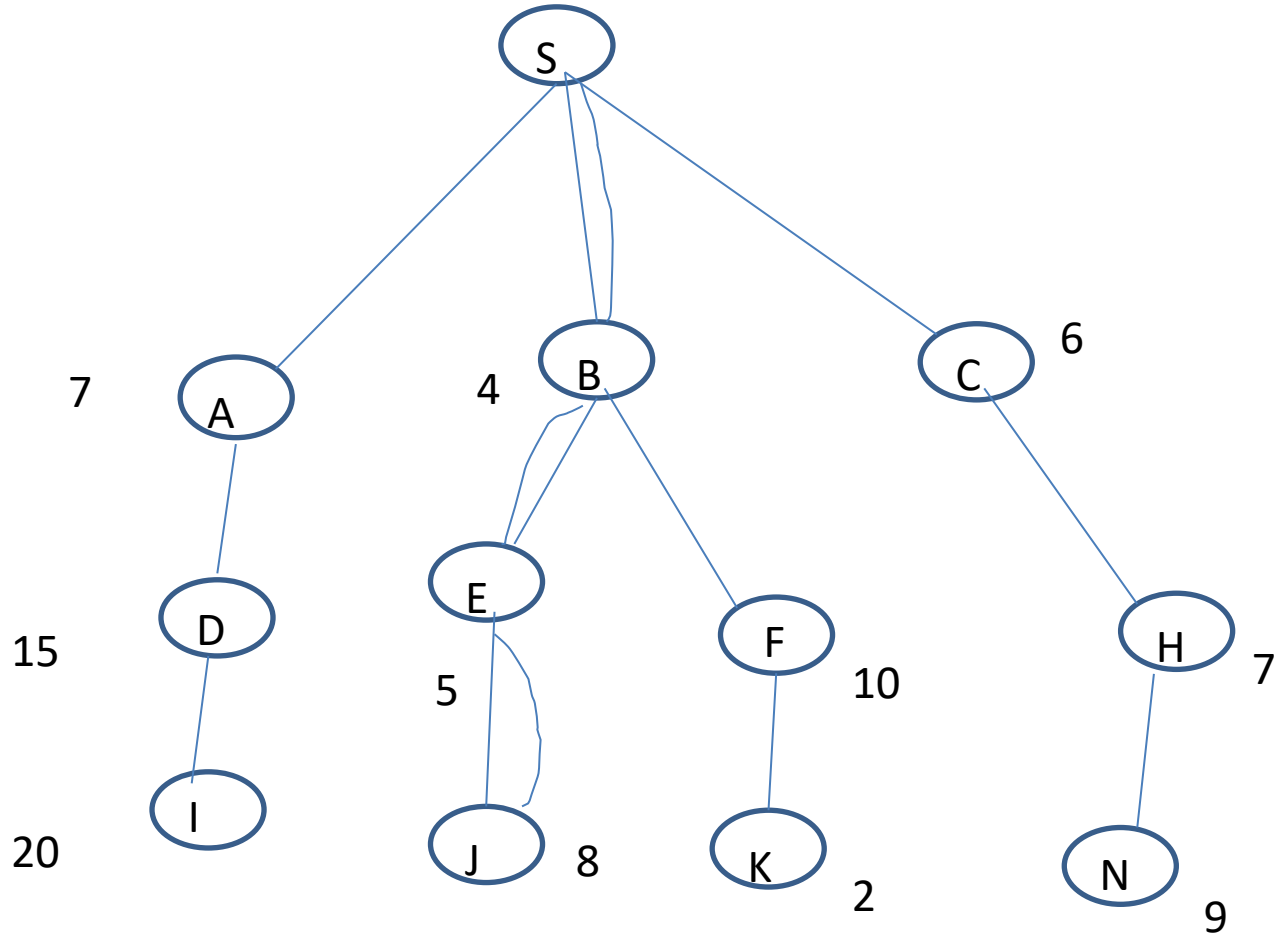
For simple problems, exhaustive generate-and-test is reasonable technique. For harder problems, the heuristic generate-and-test is not effective technique. This technique can be made effective by combining with other techniques.

HILL CLIMBING SEARCH: Hill climbing is the variant of generate –and- test in which feedback from the test procedure is used to help . The generator decide which direction to move in the search space . Here the G &T function is augmented by an heuristic function which measure the closeness of the current state to the goal state.

Hill climbing is often used when a good heuristics function is available for evaluating state but no other useful knowledge is available.

Simple Hill climbing: In simple hill climbing, the first closer node is chosen.

Steepest –Ascent hill climbing-In steepest ascent hill climbing all successors are compared and the closest to the solution is chosen. Also called gradient search.



Start at S

Children of S= [A(7),B(4), C(6)]

Best child of S=B(4)

Children of S=[E(5),F(10)]

Best child of B=E(5)

Children of E=[J(8)]

Difficulties in Hill climbing:

(a) A "**local maximum** " which is a state better than all its neighbors , but is not better than some other states farther away. Local maxim sometimes occur with in sight of a solution. In such cases they are called " Foothills".

(b) A "**plateau**" which is a flat area of the search space, in which neighboring states have the same value. On a plateau, it is not possible to determine the best direction in which to move by making local comparisons.

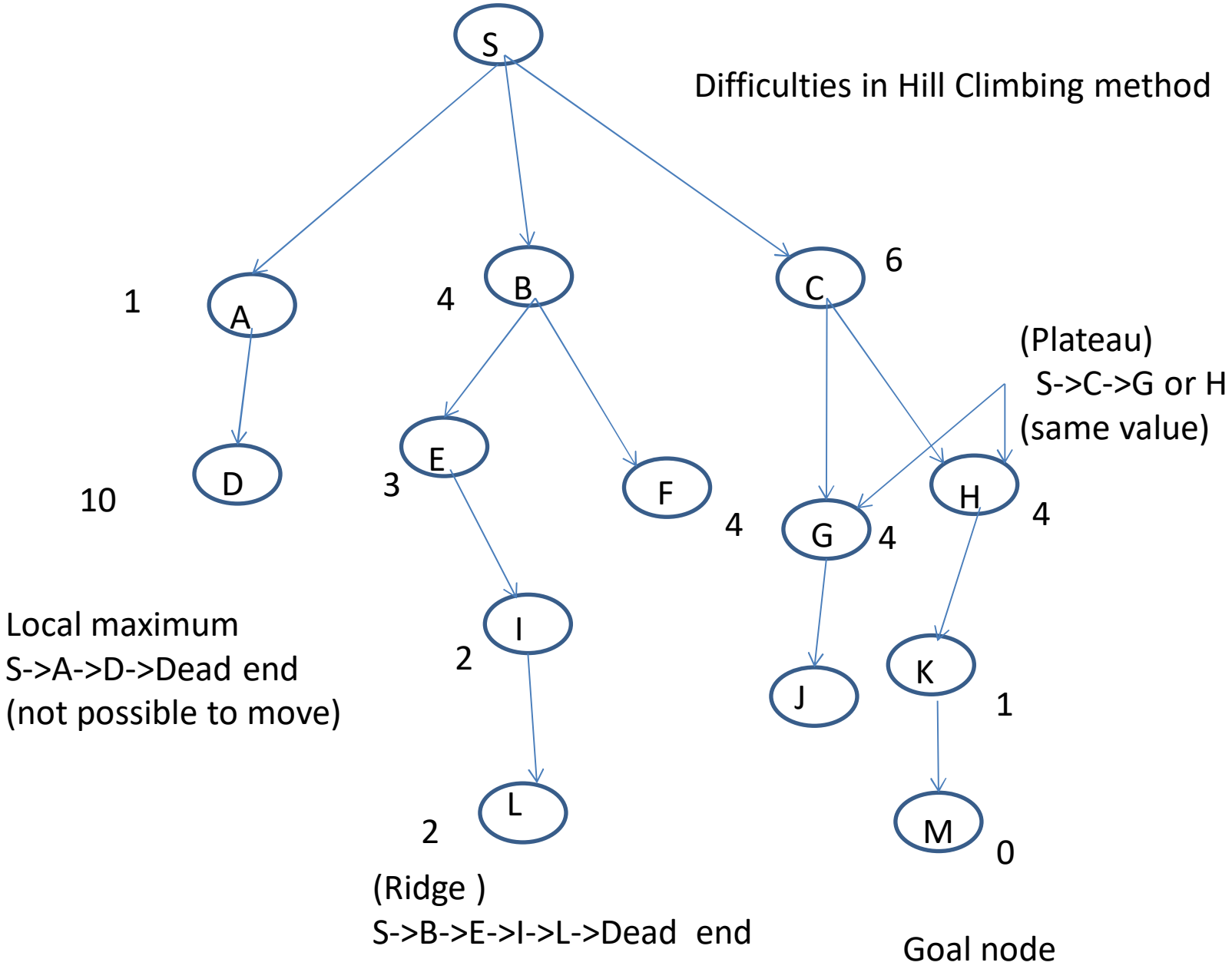
(c) A "**ridge**" which is an area in the search that is higher than the surrounding areas, but can not be searched in a simple move.

To overcome theses problems we can

(a) Back track to some earlier nodes and try a different direction. This is a good way of dealing with local maxim.

(b) Make a big jump an some direction to a new area in the search. If the rules available describe only single small steps then apply them several times in the same direction

(c) Applying two more rules of the same rule several times, before testing. This is corresponding to moving in several directions at once


Difficulties in Hill Climbing method
S
A
B
C
D
E
F
G
H
I
J
K
L
M
1
4
6
10
3
4
4
2
1
2
0
(Plateau)
S->C->G or H
(same value)
Local maximum
S->A->D->Dead end
(not possible to move)
(Ridge)
S->B->E->I->L->Dead end
Goal node

Algorithm:

1.Evaluate the initial state If it is goal state then quit ,otherwise make the current state this initial state and proceed;

2. Loop until a solution is found or until there are no new operator left to be applied in the current state

(a)Select an operator an operator and apply to the current state and procedure all its children as a new state

(b) Evaluate the best new state.

(i) If this state is goal state then return

(ii) If not a goal state but better than the current state then make it the current state

(iii) If not better than current state then continue in loop.

Best-First Search: This method is similar to the hill climbing, In hill climbing ,one move to be selected and all the other is rejected, never to be considered .But in best-first search we can not reject node but kept around so that they can be revisited later.

Family of best first search algorithms exists with different evaluation functions.

A key component is a heuristic function h*(n):*

h(n) = estimated cost of the cheapest path from node n to a goal node.

If n is goal node, h(n)=0.

Special cases:

- greedy best-first search
- A* search

Greedy Search:

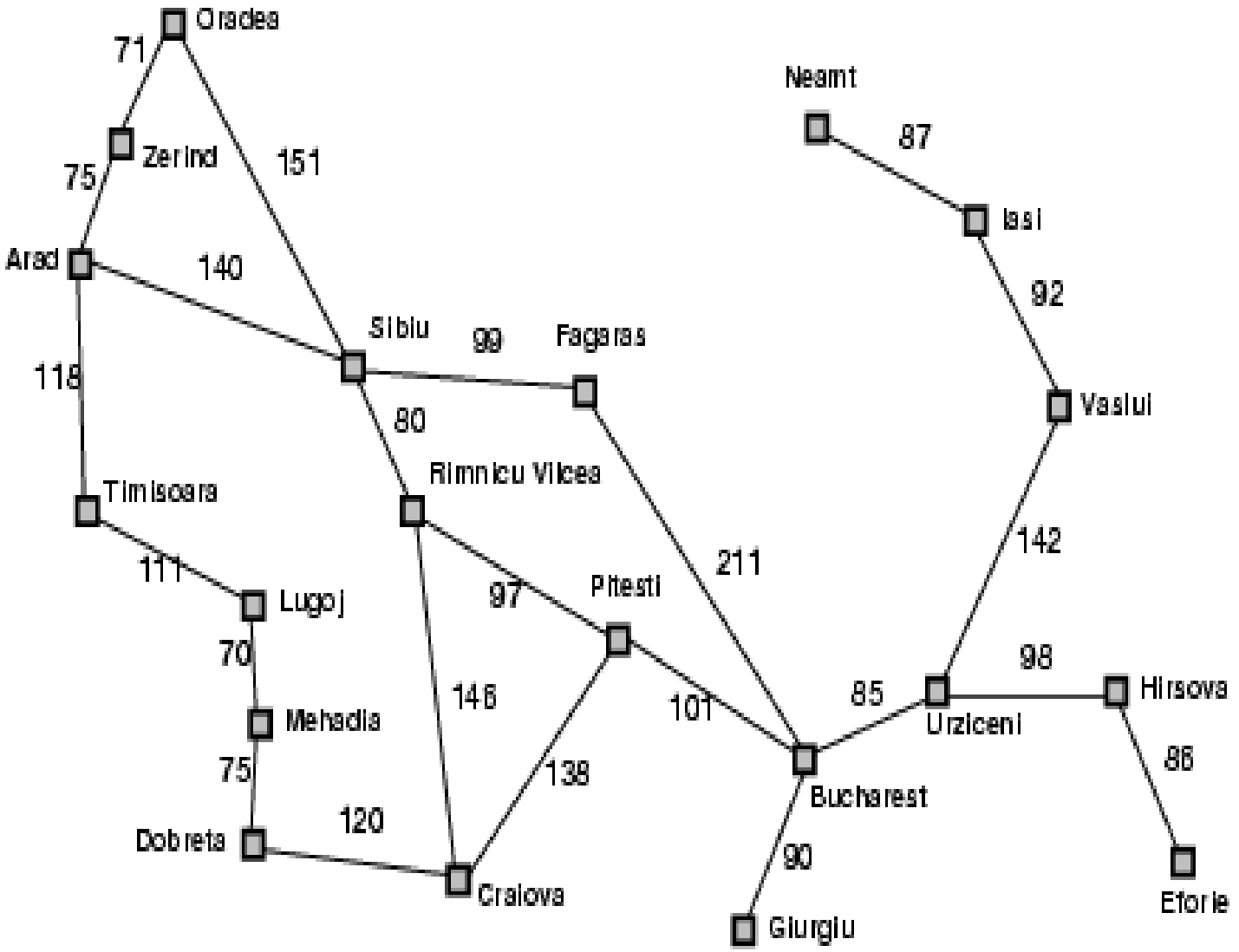
Evaluation function $f(n) = h(n)$ (heuristic)

= estimate of cost from n to *goal.*

e.g., $h_{SLD}(n)$ = straight-line distance from n to Bucharest.

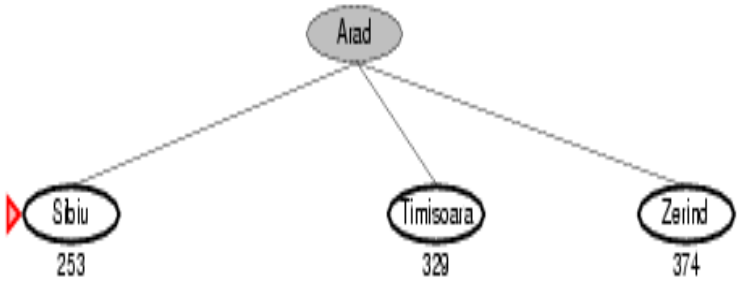
Greedy best-first search expands the node that appears to be closest to goal

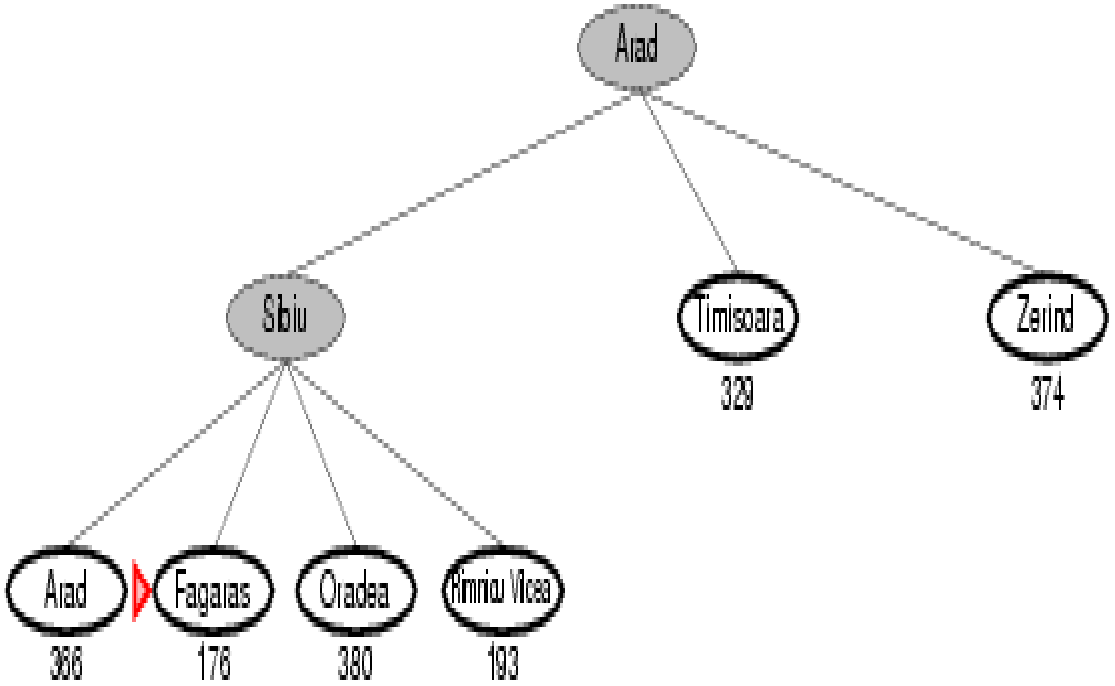
Romania with step costs in km

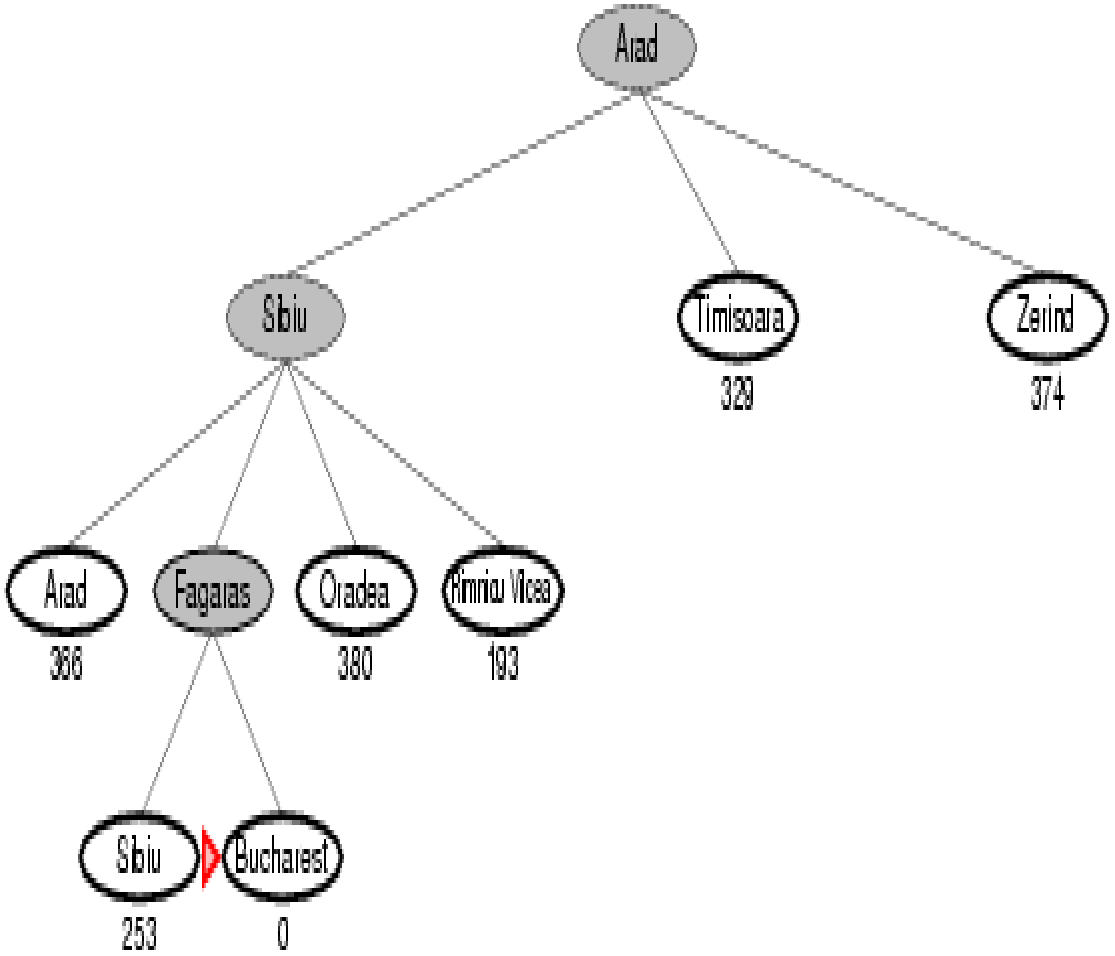


Straight-line distance to Bucharest	
Arad	366
Bucharest	0
Craiova	160
Dobreta	242
Eforie	161
Fagaras	176
Giurgiu	77
Hirsova	151
Iasi	226
Lugoj	244
Mehadia	241
Neamt	234
Oradea	380
Pitesti	10
Rimnicu Vilcea	193
Sibiu	253
Timisoara	329
Urziceni	80
Vaslui	199
Zerind	374

Greedy best-first search example




Arad
Sibiu
Timisoara
329
Zerind
374
Arad
366
Fagaras
176
Oradea
380
Rimnicu Vicea
193


Arad
Sibiu
Timisoara
329
Zerind
374
Arad
366
Fagaras
Oradea
380
Rimnicu Vicea
193
Sibiu
253
Bucharest
0

Properties:

Complete? No – can get stuck in loops, e.g.,

Iasi → Neamt → Iasi → Neamt →

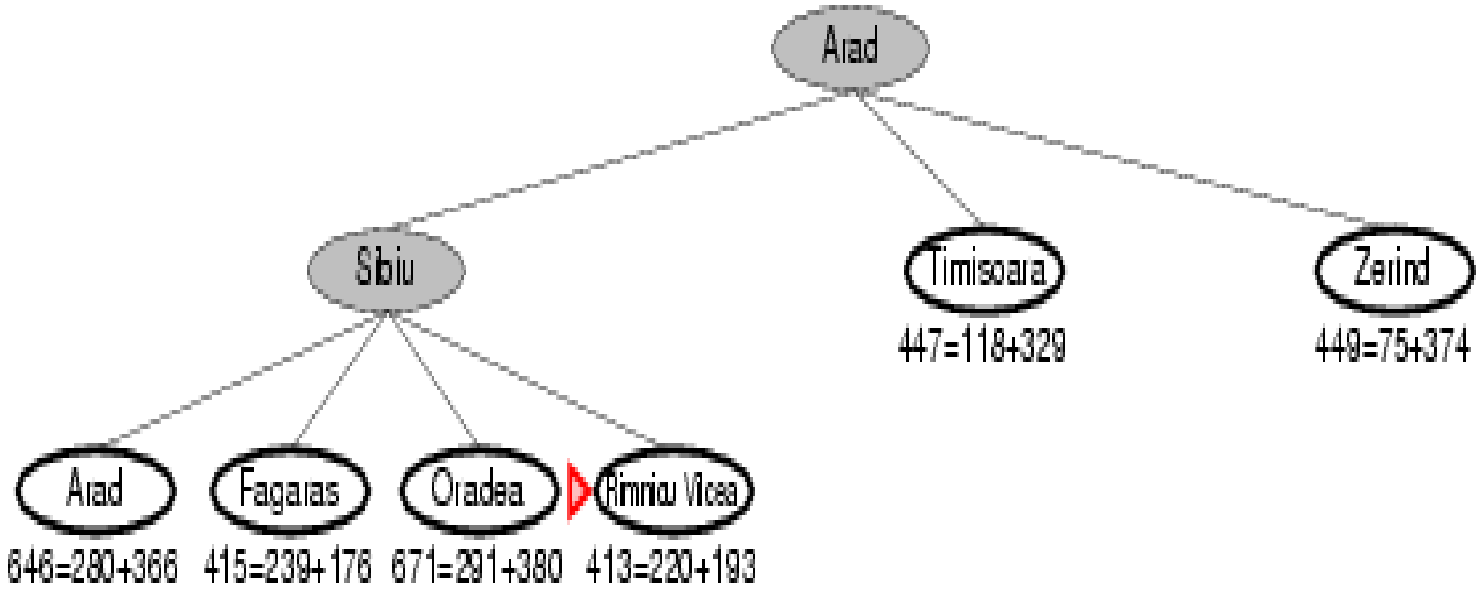
Time? $O(b^m)$, but a good heuristic can give dramatic improvement

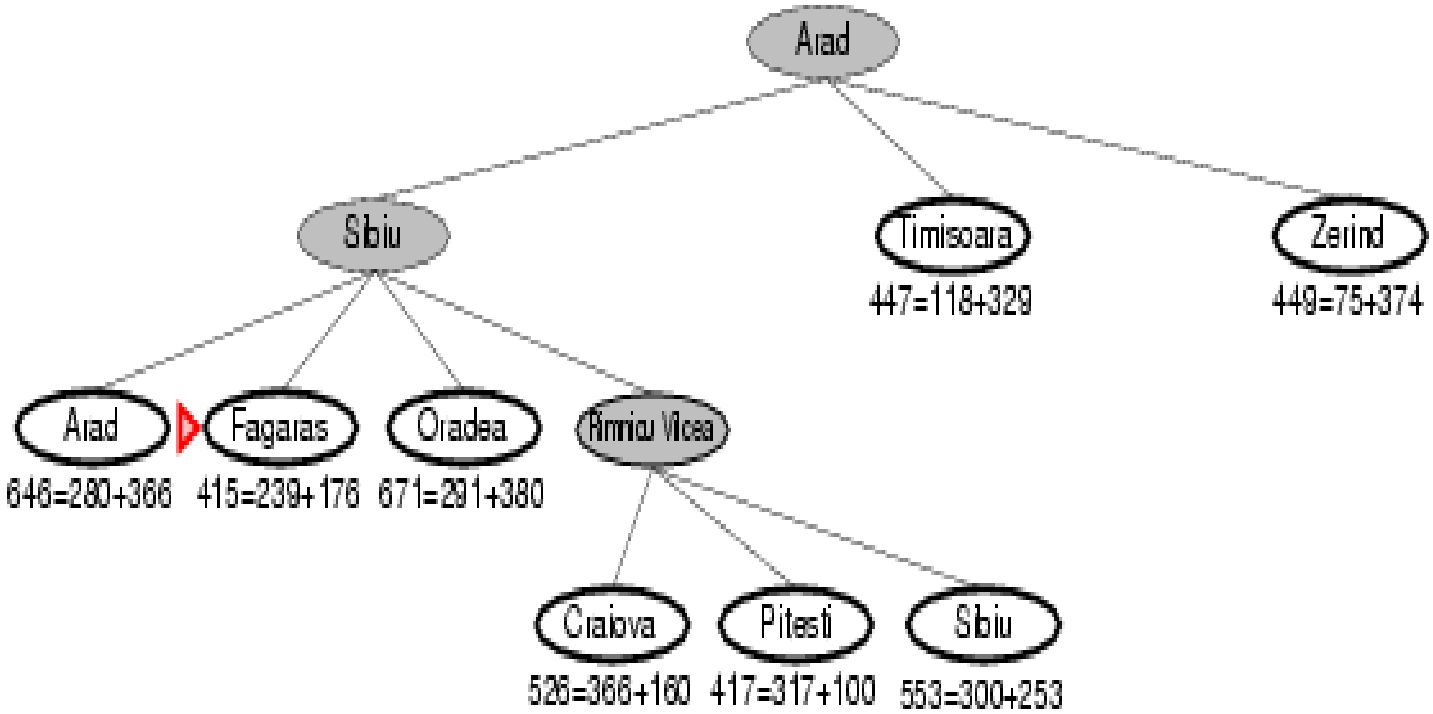
Space? $O(b^m)$ -- keeps all nodes in memory

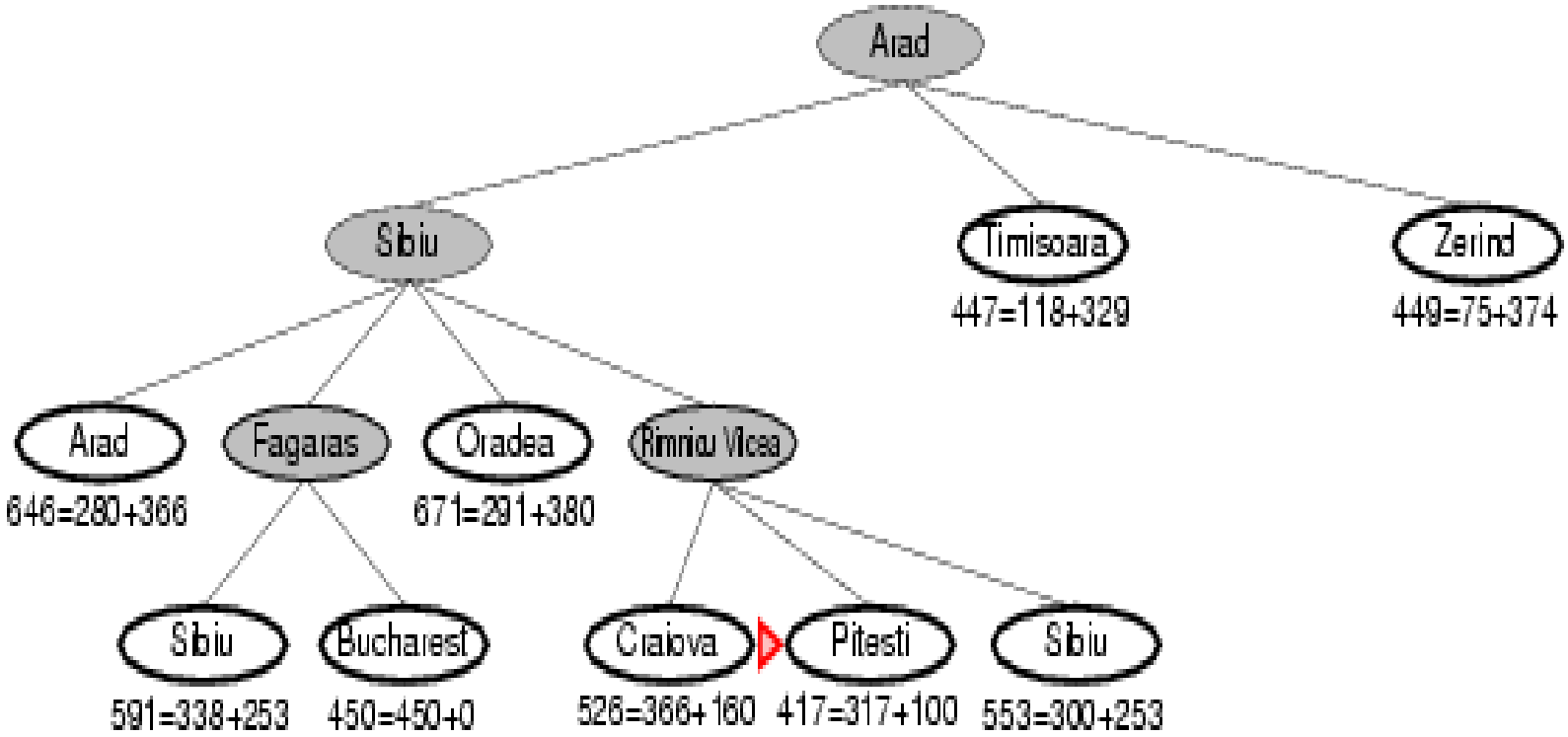
Optimal? No

A* search:

Idea: avoid expanding paths that are already expensive.
Evaluation function *f(n) = g(n) + h(n).*
g(n) = cost so far to reach *n.*
h(n) = estimated cost from *n* to goal.
f(n) = estimated total cost of path through *n* to goal.


Arad
Sibiu
Timisoara
Zerind
447=118+329
449=75+374
Arad
Fagaras
Oradea
Rimnicu Vicea
646=280+366
415=239+176
671=291+380
413=220+193


Arad
Sibiu
Timisoara
Zerind
447=118+329
449=75+374
Arad
Fagaras
Oradea
Rimnicu Vilcea
646=280+366
415=239+176
671=291+380
Craiova
Pitesti
Sibiu
526=366+160
417=317+100
553=300+253


Arad
Sibiu
Timisoara
447=118+329
Zerind
449=75+374
Arad
646=280+366
Fagaras
Oradea
671=291+380
Rimnicu Vicea
Sibiu
591=338+253
Bucharest
450=450+0
Craiova
526=366+160
Pitesti
417=317+100
Sibiu
553=300+253

Admissible heuristics:

A heuristic *h(n)* is admissible if for every node *n*,

$h(n) \leq h^*(n)$, where $h^*(n)$ is the true cost to reach the goal state from *n*.

An admissible heuristic never overestimates the cost to reach the goal, i.e., it is optimistic.

Example: $h_{SLD}(n)$ (never overestimates the actual road distance).

Theorem: If *h(n)* is admissible, A* using `TREE-SEARCH` is optimal.

Properties:

Complete- Yes.

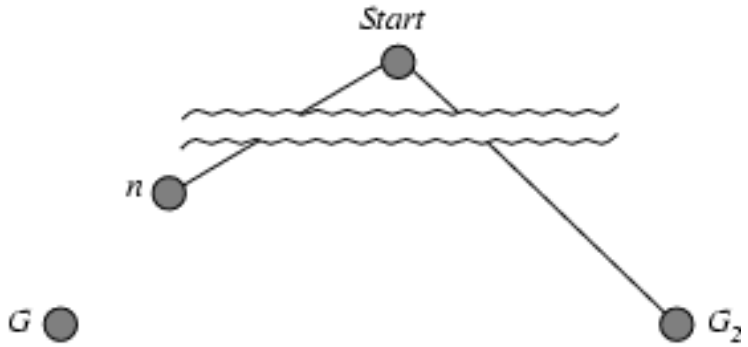
Time- Exponential.

Space-Keeps all nodes in memory.

Optimal- Yes.

Optimality of A* (proof):

Suppose some suboptimal goal G_2 has been generated and is in the fringe. Let *n* be an unexpanded node in the fringe such that *n* is on a shortest path to an optimal goal *G*.

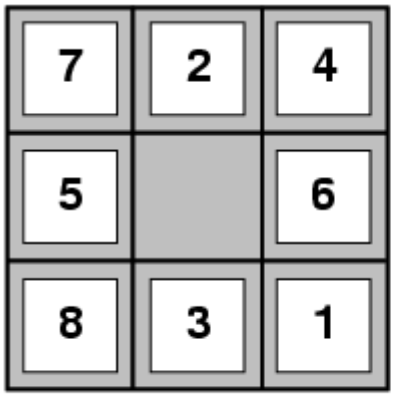


$f(G_2)$	$> f(G)$	from above
$h(n)$	$\leq h^*(n)$	since h is admissible
$g(n) + h(n)$	$\leq g(n) + h^*(n)$	
$f(n)$	$\leq f(G)$	

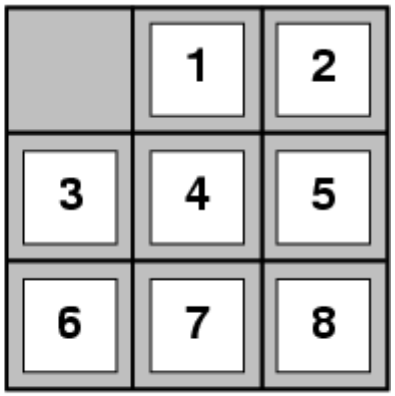
Hence $f(G_2) > f(n)$, and A* will never select G_2 for expansion.

Admissible heuristics:

E.g., for the 8-puzzle:
$h_1(n)$ = number of misplaced tiles
$h_2(n)$ = total Manhattan distance
(i.e., no. of squares from desired location of each tile)



Start State



Goal State

$h_1(S) = ?$ 8
$h_2(S) = ?$ 3+1+2+2+2+3+3+2 = 18

Dominance: If $h_2(n) \geq h_1(n)$ for all n (both admissible) then h_2 dominates h_1
h_2 is better for search.

Properties of Heuristic Algorithms:

Admissibility: Algorithm A is admissible if it is guaranteed to return an optimal solution when one exists.
Completeness: Algorithm A is complete if it is guaranteed to return a solution when one exists.
Dominance: A1 dominates A2 if the heuristic function of A1 that of A2.
Optimality: Algorithm A is optimal over a class of algorithms if A dominates all members of the class.

Constraints Satisfaction: It is a heuristics based technique. Constraints satisfaction is a process of finding a solution to set of constraints that impose condition that the variable must satisfy. A solution is therefore set of value for the variable that satisfy all the constraints .

How to solve problem:

1. We need to analyzing the problem.
2. We need to derive the constraints given in problem.
3. We need to derive the solution of given constraints.
4. We need to find whether find a good state if we not reach to goal state then we need to Guess, that has add new constraints and again solve find solution of problem.

1.Crypt-arithmetic problem
2. N-Queen Problem
3. Map coloring problem
4. Crossword Puzzle

1.Crypt-arithmetic problem

Ex. 1.TWO + TWO=FOUR 2. SEND+MORE=MONEY
3. NO+NO=YES 4.HERE+SHE= COMES
5.TOM+NAG=GOAT ODD+ODD=EVEN

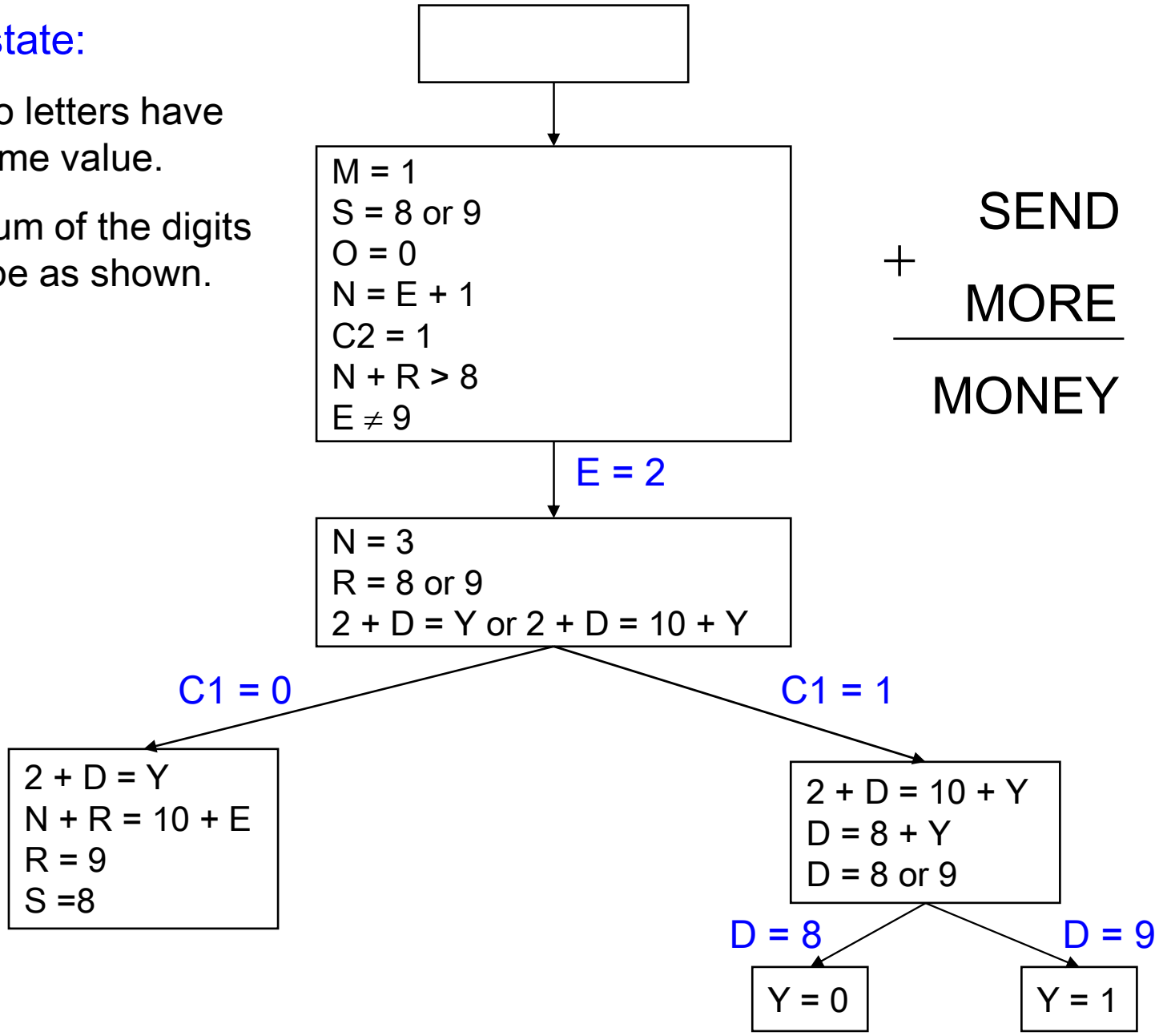
Constraints: 1. Assign some unique digit to each letter
2. We can't assign deferent digit of same letter
3.No two letter have same digit

Two-step process:

1. Constraints are discovered and propagated as far as possible.
2. If there is still not a solution, then search begins, adding new constraints.

Initial state:

- No two letters have the same value.
- The sum of the digits must be as shown.



SEND
\+ MORE
MONEY

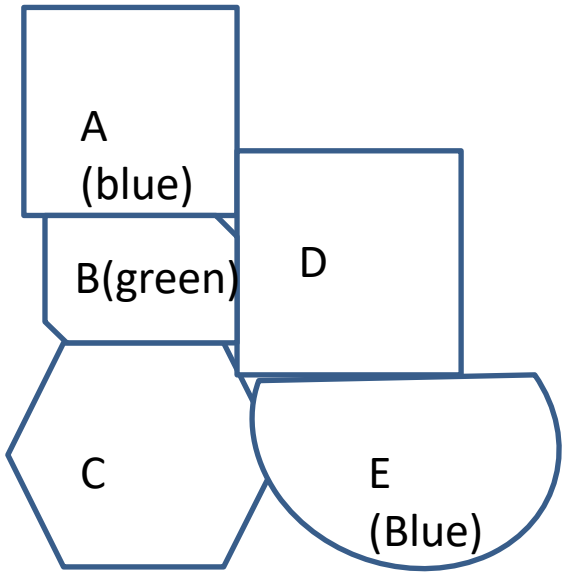
Two kinds of rules:

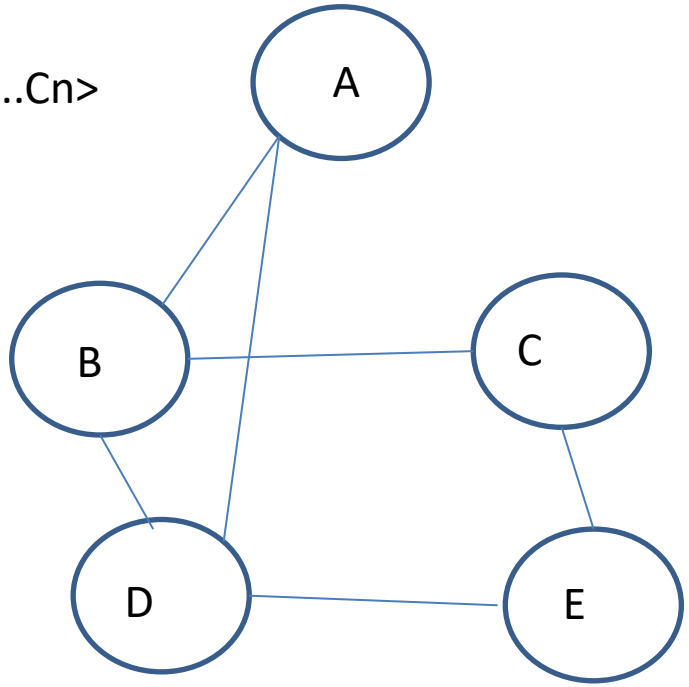
1. Rules that define valid constraint propagation.
2. Rules that suggest guesses when necessary.

A map coloring problem: We are given a map, i.e. a planar graph, and we are told to color it using **k** colors, so that no two neighboring countries have the same color.

You have to color a planner map using only four colors .In such a way no two adjacent regions have the color.
This map is represented by a graph .each regions corresponding to vertex of a graph .If two regions are adjacent ,there is an edge connecting the corresponding vertices.

The vertices are < v1,v2,v3……Vn >
The color are represented by < c1,c2,c3….Cn>
A state represented as a N-tupple.
Ex.





Particular state of graph : Representation of current state

{ blue, green ,X,X blue)

Initial state: { x,x,x,x}

Goal state :

if Xi and Xj are adjacent

Color(i) not equal to color(j).

All regions have colored.

What are different operation apply the graph change the color of a state i to c…

->change (i,c)

->change(2,yellow)-> B is color(where B is state)

-> change(3,green) -> C is color

Adversarial Search: A framework for formulating a multi-person game as a search problem. We will consider games in which the players alternate making moves and try respectively to maximize and minimize a scoring function (also called utility function). we will only consider games with the following two properties:

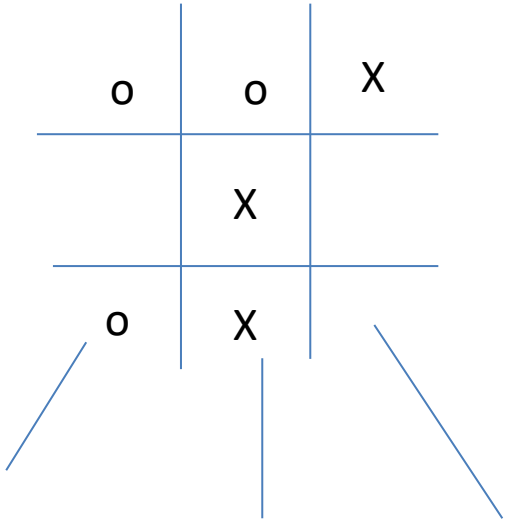
- **Two player** - we do not deal with coalitions, etc.
- **Zero sum** - one player's win is the other's loss; there are no cooperative victories

Game tree:

The above category of games can be represented as a tree where the nodes represent the current state of the game and the arcs represent the moves. The game tree consists of all possible moves for the current players starting at the root and all possible moves for the next player as the children of these nodes, and so forth, as far into the future of the game as desired. Each individual move by one player is called a "ply". The leaves of the game tree represent terminal positions as one where the outcome of the game is clear (a win, a loss, a draw, a payoff). Each terminal position has a score. High scores are good for one of the player, called the MAX player. The other player, called MIN player, tries to minimize the score. For example, we may associate 1 with a win, 0 with a draw and -1 with a loss for MAX.

Ex. **tic-tac-toe**

In tic-tac-toe, as in many turn based, adversarial games, the game can end in one of only three ways: win, lose, draw. We may then assign the utility, 1, to terminal states in which the game is decided in the agent's favor, -1 to those states in which the agent has lost the game, and 0 to the states in which a draw has been reached.



X's turn(MAX)

- Above is a section of a game tree for tic tac toe. Each node represents a board position, and the children of each node are the legal moves from that position

Minimax Algorithm

How do we compute our optimal move? We will assume that the opponent is rational; that is, the opponent can compute moves just as well as we can, and the opponent will always choose the optimal move with the assumption that we, too, will play perfectly. One algorithm for computing the best move is the minimax algorithm:

```
minimax(player,board)
        if(game over in current board position)
  return winner
        children = all legal moves for player from this board
  if(max's turn)
        return maximal score of calling minimax on all the children
else (min's turn)
        return minimal score of calling minimax on all the children
```

If the game is over in the given position, then there is nothing to compute; minimax will simply return the score of the board. Otherwise, minimax will go through each possible child, and (by recursively calling itself) evaluate each possible move. Then, the best possible move will be chosen, where ‘best’ is the move leading to the board with the most positive score for player 1, and the board with the most negative score for player 2.

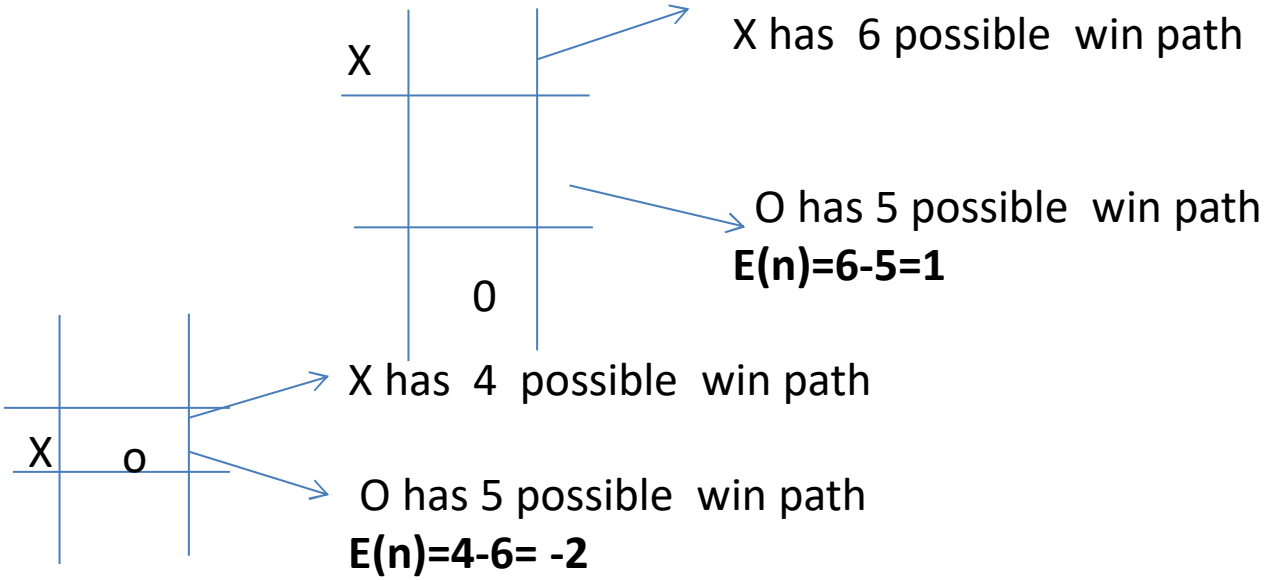
Heuristic evaluation function:

-An evaluation function estimate how good the current board configuration is for a player

E(n)=M(n)- O(n) [subtract the opponents score from the players]

M(n) – total of MAX player possible wining line
O(n) – total of Opponent possible wining line
E(n) – total evaluation for state n

- values range from –infinity (loss) to +infinity(win) or [-1,+1]



Properties:

Complete - **yes(** if tree is infine)
optimal - **yes**(against an optimal opponent)
Time complexity- **O(b^m)** or **O(b^m)**
space - **O(b*m)**

m – maximum depth of the tree; b – legal moves

MiniMax is two player game, the player are referred to as MAX(the player) and MIN(the opponent).

MAX is the player trying to maximize its score and **MIN** is the opponent trying to minimize **MAX's** score.

Optimal strategy for MINIMAX: Design to find optimal strategy for max and find best move.

Generate the whole game tree to leaves. Apply evaluated function to leaves.

Back –up values from leaves toward the root.

- a MAX node compute the max of its child values.
-a MIN node compute the min of its child values.

4. When value reaches the root: Choose max value and the corresponding move.

Component: -Initial ,successor function, terminal state, utility function

Note: Higher utility value good for MAX and lower value bad for MAX

Alpha–beta pruning:

The minimax algorithm is a way of finding an optimal move in a two player game. *Alpha-beta pruning* is a way of finding the optimal minimax solution while avoiding searching sub-trees of moves which won't be selected. In the search tree for a two-player game, there are two kinds of nodes, nodes representing *your* moves and nodes representing *your opponent's* moves.

***MAX* nodes:** The goal at a MAX node is to maximize the value of the sub-tree rooted at that node. To do this, a MAX nod

***MIN* nodes:** The goal at a MIN node is to minimize the value of the sub-tree rooted at that node. To do this, a MIN node chooses the child with the least (smallest) value, and that becomes the value of the MIN node.

Alpha-beta pruning gets its name from two bounds that are passed along during the calculation, which restrict the set of possible solutions based on the portion of the search tree

β- Beta is the *minimum upper bound* of possible solutions

α -Alpha is the *maximum lower bound* of possible solutions

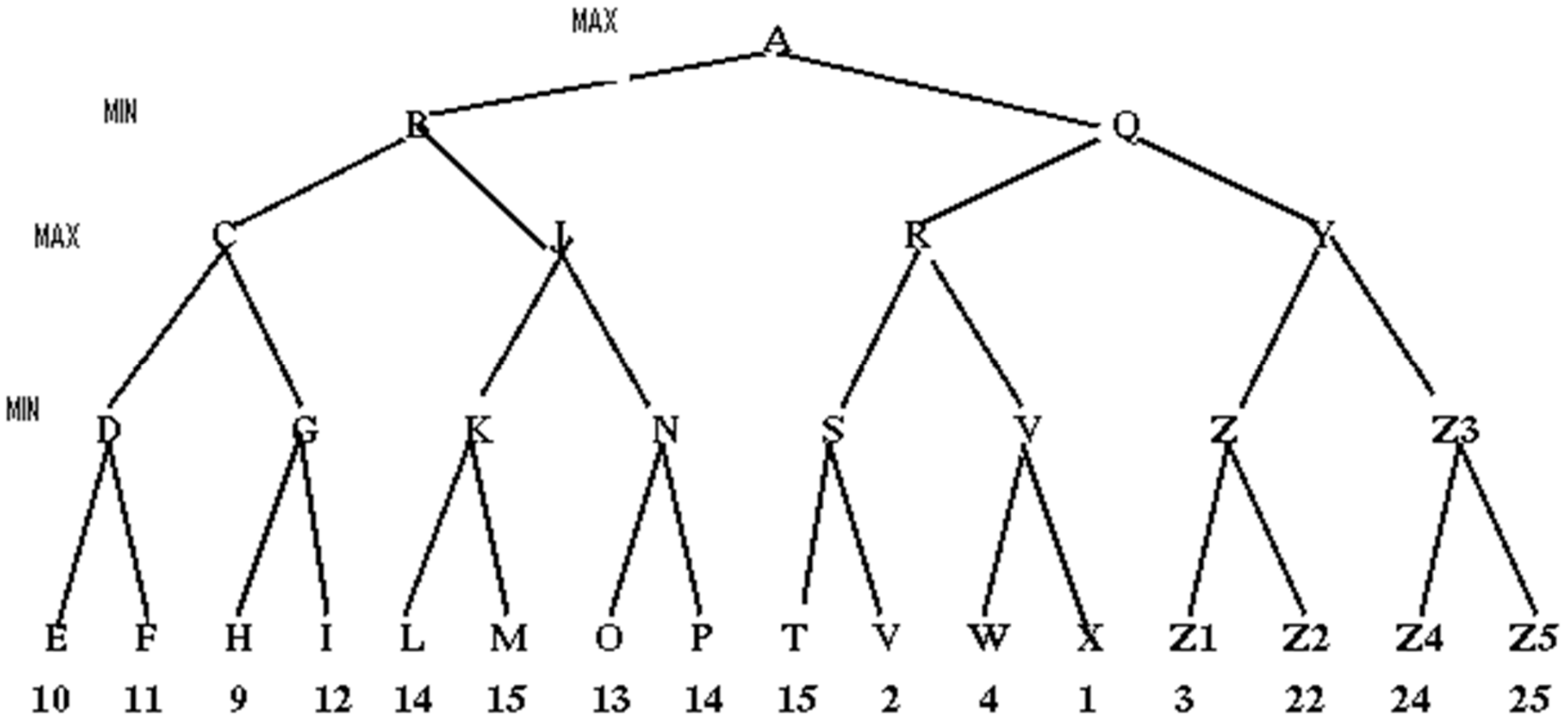
Thus, when any new node is being considered as a possible path to the solution, it can only work if:

$$\alpha <= N <= \beta$$

where N is the current estimate of the value of the node.

Example: As for upper and lower bounds, all you know is that it's a number less than infinity and greater than negative infinity. Thus, here's what the initial situation looks like: Both are equivalent to

alpha beta pruning with the following problem:

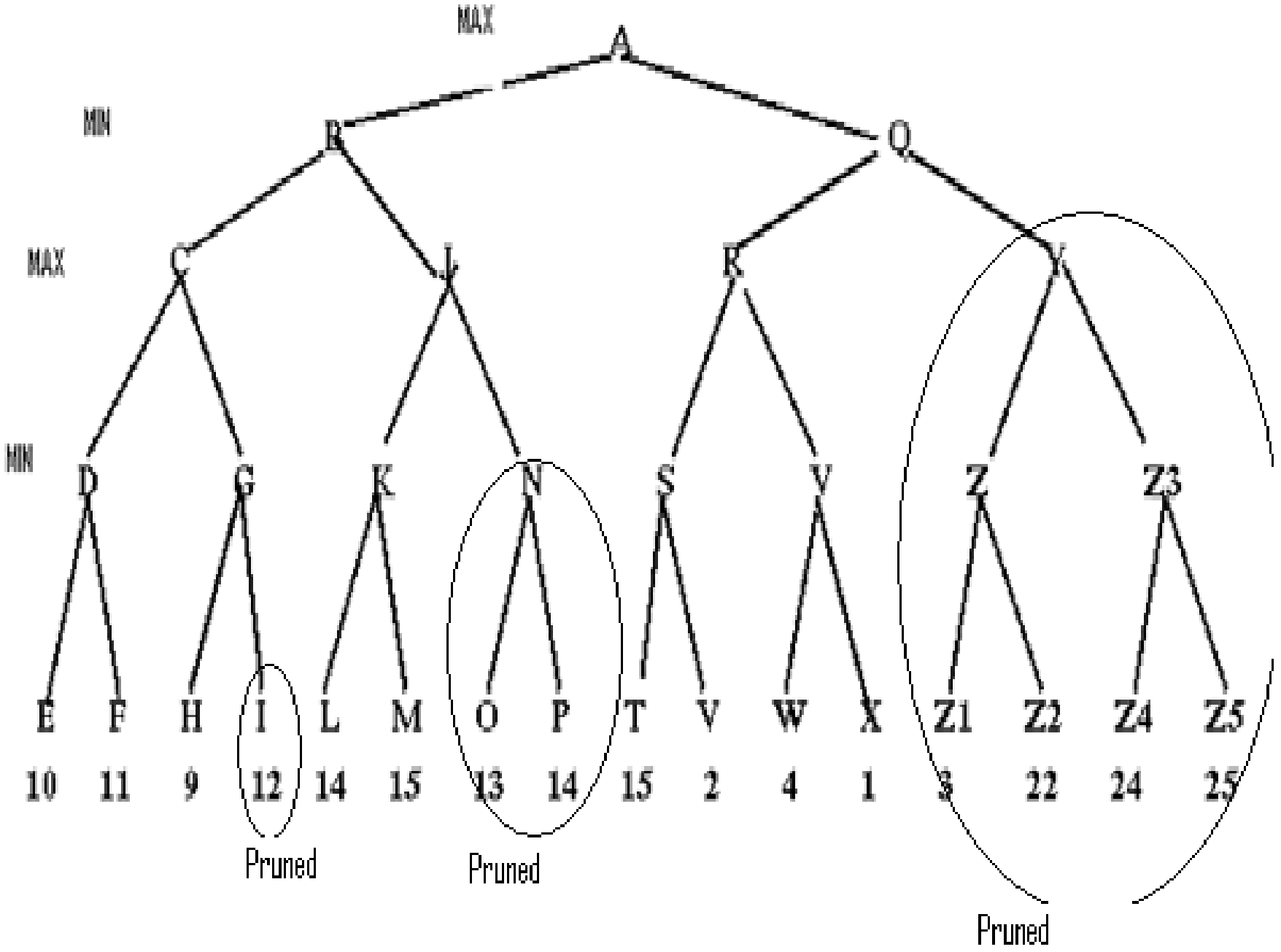


Solution: Alpha—beta pruning can be applied to trees of any depth, and it is often possible to prune entire subtrees rather than just leaves. The general principle is this: consider a node n somewhere in the tree, such that Player has a choice of moving to that node. If Player has a better choice *in,* either at the parent node of n or at any choice point further up. then n *will never be reached in actual play.* So once we have found out enough about n (by examining some of its descendants) to reach this conclusion, we can prune it.

At last min level when E comes then D≤10, F comes as 11 so now D is 10. Now at MAX level it is clear that C will at least be 10. Now at MIN level H comes as 9 so it is confirmed that G will be less than equal to 9. Now at above MAX level 10 has already been achieved so why to go for a value which is less than 9 So I will be pruned.

- So C is confirmed as 10. So at above MIN level B will at most be 10. Now at lower min level when L comes as 14, K is at most 14, after confirming M as 15 K is finalized as 14. So now at next MAX level J is at least 14. But at above MIN level B has already got 10 then there is no need to explore N. So it is pruned.

- So B is confirmed as 10. Now A is at least 10. Next T comes as 15. So S is at most 15. After getting U as 2, it is confirmed as 2. So next level R is at least 2. Now W comes as 4 So V is at most 4, So R could be 4 so X will be checked. After getting X as 1 V is confirmed as 1. So R is confirmed as 2. It means Q is at most 2. So No need to explore Y. it will be pruned.


MAX
A
MIN
B
Q
MAX
C
J
R
X
MIN
D
G
K
N
S
V
Z
Z3
E
F
H
I
L
M
O
P
T
V
W
X
Z1
Z2
Z4
Z5
10
11
9
12
14
15
13
14
15
2
4
1
3
22
24
25
Pruned
Pruned
Pruned