MODULE 2

ARTIFICIAL INTELLIGENCE

INFORMED SEARCH ALGORITHMS

- So far we have talked about the uninformed search algorithms which looked through search space for all possible solutions of the problem without having any additional knowledge about search space.
- But informed search algorithm contains an array of knowledge such as how far we are from the goal, path cost, how to reach to goal node, etc. This knowledge help agents to explore less to the search space and find more efficiently the goal node.
- The informed search algorithm is more useful for large search space. Informed search algorithm uses the idea of heuristic, so it is also called **Heuristic search**.

Heuristics function

- Heuristic is a function which is used in Informed Search, and it finds the most promising path.
- It takes the current state of the agent as its input and produces the estimation of how close agent is from the goal.
- The heuristic method, however, might not always give the best solution, but it guaranteed to find a good solution in reasonable time.
- Heuristic function estimates how close a state is to the goal. It is represented by **h(n)**, and it calculates the cost of an optimal path between the pair of states.
- The value of the heuristic function is always positive.

$$h(n) <= h^*(n)$$

- Here h(n) is heuristic cost, and h*(n) is the estimated cost. Hence heuristic cost should be less than or equal to the estimated cost.

Pure Heuristic Search

- Pure heuristic search is the simplest form of heuristic search algorithms. It expands nodes based on their heuristic value h(n). It maintains two lists, OPEN and CLOSED list. In the CLOSED list, it places those nodes which have already expanded and in the OPEN list, it places nodes which have yet not been expanded.
- On each iteration, each node n with the lowest heuristic value is expanded and generates all its successors and n is placed to the closed list. The algorithm continues unit a goal state is found.

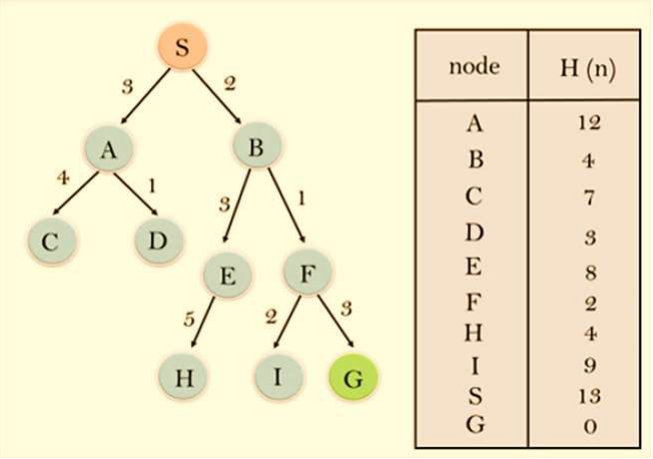
Best-first Search Algorithm (Greedy Search)

- Greedy best-first search algorithm always selects the path which appears best at that moment. It is the combination of depth-first search and breadth-first search algorithms.
- It uses the heuristic function and search. Best-first search allows us to take the advantages of both algorithms. With the help of best first search, at each step, we can choose the most promising node.
- In the best first search algorithm, we expand the node which is closest to the goal node and the closest cost is estimated by heuristic function, i.e.

$$h(n)= g(n)$$

- Were, h(n)= estimated cost from node n to the goal.

EXAMPLE



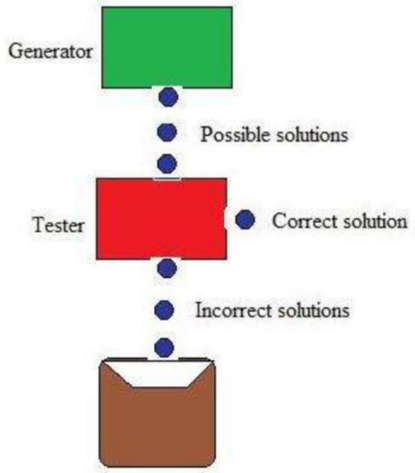
node	H (n)
A	12
B	4
C	7
D	3
E	8
F	2
H	4
I	9
S	13
G	0

- **Initialization:** Open [A, B], Closed [S]
- **Iteration 1:** Open [A], Closed [S, B]
- **Iteration2:** Open [E, F, A], Closed [S, B]
  : Open [E, A], Closed [S, B, F]
- **Iteration 3:** Open [I, G, E, A], Closed [S, B, F]
  : Open [I, E, A], Closed [S, B, F, G]
- Hence the final solution path will be: **S----> B----->F----> G**

GENERATE AND TEST

- The generate-and-test strategy is the simplest of all the approaches. It consists of the following steps:
- **Algorithm: Generate-and-Test**
- 1. Generate a possible solution. For some problems. this means generating a particular point in the problem space. For others, it means generating a path from a start state.
- 2. Test to see if this is actually a solution by comparing the chosen point or the endpoint of the chosen path to the set of acceptable goal states.
- 3. If a solution has been found, quit. Otherwise, return to step 1.

- The following diagram shows the Generate and Test Heuristic Search Algorithm



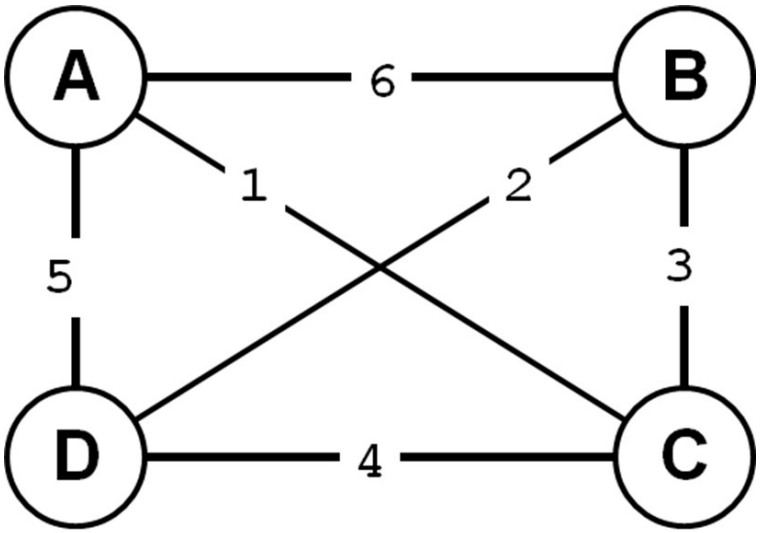
- To approaches are followed while generating solutions,
- Generate-and-test, like depth-first search, requires that complete solutions be generated for testing.
- Solutions can also be generated randomly but the solution is not guaranteed.

Example: coloured blocks

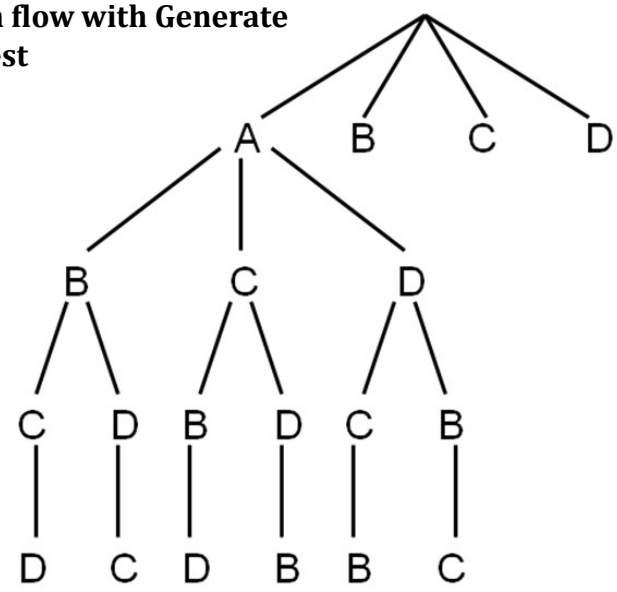
- "Arrange four 6-sided cubes in a row, with each side of each cube painted one of four colors, such that on all four sides of the row one block face of each color are showing."
- **Heuristic**: If there are more red faces than other colours then, when placing a block with several red faces, use few of them as possible as outside faces.

Example – Traveling Salesman Problem (TSP)

- A salesman has a list of cities, each of which he must visit exactly once. There are direct roads between each pair of cities on the list. Find the route the salesman should follow for the shortest possible round trip that both starts and finishes at any one of the cities.
- Traveler needs to visit n cities.
- Know the distance between each pair of cities.
- Want to know the shortest route that visits all the cities once.



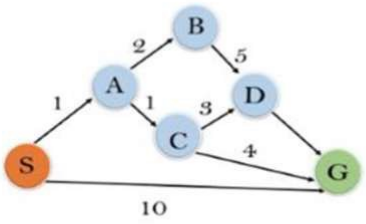
Search flow with Generate and Test



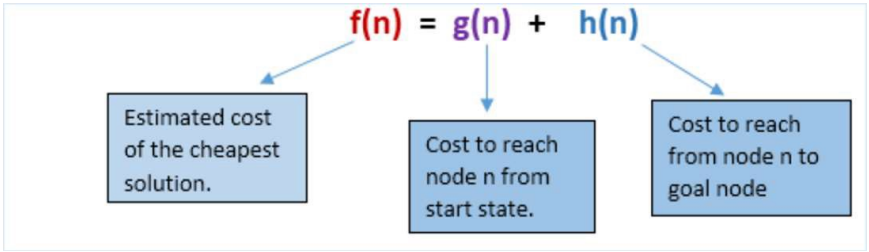
Search for	Path	Length of Path
1	ABCD	19
2	ABDC	18
3	ACBD	12
4	ACDB	13
5	ADBC	16
Continued		

Finally, select the path whose length is less.

A* Algorithm



State	h(n)
S	5
A	3
B	4
C	2
D	6
G	0



Given an initial state of a 8-puzzle problem and final state to be reached-

2	8	3
1	6	4
7		5

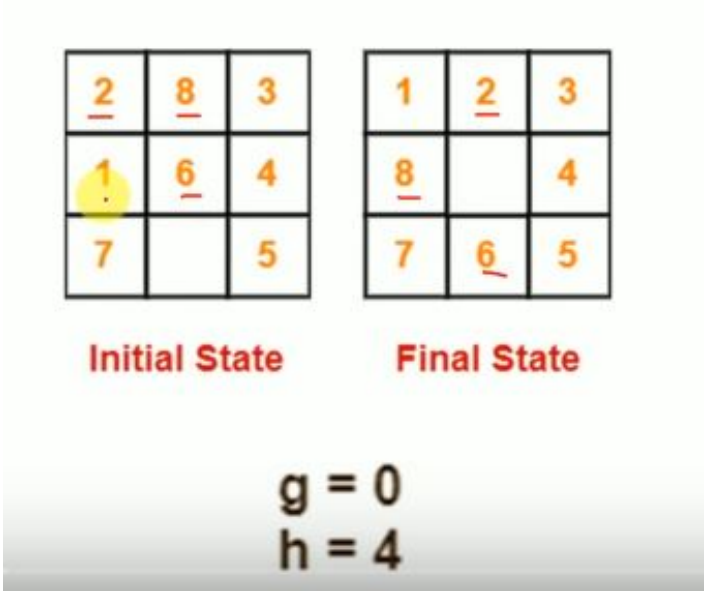
Initial State

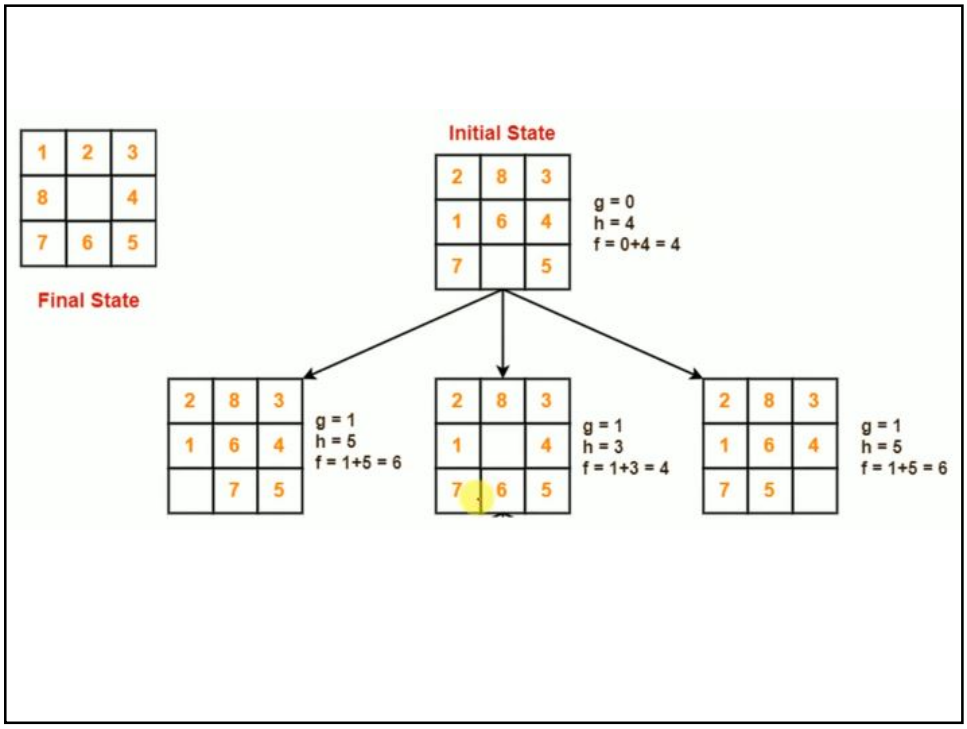
1	2	3
8		4
7	6	5

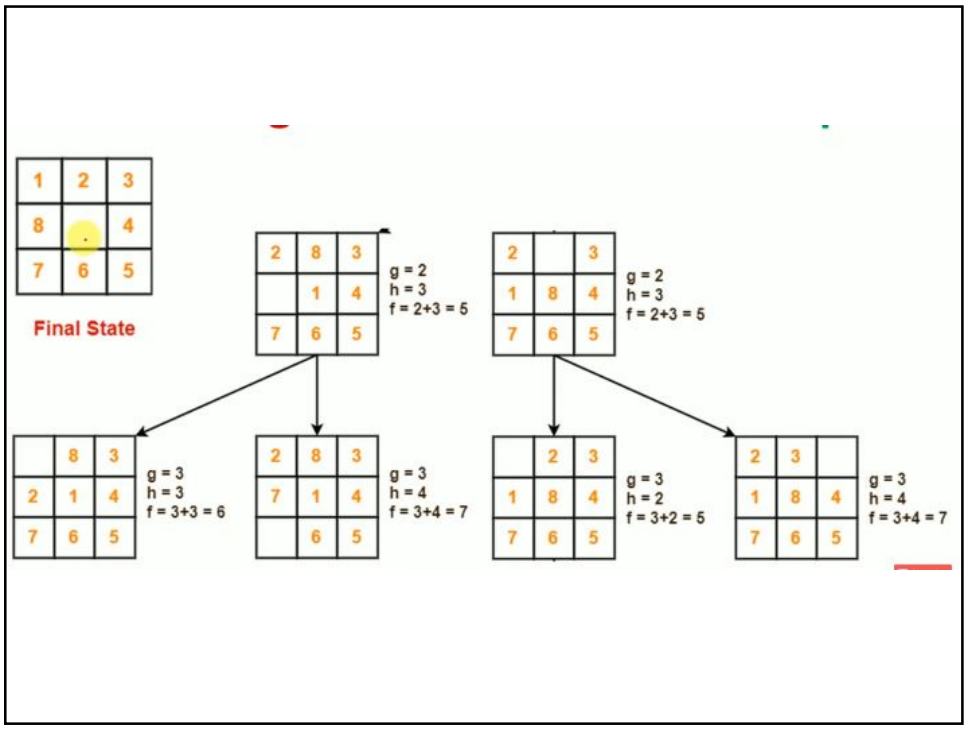
Final State

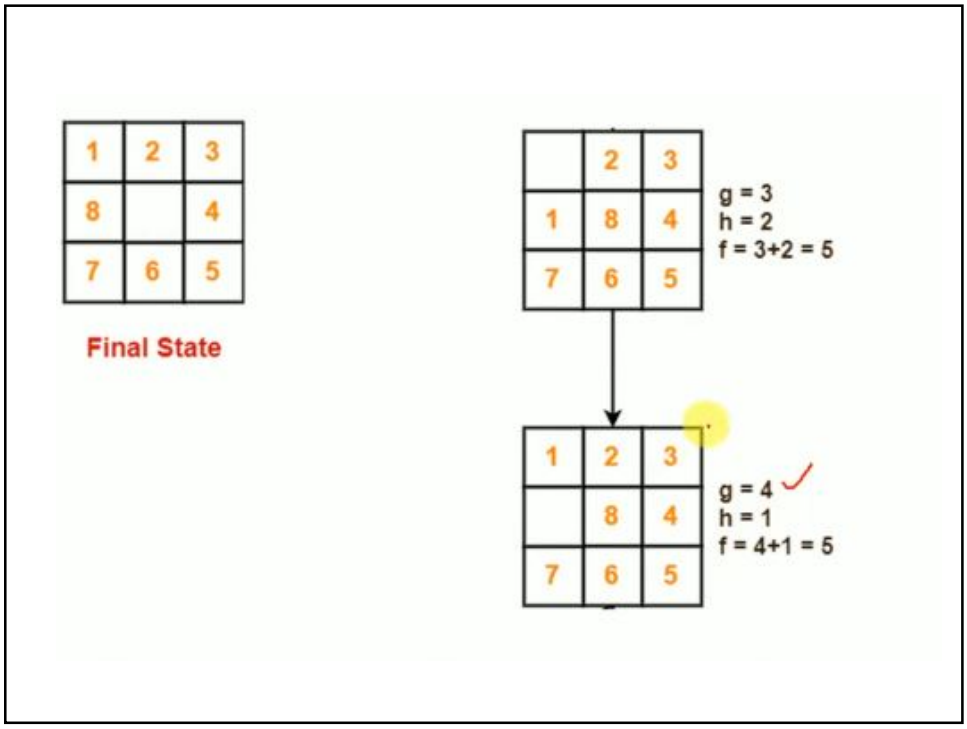
$$f(n) = g(n) + h(n)$$

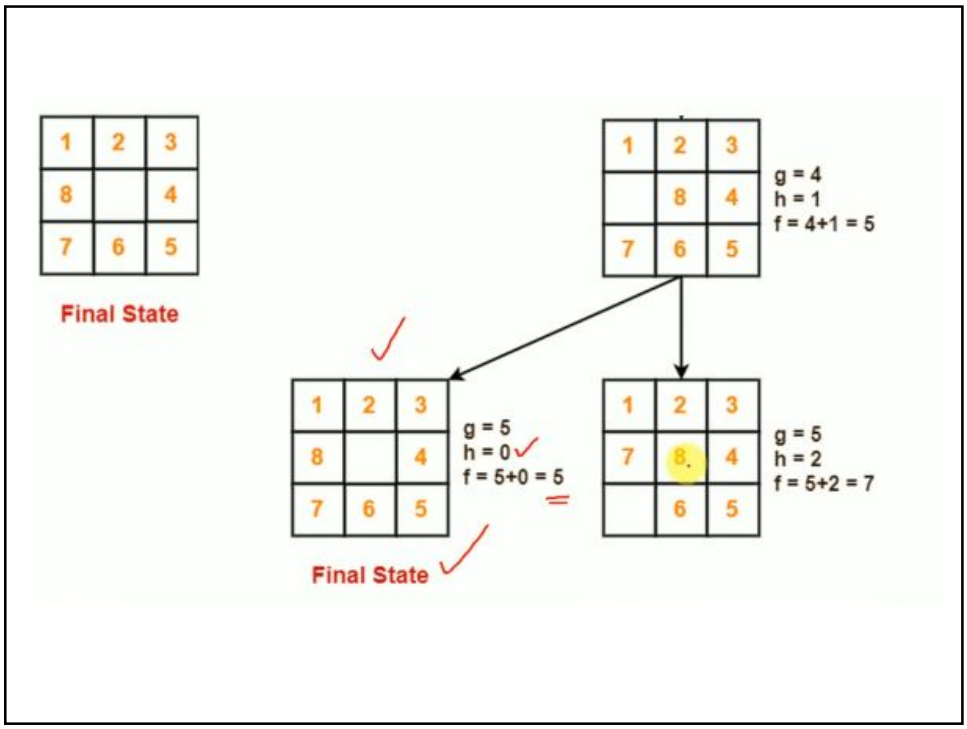
Consider **g(n) = Depth of node** and **h(n) = Number of misplaced tiles.**



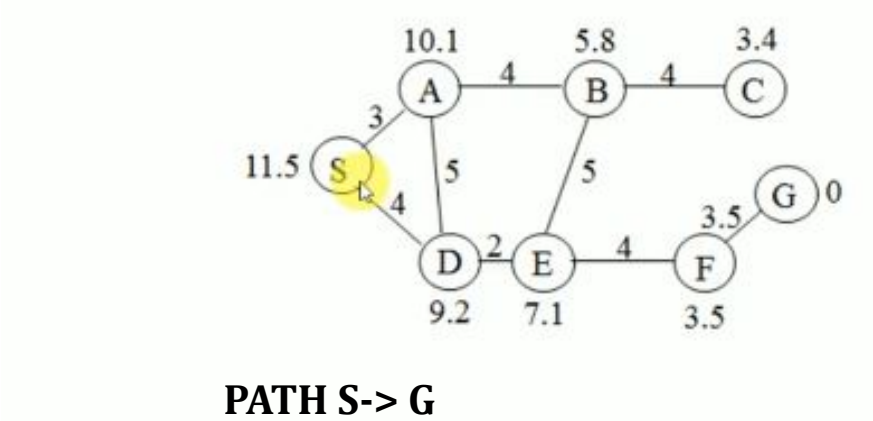

Final State
1 2 3
8 4
7 6 5
Initial State
2 8 3
1 6 4
7 5
g = 0
h = 4
f = 0+4 = 4
2 8 3
1 6 4
7 5
g = 1
h = 5
f = 1+5 = 6
2 8 3
1 4
7 6 5
g = 1
h = 3
f = 1+3 = 4
2 8 3
1 6 4
7 5
g = 1
h = 5
f = 1+5 = 6


Final State
1 2 3
8 4
7 6 5
2 8 3
1 4
7 6 5
g = 2
h = 3
f = 2+3 = 5
2 3
1 8 4
7 6 5
g = 2
h = 3
f = 2+3 = 5
8 3
2 1 4
7 6 5
g = 3
h = 3
f = 3+3 = 6
2 8 3
7 1 4
6 5
g = 3
h = 4
f = 3+4 = 7
2 3
1 8 4
7 6 5
g = 3
h = 2
f = 3+2 = 5
2 3
1 8 4
7 6 5
g = 3
h = 4
f = 3+4 = 7

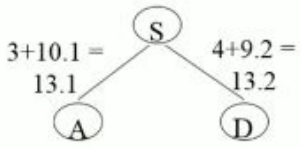

1 2 3
8 4
7 6 5
Final State
2 3
1 8 4
7 6 5
g = 3
h = 2
f = 3+2 = 5
1 2 3
8 4
7 6 5
g = 4
h = 1
f = 4+1 = 5

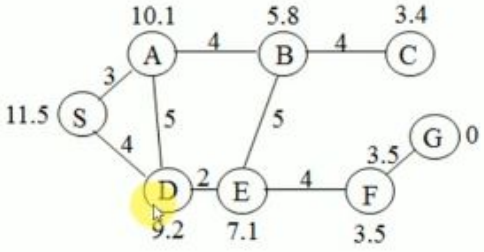

1 2 3
8 4
7 6 5
Final State
1 2 3
8 4
7 6 5
g = 4
h = 1
f = 4+1 = 5
1 2 3
8 4
7 6 5
g = 5
h = 0
f = 5+0 = 5
Final State
1 2 3
7 8 4
6 5
g = 5
h = 2
f = 5+2 = 7

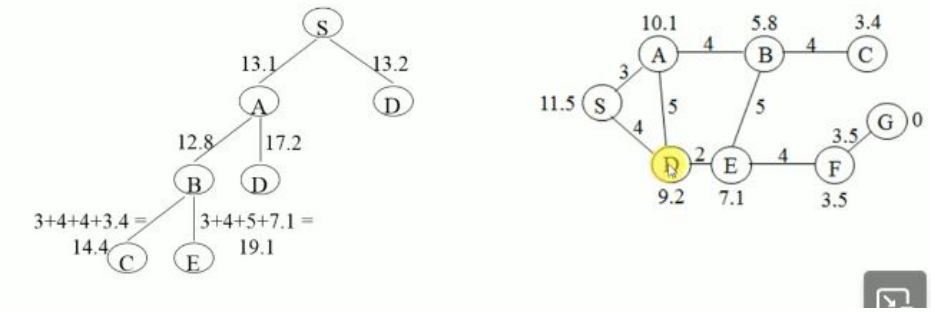
A* ALGORITHM PROBLEMS

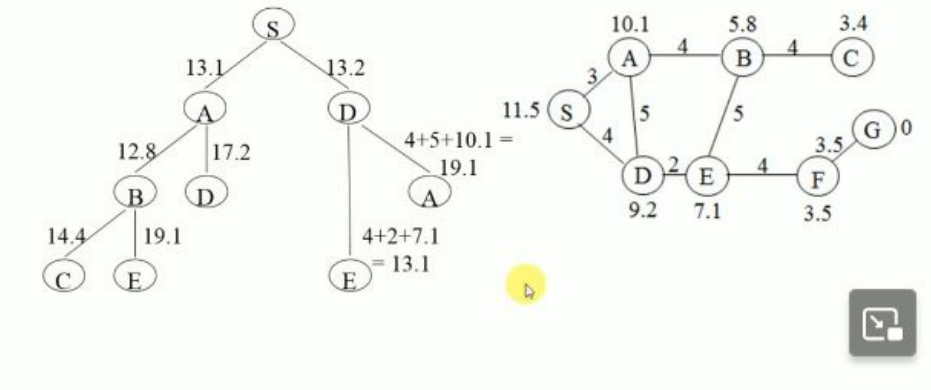


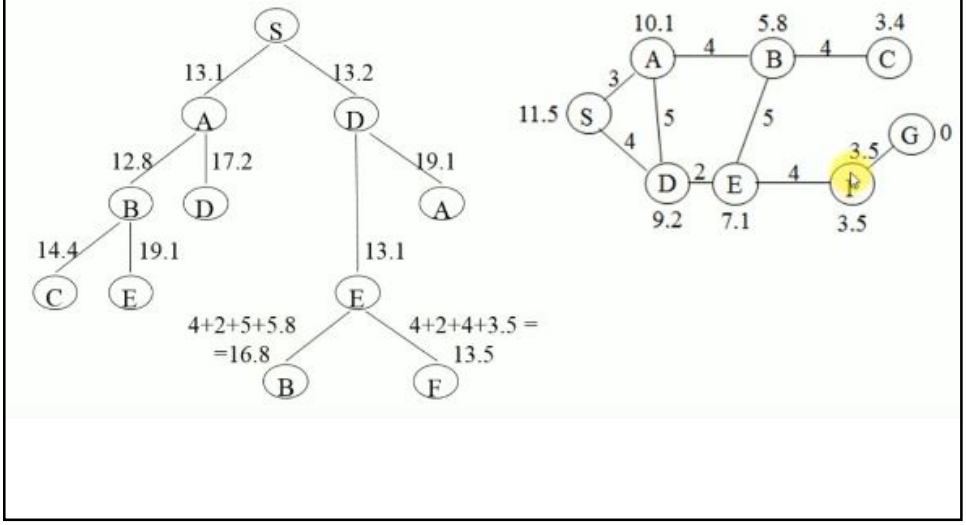
PATH S-> G

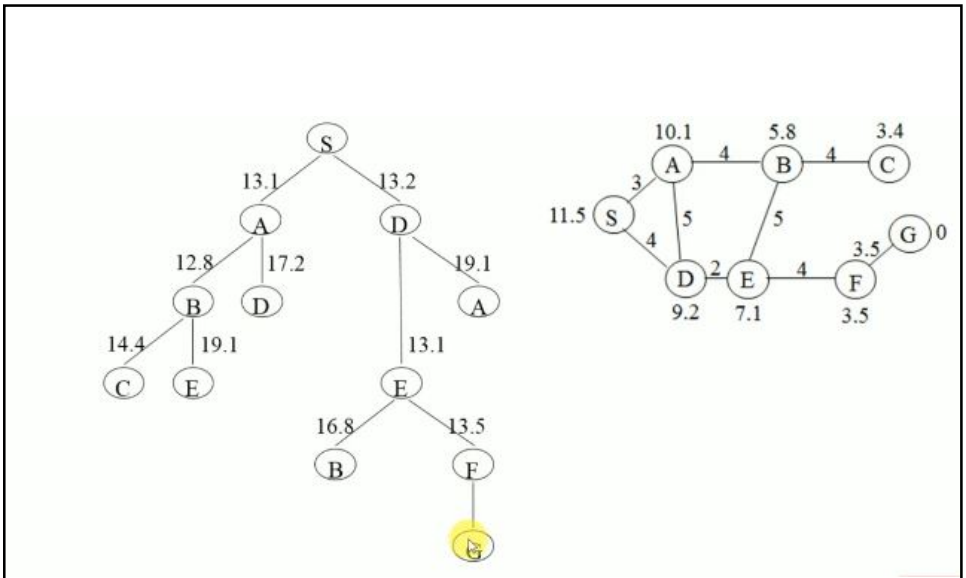




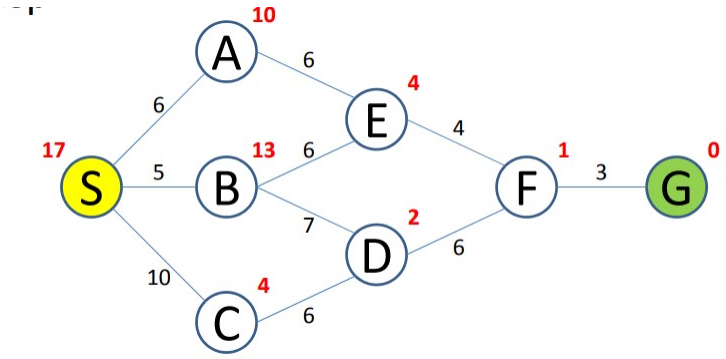

S
13.1
13.2
A
D
12.8
17.2
B
D
3+4+4+3.4 =
14.4
3+4+5+7.1 =
19.1
C
E
10.1
5.8
3.4
A
4
B
4
C
3
11.5
S
5
5
G
0
4
3.5
D
2
E
4
F
9.2
7.1
3.5


S
13.1
13.2
A
D
12.8
17.2
4+5+10.1 =
19.1
B
D
A
14.4
19.1
4+2+7.1
= 13.1
C
E
E
10.1
5.8
3.4
A
4
B
4
C
3
11.5
S
5
5
G
0
4
3.5
D
2
E
4
F
9.2
7.1
3.5

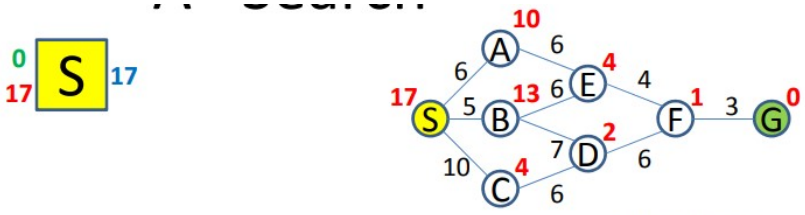

S
13.1
13.2
A
D
12.8
17.2
19.1
B
D
A
14.4
19.1
13.1
C
E
E
4+2+5+5.8
=16.8
4+2+4+3.5 =
13.5
B
F
10.1
5.8
3.4
A
4
B
4
C
3
11.5
S
5
5
G
0
4
3.5
D
2
E
4
F
9.2
7.1
3.5


S
13.1
13.2
A
D
12.8
17.2
19.1
B
D
A
14.4
19.1
13.1
C
E
E
16.8
13.5
B
F
G
10.1
5.8
3.4
A
4
B
4
C
3
11.5
S
5
5
G
0
4
3.5
D
2
E
4
F
9.2
7.1
3.5

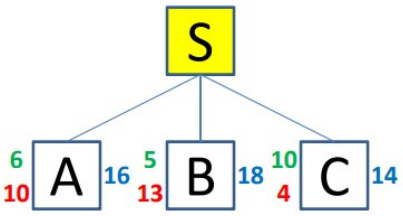
A* ALGORITHM PROBLEMS

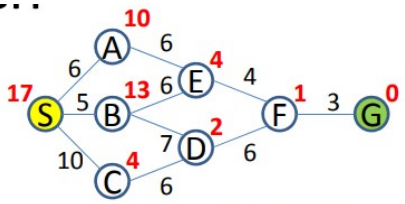


PATH S-> G

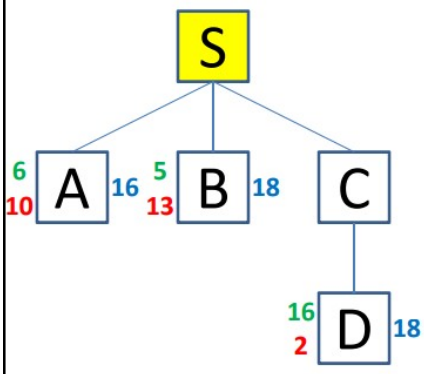


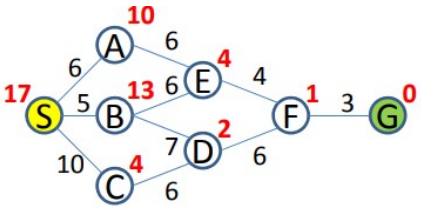
QUEUE:
S



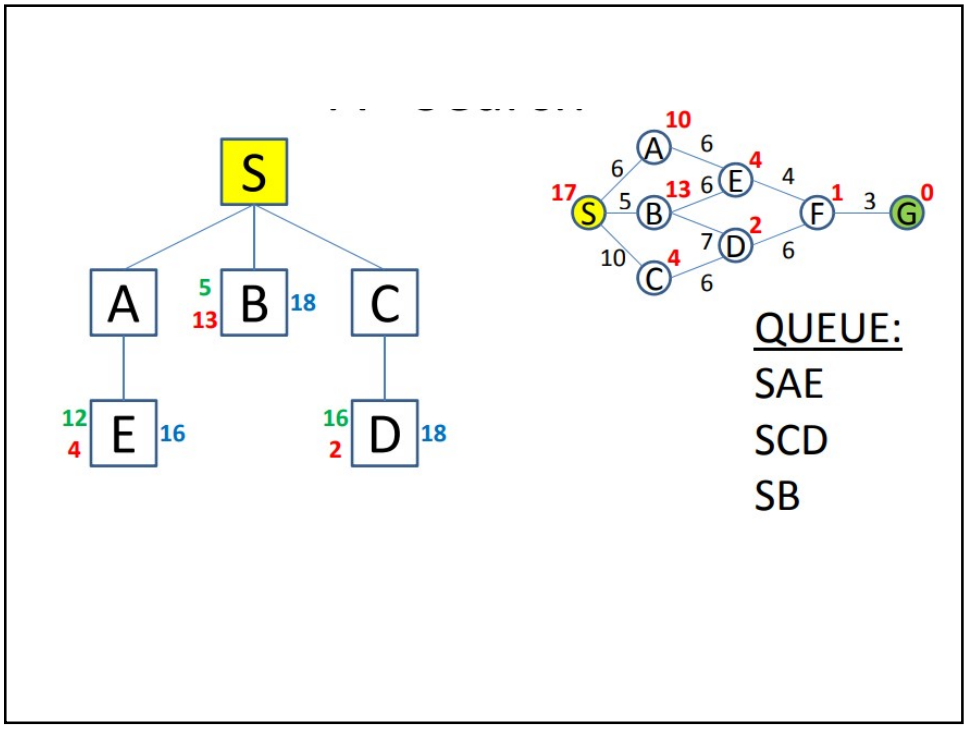


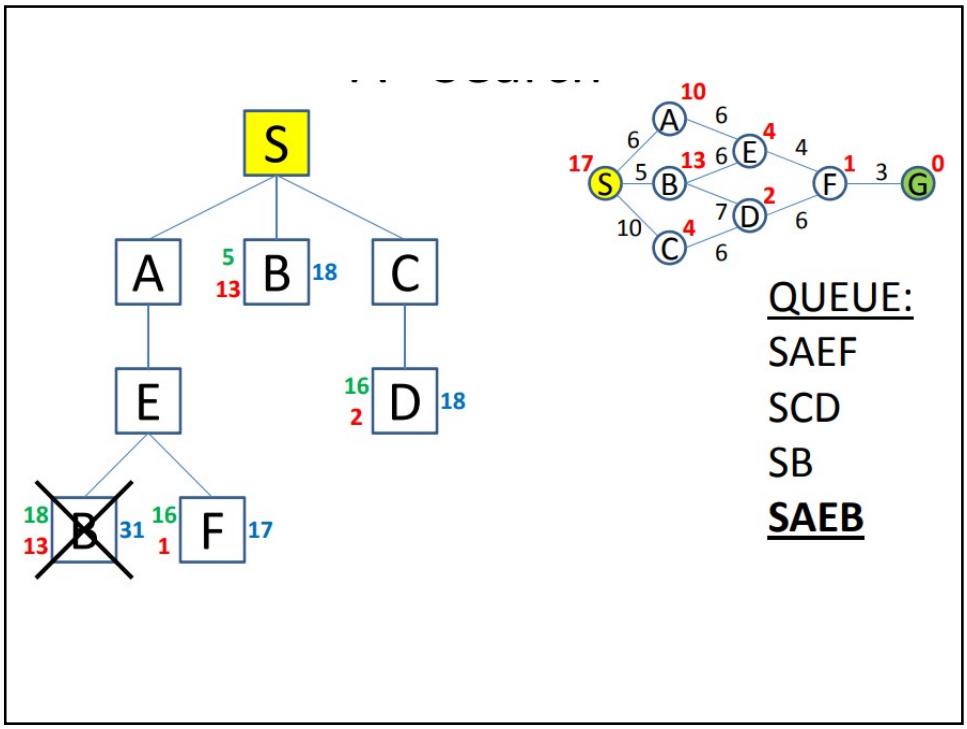
QUEUE:
SC
SA
SB

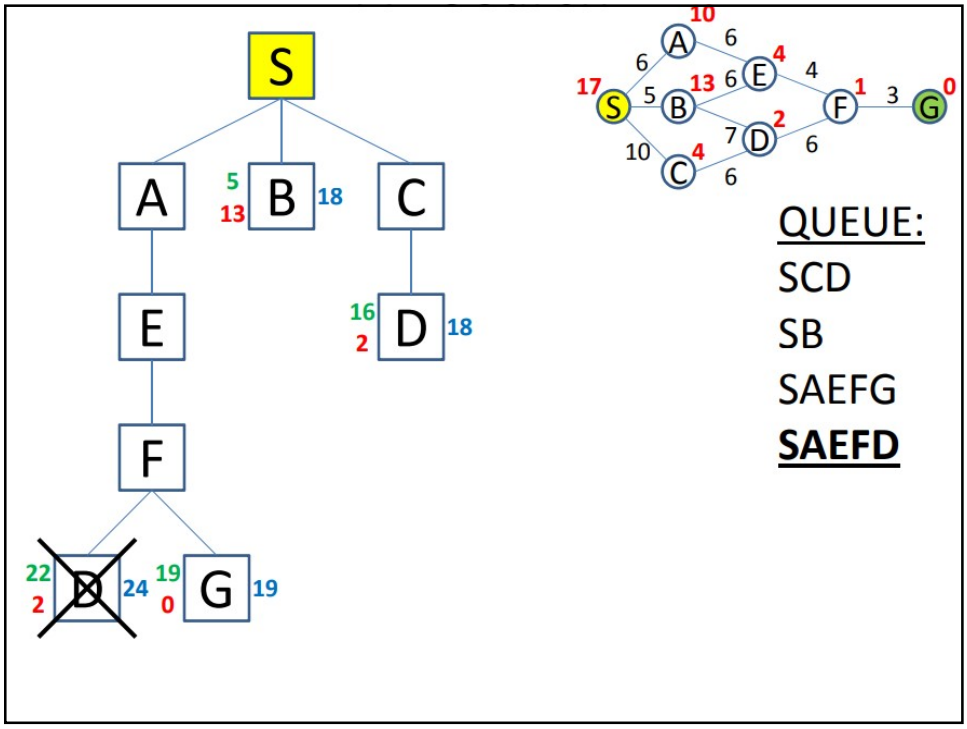


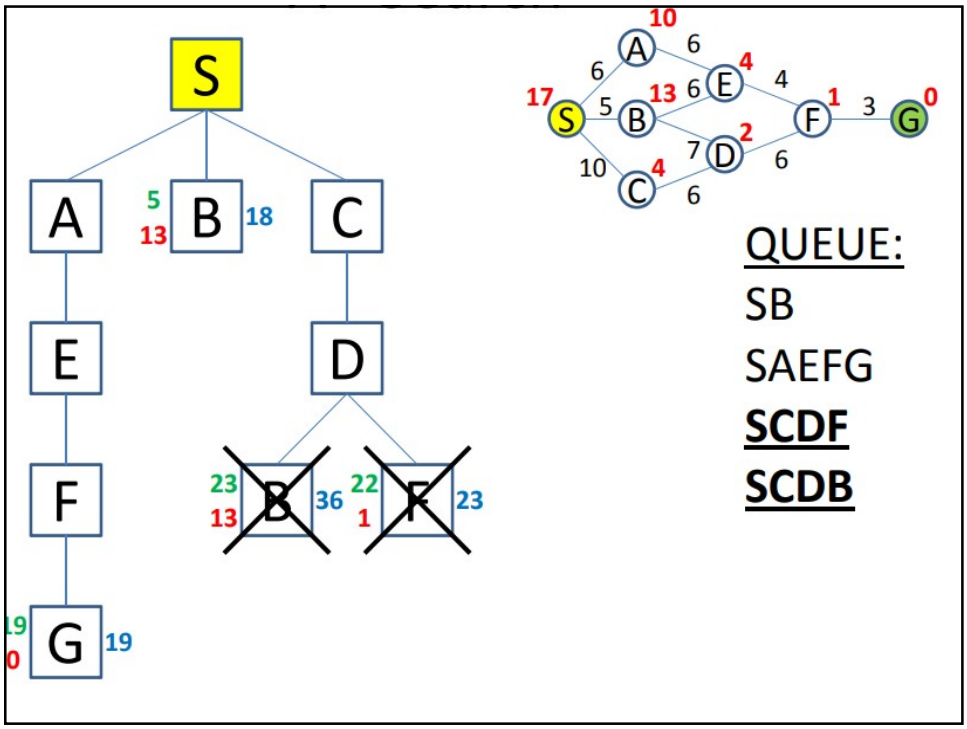


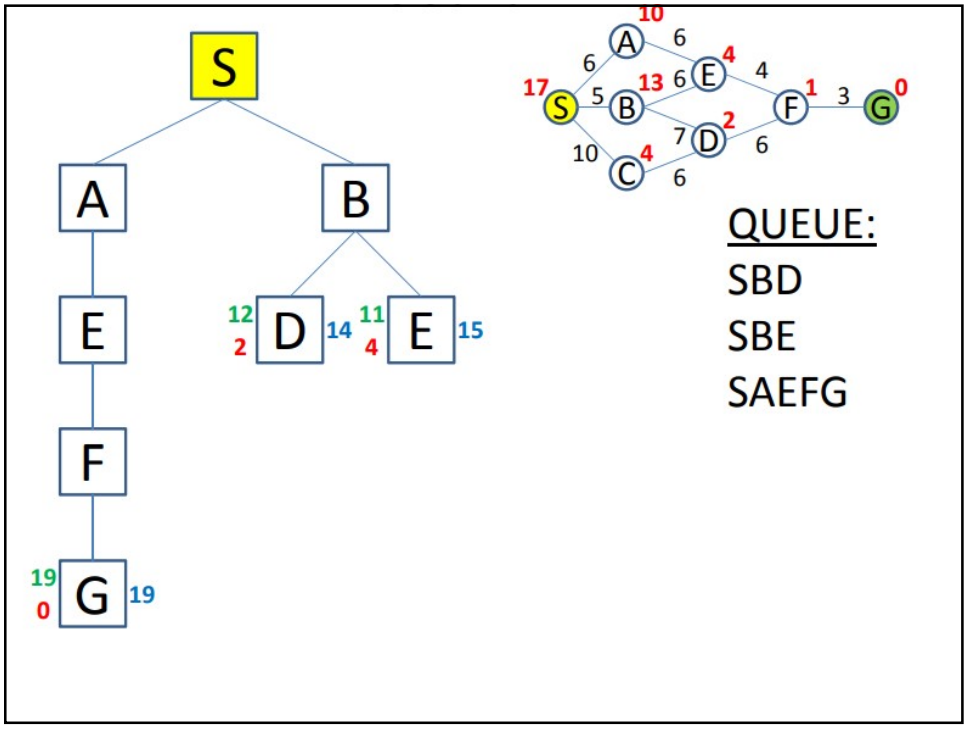
QUEUE:
SA
SCD
SB

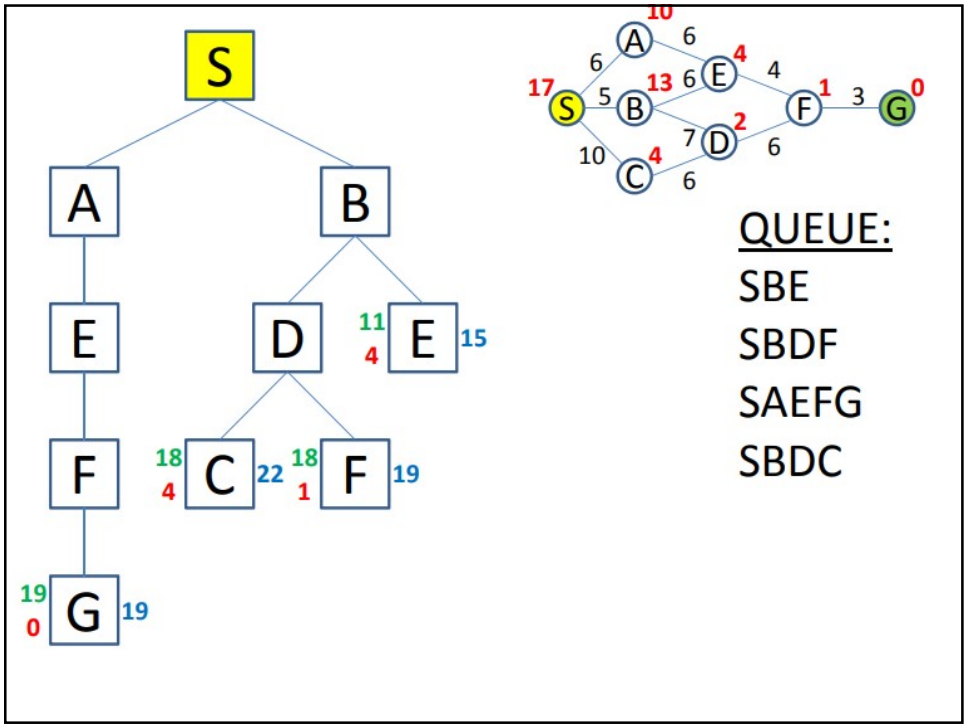

S
A
B
C
E
D
5
13
18
12
4
16
16
2
18
10
6
6
4
4
17
S
5
13
6
E
1
3
0
B
F
G
2
7
D
10
4
6
6
C
QUEUE:
SAE
SCD
SB

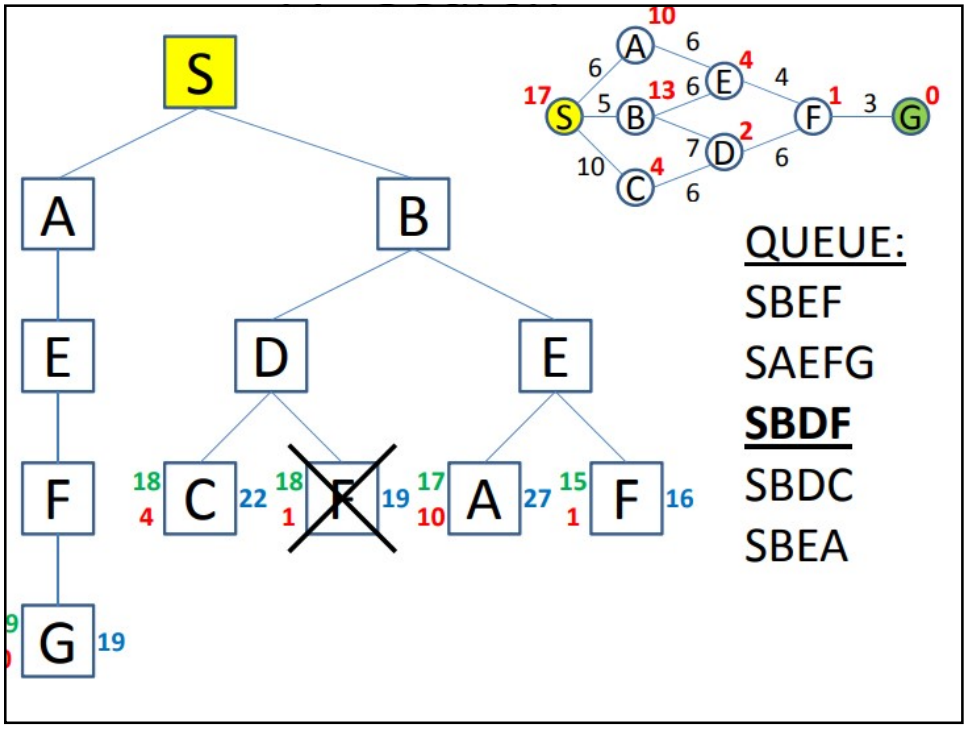

S
A
B
C
E
D
F
5
13
18
16
2
18
18
13
31
16
1
17
10
6
6
4
4
17
S
5
13
6
E
1
3
0
B
F
G
2
7
D
10
4
6
6
C
QUEUE:
SAEF
SCD
SB
SAEB

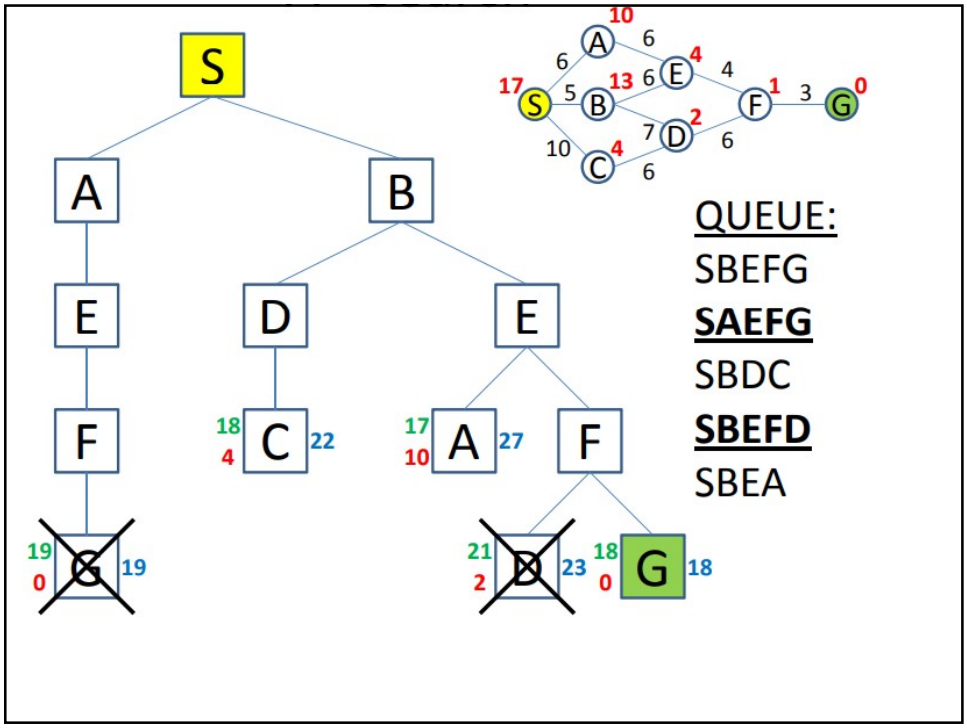

S
A
5
13
B
18
C
E
16
2
D
18
F
22
2
B
24
19
0
G
19
10
17
S
A
E
B
F
G
D
C
QUEUE:
SCD
SB
SAEFG
SAEFD


S
A
5
13
B
18
C
E
D
F
23
13
B
36
22
1
F
23
19
0
G
19
10
17
S
A
E
B
F
G
D
C
QUEUE:
SB
SAEFG
SCDF
SCDB

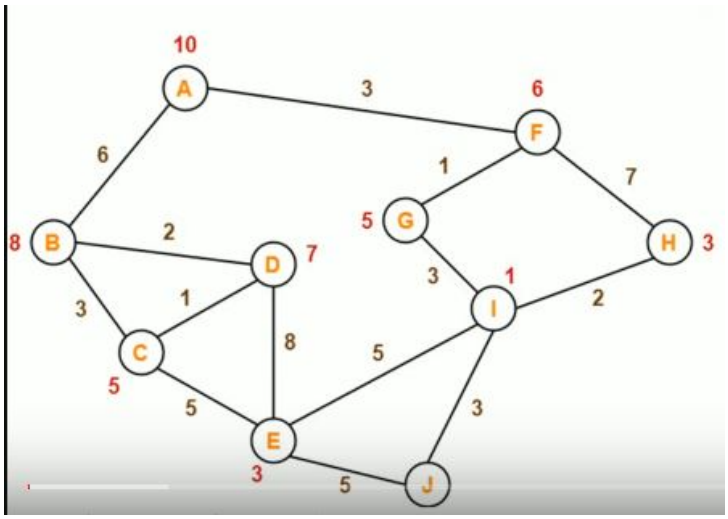

S
A
B
E
F
G
D
E
19
0
19
12
2
14
11
4
15
QUEUE:
SBD
SBE
SAEFG


S
A
B
E
F
G
D
E
C
F
19
0
19
11
4
15
18
4
22
18
1
19
QUEUE:
SBE
SBDF
SAEFG
SBDC


S
A
B
E
D
E
F
C
F
A
F
G
QUEUE:
SBEF
SAEFG
SBDF
SBDC
SBEA


S
A
B
E
D
E
F
C
A
F
G
D
G
QUEUE:
SBEFG
SAEFG
SBDC
SBEFD
SBEA

A* ALGORITHM PROBLEMS



PATH A-> J

Step-01:

- We start with node A.
- Node B and Node F can be reached from node A.
- A* Algorithm calculates

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

- $f(B) = g(B) + h(B) = 6 + 8 = 14$
- $f(F) = g(F) + h(F) = 3 + 6 = 9$
- Since $f(F) < f(B)$, so it decides to go to node F.
- Path- A → F

Step-02:

- Node G and Node H can be reached from node F.
- A* Algorithm calculates

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

- $f(G) = g(G) + h(G) = (3+1) + 5 = 9$
- $f(H) = g(H) + h(H) = (3+7) + 3 = 13$
- Since $f(G) < f(H)$, so it decides to go to node G.
- Path - A → F → G

Step-03:

- Node I can be reached from node G.
- A* Algorithm calculates

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

- $f(I) = g(I) + h(I)$

$$= (3+1+3) + 1 = 8$$

- So it decides to go to node I.
- Path - A → F → G → I

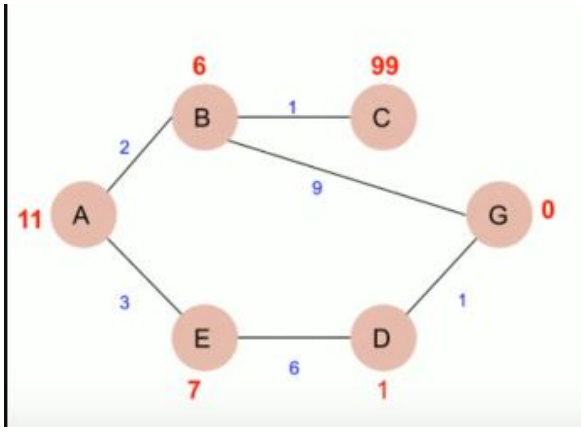
Step-04:

- Node E, Node H and Node J can be reached from node I.
- A* Algorithm calculates

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

- f(E) = (3+1+3+5) + 3 = 15
- f(H) = (3+1+3+2) + 3 = 12
- f(J) = (3+1+3+3) + 0 = 10
- Since f(J) is least, so it decides to go to node J.
- Path - A → F → G → I → J

A* ALGORITHM PROBLEMS



PATH A-> G

- Now from A, we can go to point B or E, so we compute f(x) for each of them,
- A → B = g(B) + h(B) = 2 + 6 = 8
- A → E = g(E) + h(E) = 3 + 7 = 10
- Since the cost for A → B is less, we move forward with this path and compute the f(x) for the children nodes of B

- Now from B, we can go to point C or G, so we compute f(x) for each of them,
- A → B → C = (2 + 1) + 99= 102
- A → B → G = (2 + 9) + 0 = 11

- Now from A, we can go to point B or E, so we compute f(x) for each of them,
- A → B = g(B) + h(B) = 2 + 6 = 8
- A → E = g(E) + h(E) = 3 + 7 = 10
- Since the cost for A → B is less, we move forward with this path and compute the f(x) for the children nodes of B

- Now from E, we can go to point D, so we compute f(x),
- A → E → D = (3 + 6) + 1 = 10

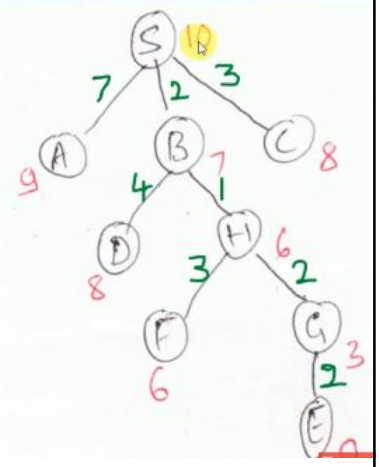
- Now compute the f(x) for the children of D
- A → E → D → G = (3 + 6 + 1) +0 = **10**
- Now comparing all the paths that lead us to the goal, we conclude that A → E → D → G is the most cost-effective path to get from A to G.

A* Search Algorithm Explained - Artificial Intelligence

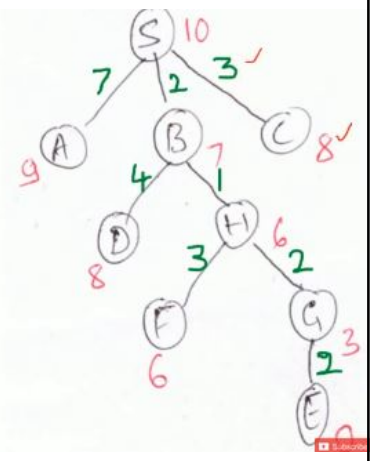
1. Start with OPEN containing only the initial node. Set that node's **g** value to **0**, its **h'** value to whatever it is, and its **f'** value to **h' + 0**, or **h'**. Set CLOSED to the empty list.
2. Until a goal node is found, repeat the following procedure: If there are no nodes on OPEN. report failure. Otherwise, pick the node on OPEN with the lowest **f'** value. Call it BESTNODE. Remove it from OPEN. Place on CLOSED. See if BESTNODE is a goal node. If so, exit and report a solution. Otherwise, generate the successors of BESTNODE. For each such SUCCESSOR, do the following:
   a) Set SUCCESSOR to point back to BESTNODE. These backwards 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) Compute f'(SUCCESSOR) = g(SUCCESSOR) + h'(SUCCESSOR)
   d) 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.

Heuristic Search – A* Search Solved Example

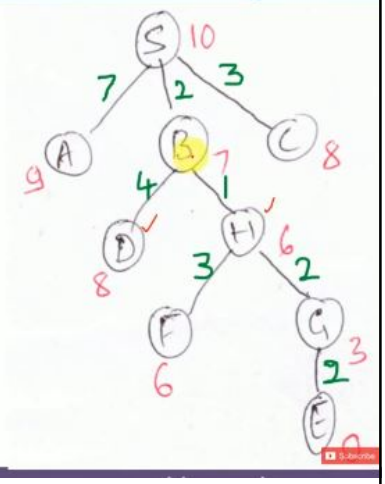
Open					Closed	
Node	g(n)	h(n)	f(n)		Node	Parent
S	0	10	10			



Open					Closed	
Node	g(n)	h(n)	f(n)		Node	Parent
A	7	9	16		S	
B	2	7	9			
C	3	8	11			



Open					Closed	
Node	g(n)	h(n)	f(n)		Node	Parent
					S	
C	3	8	11		B	S ✓
A	7	9	16			



Types of algorithms in Adversarial search

- In a normal search, we follow a sequence of actions to reach the goal or to finish the game optimally. But in an adversarial search, the result depends on the players which will decide the result of the game.
- It is also obvious that the solution for the goal state will be an optimal solution because the player will try to win the game with the shortest path and under limited time.
- There are following types of adversarial search:

1. Min max Algorithm
2. Alpha-beta Pruning

MIN MAX ALGORITHM

- Min-max algorithm is a recursive or backtracking algorithm which is used in decision-making and game theory. It provides an optimal move for the player assuming that opponent is also playing optimally.
- Min-Max algorithm uses recursion to search through the game tree.
- Min-Max algorithm is mostly used for game playing in AI, such as Chess, Checkers, tic-tac-toe, go etc.
- This Algorithm computes the min-max decision for the current state.

- In this algorithm two players play the game, one is called MAX and other is called MIN.
- MAX will select the maximized value
- MIN will select the minimized value.
- The min-max algorithm performs a depth-first search algorithm for the exploration of the complete game tree.
- The min-max algorithm proceeds all the way down to the terminal node of the tree, then backtrack the tree as the recursion

Minimax search procedure

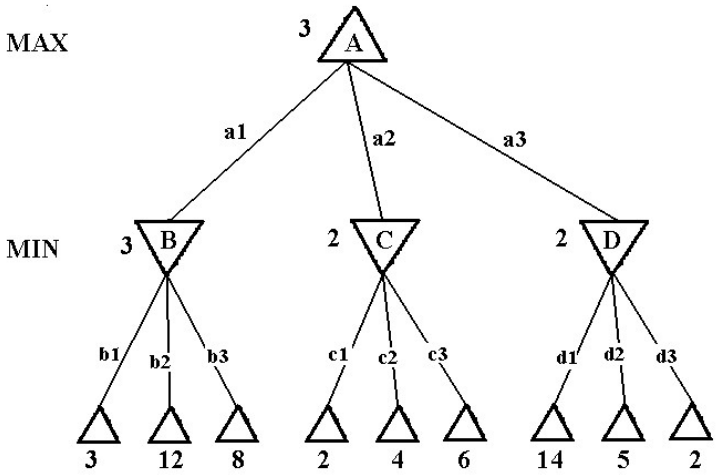
- Consider games with 2 players. The opponents in a game are referred to as MIN and MAX.
- MAX represents the player trying to win or to MAXimize his advantage.
- MIN is the opponent who attempts to MINimize MAX's score.
- MAX moves first, and then they take turns moving until the game is over.
- At the end of the game, points are awarded to the winning player and penalties are given to the loser.

Search strategies

Consider the game tree

- The ▲ nodes are MAX nodes, in which it is MAX's turn to move and the nodes▼ are MIN nodes. The terminal states show the utility values for MAX.
- The possible moves for MAX at the root node are labeled a1, a2, a3. the possible replies to a1 for MIN are b1, b2, b3 and so on. This game ends after one move each by MAX and MIN.
- Given a game tree, the optimal strategy can be determined by examining the min-max value of each node, which we write as min max value (n).
- The min max-value of a node is the utility for MAX of being in the corresponding state.
- The minmax-value of a terminal state is just its utility.

Example



MAX will prefer to move to a state of maximum value, whereas MIN prefers a state of minimum value.
So we have

$$\text{Minimax-value}(n) = \begin{cases} \text{Utility}(n), \textbf{ if n is a terminal state} \\ \text{MAX}_{s \in \text{successors}(n)} \text{minimax-value}(s), \textbf{ If n is a MAX node} \\ \text{MIN}_{s \in \text{successors}(n)} \text{minimax-value}(s), \textbf{ If n is a MIN node} \end{cases}$$

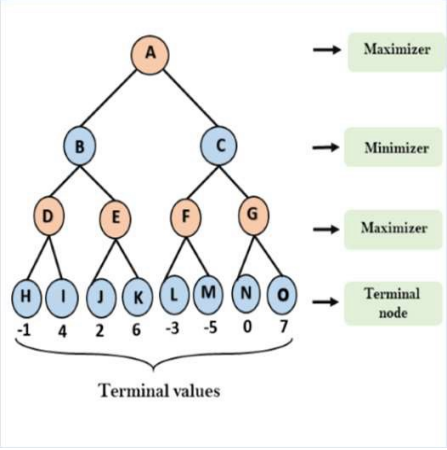
- Minmax-value(A) = max [min (3,12,8), min (2,4,6), min (14,5,2)]
  = max [3, 2, 2] = 3

Working of Min-Max Algorithm

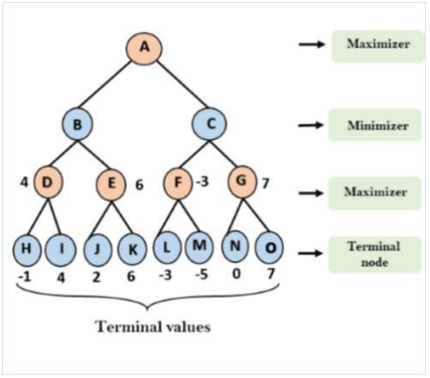
- The working of the minmax algorithm can be easily described using an example. Below we have taken an example of game-tree which is representing the two-player game.
- In this example, there are two players one is called Maximizer and other is called Minimizer.
- Maximizer will try to get the Maximum possible score, and Minimizer will try to get the minimum possible score.
- This algorithm applies DFS, so in this game-tree, we have to go all the way through the leaves to reach the terminal nodes.
- At the terminal node, the terminal values are given so we will compare those value and backtrack the tree until the initial state occurs.

Steps involved in solving the two-player game tree

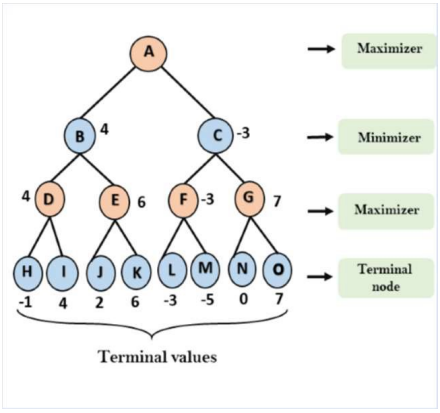
- Step-1: In the first step, the algorithm generates the entire game-tree and apply the utility function to get the utility values for the terminal states.
- In the tree diagram, let's take A is the initial state of the tree.
- Suppose maximizer takes first turn which has worst-case initial value = -infinity, and minimizer will take next turn which has worst-case initial value = +infinity.



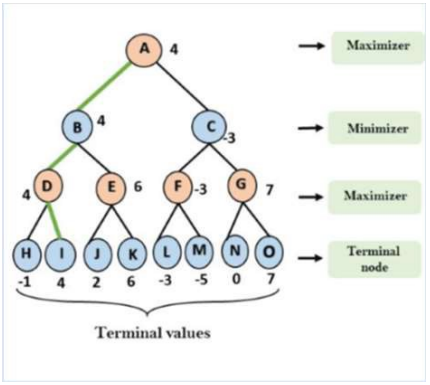
- Step 2: Now, first we find the utilities value for the Maximizer, its initial value is -∞, so we will compare each value in terminal state with initial value of Maximizer and determines the higher nodes values. It will find the maximum among the all.
- For node D: max(-1, -∞) => max(-1,4)= 4
- For Node E : max(2, -∞) => max(2, 6)= 6
- For Node F :max(-3, -∞) => max(-3,-5)= -3
- For node G: max(0, -∞) = max(0, 7) = 7



- Step 3: In the next step, it's a turn for minimizer, so it will compare all nodes value with +∞, and will find the 3rd layer node values.
- For node B= min(4,6) = 4
- For node C= min (-3, 7) = -3



- Step 4: Now it's a turn for Maximizer, and it will again choose the maximum of all nodes value and find the maximum value for the root node.
- In this game tree, there are only 4 layers, hence we reach immediately to the root node, but in real games, there will be more than 4 layers.
- For node A max(4, -3)= 4

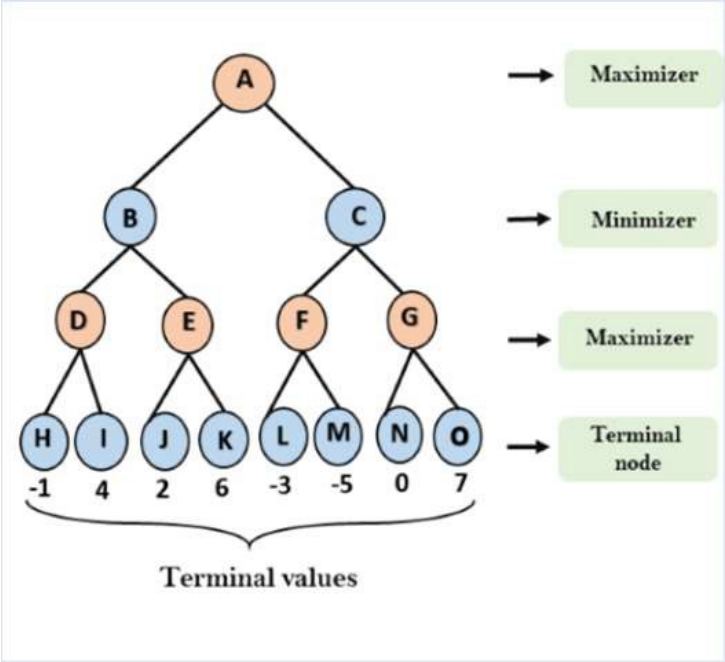


Properties of Mini-Max algorithm

- Complete- Min-Max algorithm is Complete. It will definitely find a solution (if exist), in the finite search tree.
- Optimal- Min-Max algorithm is optimal if both opponents are playing optimally.
- Time complexity- As it performs DFS for the game-tree, so the time complexity of Min-Max algorithm is O(bm), where b is branching factor of the game-tree, and m is the maximum depth of the tree.
- Space Complexity- Space complexity of Mini-max algorithm is also similar to DFS which is O(bm).

Limitation of the minimax Algorithm

- The main drawback of the minimax algorithm is that it gets really slow for complex games such as Chess, go, etc.
- This type of games has a huge branching factor, and the player has lots of choices to decide.



ALPHA BETA PRUNING

- Alpha-beta pruning is a modified version of the minmax algorithm.
- It is an optimization technique for the minimax algorithm.
- There is a technique by which without checking each node of the game tree we can compute the correct minimax decision, and this technique is called pruning.
- This involves two threshold parameter Alpha and Beta for future expansion, so it is called alpha-beta pruning. It is also called as Alpha-Beta Algorithm.
- Alpha-beta pruning can be applied at any depth of a tree, and sometimes it not only prune the tree leaves but also entire subtree.

The two-parameter

- Alpha: The best (highest-value) choice we have found so far at any point along the path of Maximizer. The initial value of alpha is -∞.
- Beta: The best (lowest-value) choice we have found so far at any point along the path of Minimizer. The initial value of beta is +∞.
- The Alpha-beta pruning to a standard minimax algorithm returns the same move as the standard algorithm does, but it removes all the nodes which are not really affecting the final decision but making algorithm slow. Hence by pruning these nodes, it makes the algorithm fast.

Condition for Alpha-beta pruning:

- The main condition which required for alpha-beta pruning is:

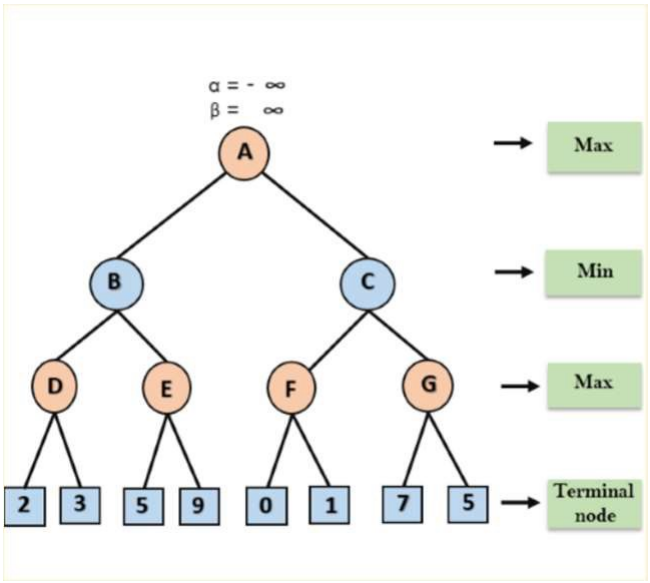
$$\alpha >= \beta$$

Key points about alpha-beta pruning

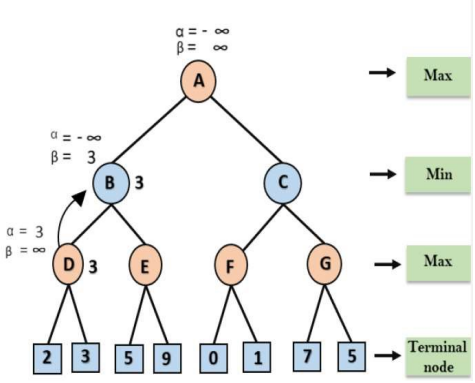
- The Max player will only update the value of alpha.
- The Min player will only update the value of beta.
- While backtracking the tree, the node values will be passed to upper nodes instead of values of alpha and beta.
- We will only pass the alpha, beta values to the child nodes.

Working of Alpha-Beta Pruning

- Let's take an example of two-player search tree to understand the working of Alpha-beta pruning
- Step 1: At the first step the, Max player will start first move from node A where $\alpha = -\infty$ and $\beta = +\infty$, these value of alpha and beta passed down to node B where again $\alpha = -\infty$ and $\beta = +\infty$, and Node B passes the same value to its child D.

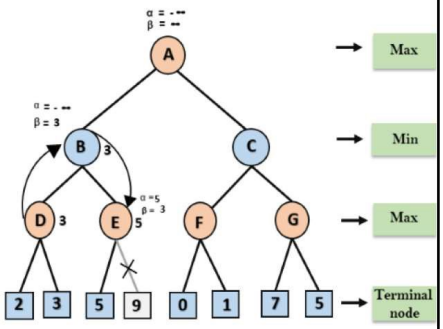


- **Step 2:** At Node D, the value of α will be calculated as its turn for Max. The value of α is compared with firstly 2 and then 3, and the max (2, 3) = 3 will be the value of α at node D and node value will also 3.
- **Step 3:** Now algorithm backtrack to node B, where the value of β will change as this is a turn of Min, Now β= +∞, will compare with the available subsequent nodes value, i.e. min (∞, 3) = 3, hence at node B now α= -∞, and β= 3.

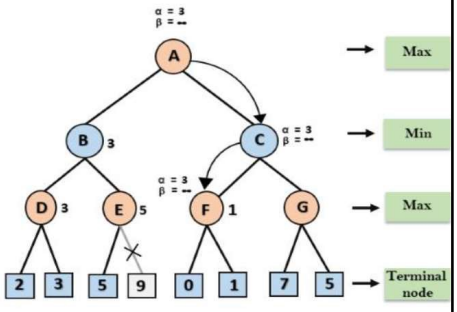


- In the next step, algorithm traverse the next successor of Node Bwhich is node E, and the values of α= -∞, and β= 3 will also be passed.

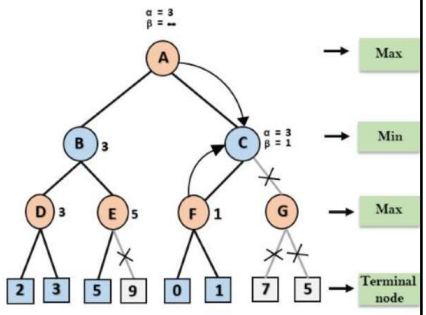
- **Step 4:** At node E, Max will take its turn, and the value of alpha will change. The current value of alpha will be compared with 5, so max (-∞, 5) = 5, hence at node E α= 5 and β= 3, where α>=β, so the right successor of E will be pruned, and algorithm will not traverse it, and the value at node E will be 5.



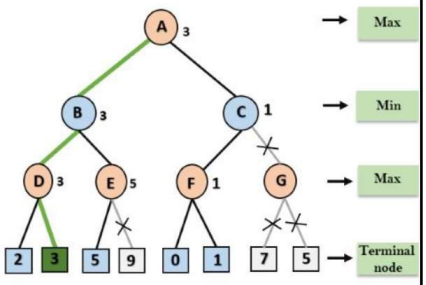
- **Step 5:** At next step, algorithm again backtrack the tree, from node B to node A. At node A, the value of alpha will be changed the maximum available value is 3 as max (-∞, 3)= 3, and β= +∞, these two values now passes to right successor of A which is Node C.
- At node C, α=3 and β= +∞, and the same values will be passed on to node F.
- **Step 6:** At node F, again the value of α will be compared with left child which is 0, and max(3,0)= 3, and then compared with right child which is 1, and max(3,1)= 3 still α remains 3, but the node value of F will become 1.



- **Step 7:** Node F returns the node value 1 to node C, at C α= 3 and β= +∞, here the value of beta will be changed, it will compare with 1 so min (∞, 1) = 1. Now at C, α=3 and β= 1, and again it satisfies the condition α>=β, so the next child of C which is G will be pruned, and the algorithm will not compute the entire sub-tree G.



- **Step 8:** C now returns the value of 1 to A here the best value for A is max (3, 1) = 3. Following is the final game tree which is the showing the nodes which are computed and nodes which has never computed. Hence the optimal value for the maximizer is 3 for this example.



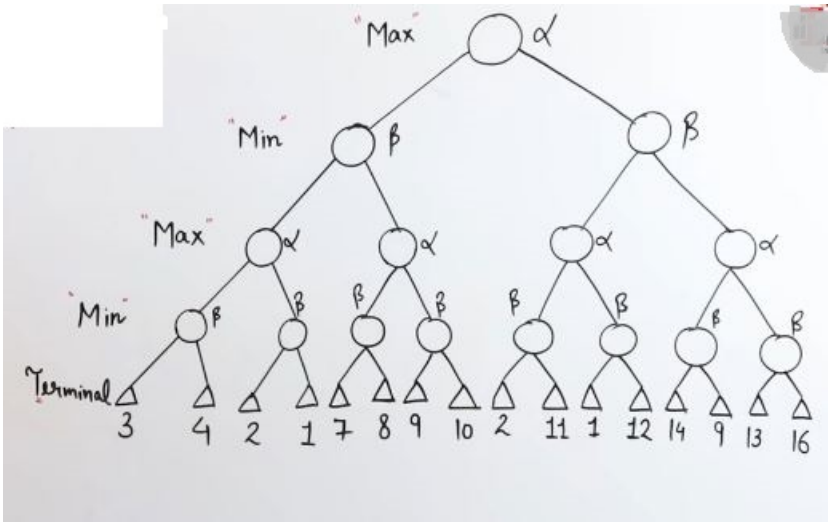
Ordering in Alpha-Beta Pruning

- The effectiveness of alpha-beta pruning is highly dependent on the order in which each node is examined. Move order is an important aspect of alpha-beta pruning.

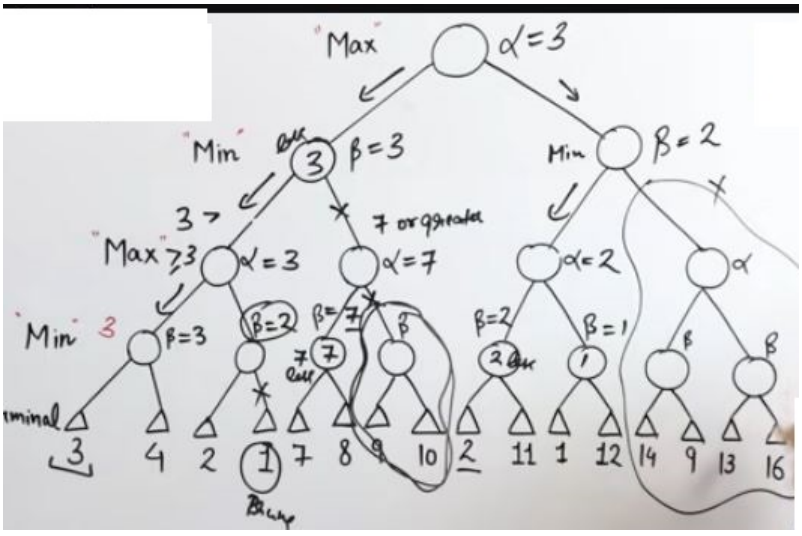
  It can be of two types:
- Worst ordering: In some cases, alpha-beta pruning algorithm does not prune any of the leaves of the tree, and works exactly as minimax algorithm. In this case, it also consumes more time because of alpha-beta factors, such a move of pruning is called worst ordering. In this case, the best move occurs on the right side of the tree. The time complexity for such an order is O(b^m).

- Ideal ordering: The ideal ordering for alpha-beta pruning occurs when lots of pruning happens in the tree, and best moves occur at the left side of the tree. We apply DFS hence it first search left of the tree and go deep twice as minimax algorithm in the same amount of time. Complexity in ideal ordering is O(b^m/2).

Problem(1) of Alpha –beta pruning



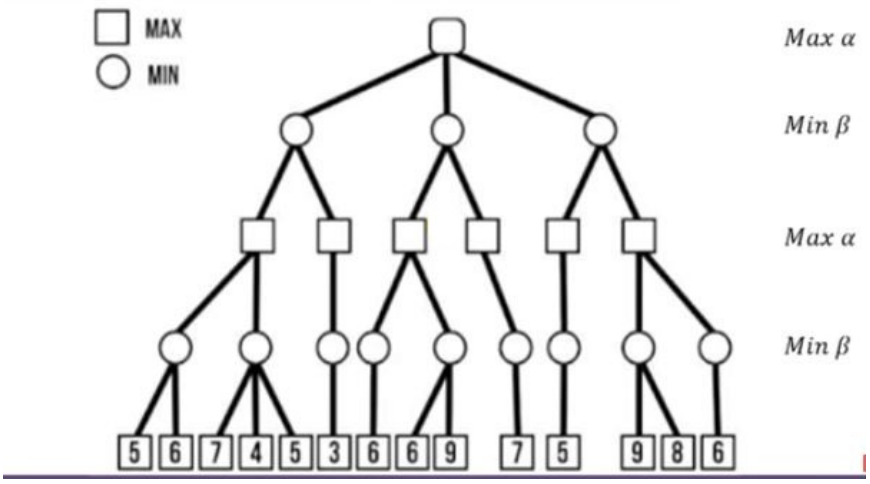
Solution



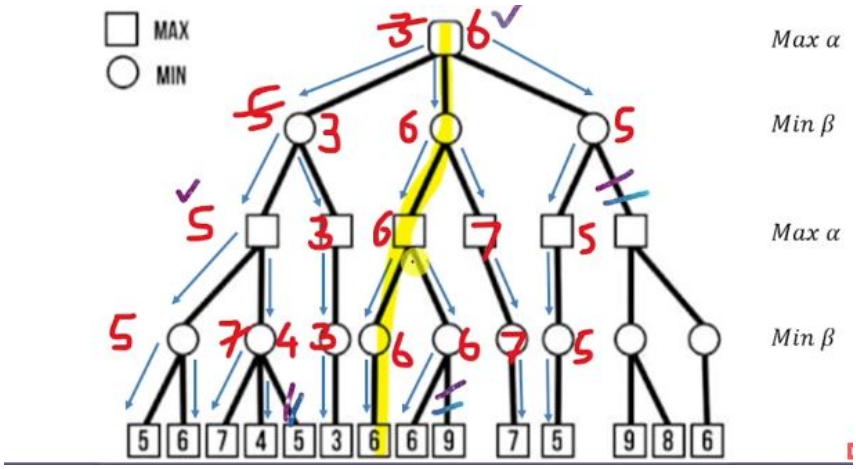
Notes

- $\boldsymbol{Alpha(\alpha) - Beta(\beta)}$ proposes to compute find the optimal path without looking at every node in the game tree.
- **Max contains** $\boldsymbol{Alpha(\alpha)}$ and **Min contains** $\boldsymbol{Beta(\beta)}$ bound during the calculation.
- In both **MIN** and **MAX** node, we return when $\boldsymbol{\alpha \geq \beta}$ which compares with its parent node only.
- Both **minimax and** $\boldsymbol{Alpha(\alpha) - Beta(\beta)}$ cut-off give same path
- $\boldsymbol{Alpha(\alpha) - Beta(\beta)}$ gives **optimal solution** as it takes less time to get the value for the root node.

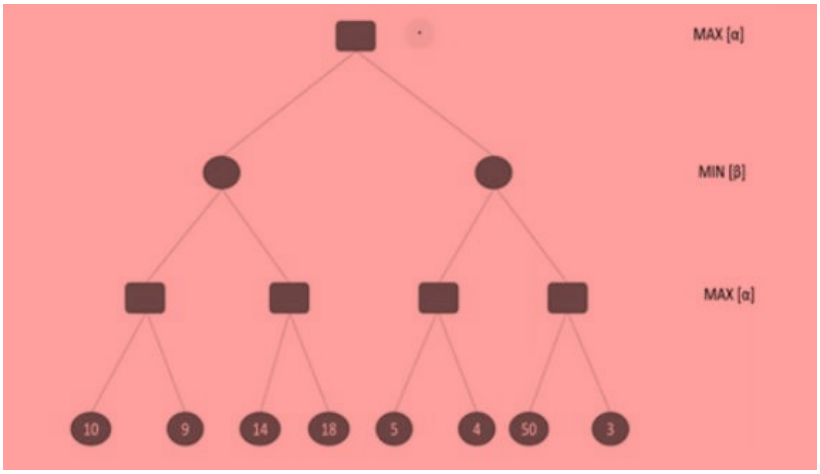
Problem(2) of Alpha –beta pruning



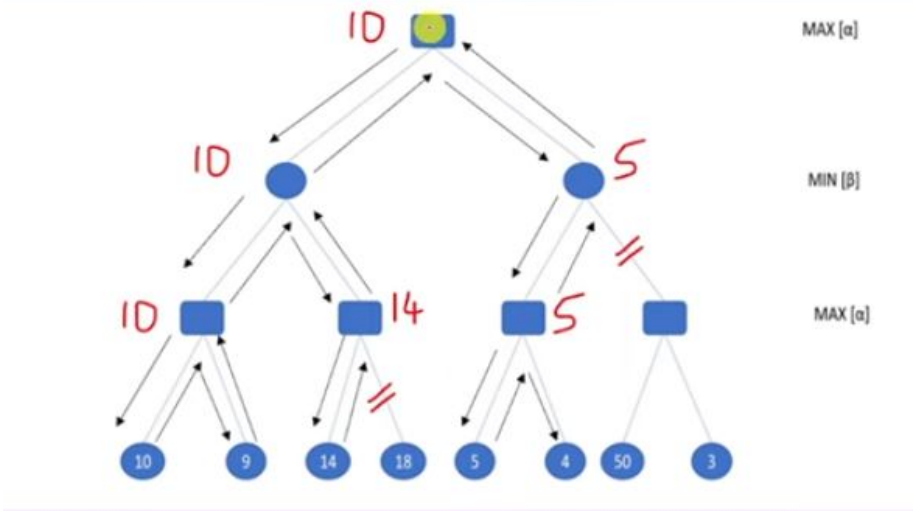
Solution



Problem(3) of Alpha –beta pruning



Solution



Hill Climbing Search Algorithm in Artificial Intelligence

1. Evaluate the initial state. If it is also goal state then return it, otherwise continue with the initial state as the current state.
2. Loop until the solution is found or until there are no new operators to be applied in the current state
   a) Select an operator that has not yet been applied to the current state and apply it to produce new state
   b) Evaluate the new state
      i. If it is a goal state, then return it and quit
      ii. If it is not a goal state but it is better than the current state, then make it as current state
      iii. If it is not better than the current state, then continue in loop.

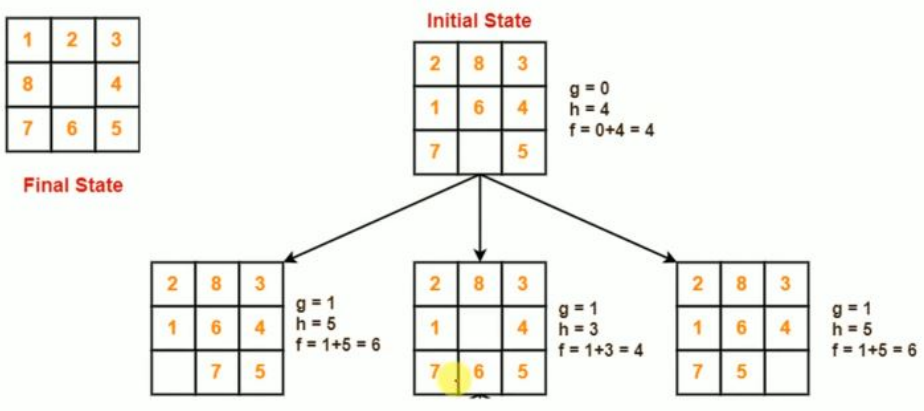
Hill Climbing Search Algorithm in Artificial Intelligence

- In hill climbing the basic idea is to always head towards a state which is better than the current one.
- So, if you are at town A and you can get to town B and town C (and your target is town D) then you should make a move IF town B or C appear nearer to town D than town A does.

HILL CLIMBING ALGORITHM

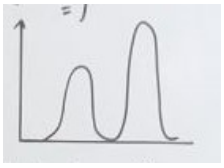
- Evaluate the initial state.
- Loop until a solution is found or there are no operators are left.
- Select and apply a new operator
- Evaluate the new state: if goal the quit.
  - If better than current state then it is new current state.

Example

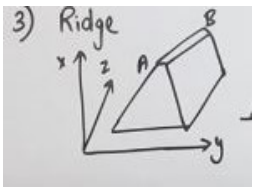


Problems in Hill Climbing

- Local Maximum.



- Plateau/ Flat Maximum.
- Ridge.



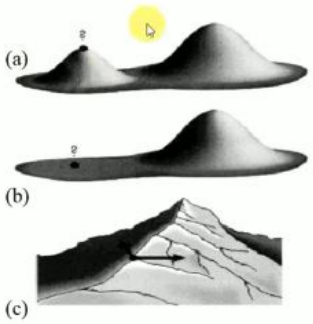
Hill Climbing Search Algorithm - Drawbacks

This simple Hill Climbing policy has three well-known drawbacks:

1. **Local Maxima**: a local maximum as opposed to global maximum.

2. **Plateaus**: An area of the search space where evaluation function is flat, thus requiring random walk.

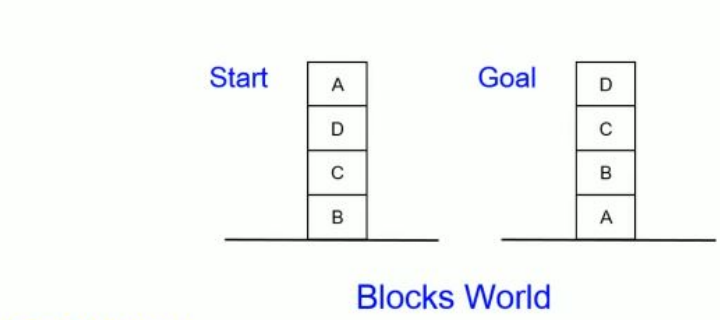
3. **Ridge**: Where there are steep slopes, and the search direction is not towards the top but towards the side.



Hill Climbing: Disadvantages

- Hill climbing using local information :
  Decides what to do next by looking only at the "immediate" consequences of its choices.
- Will terminate when at local optimum.
- The order of application of operators can make a big difference.
- Global information might be encoded in heuristic functions.

Example

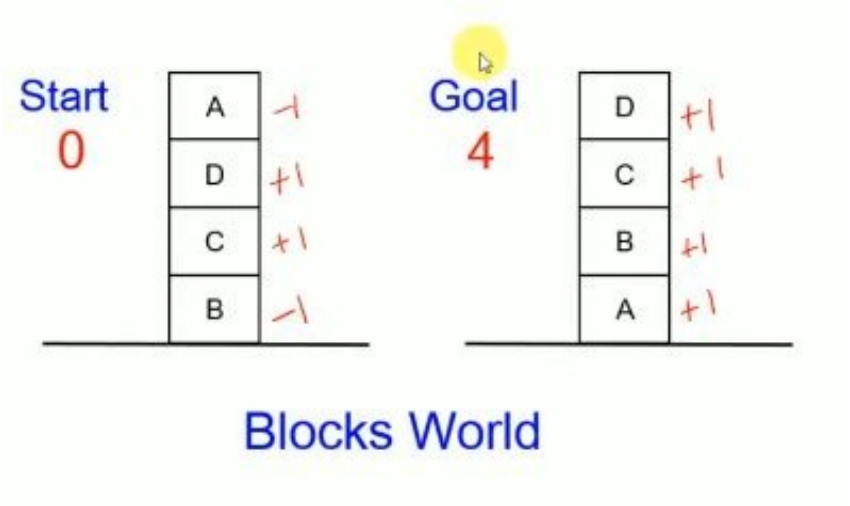


Blocks World

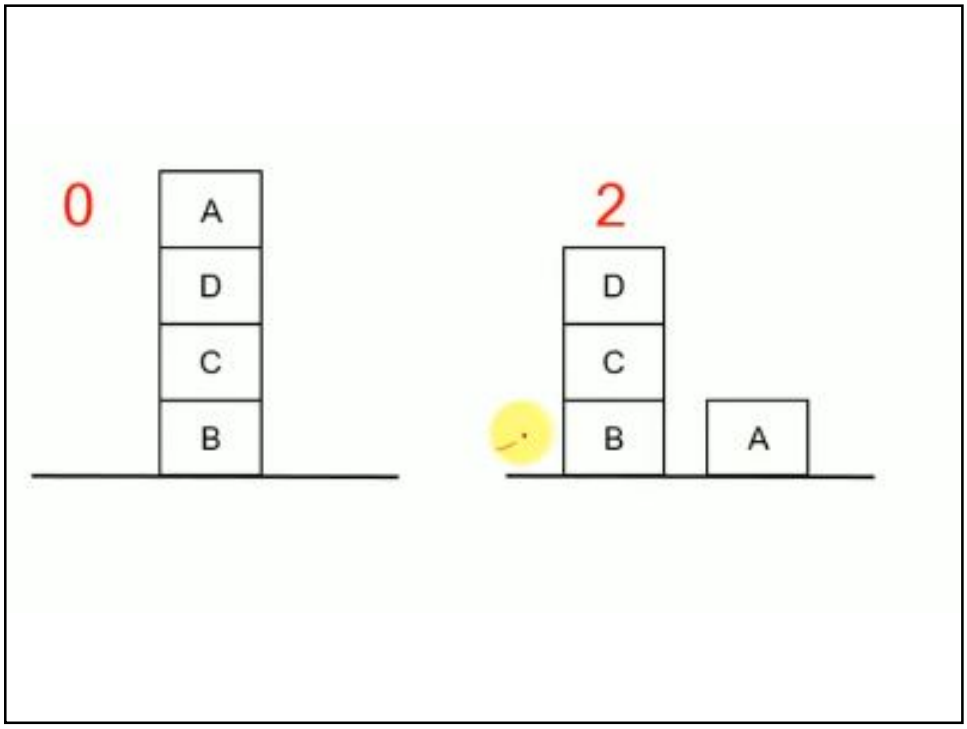
Local heuristic:

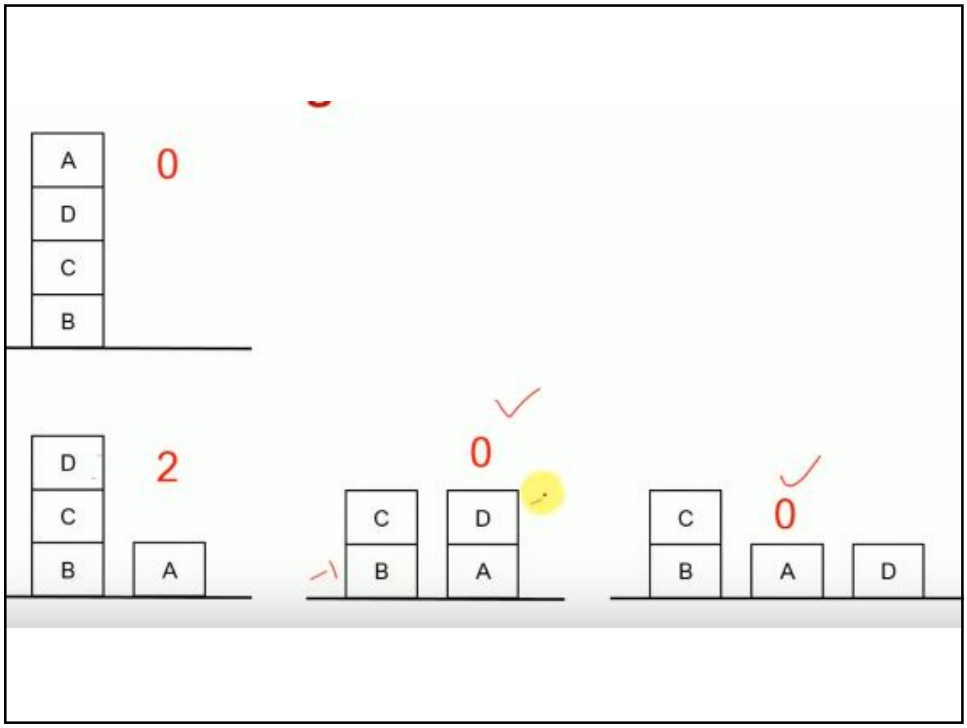
+1 for each block that is resting on the thing it is supposed to be resting on.

–1 for each block that is resting on a wrong thing.

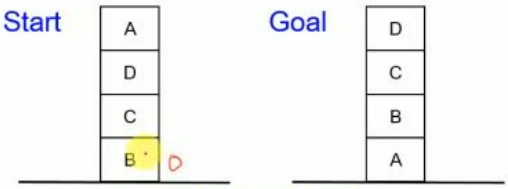


Blocks World


0
A
D
C
B
2
D
C
B
A


A
D
C
B
0
D
C
B
A
2
C
B
D
A
0
C
B
A
D
0

Hill Climbing: Global Heuristic function

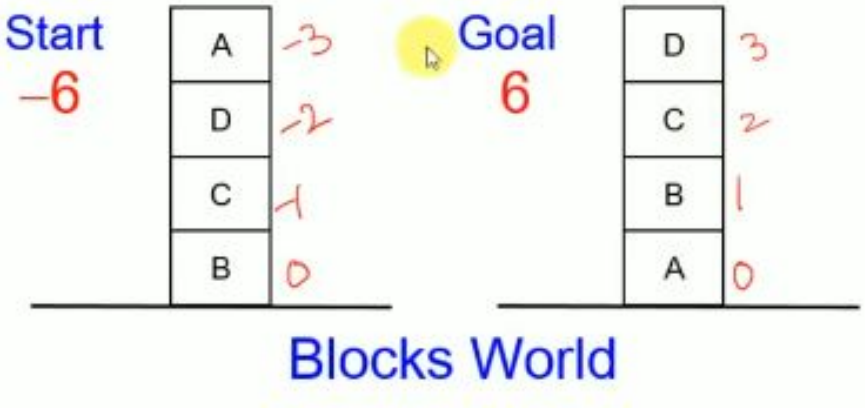


Blocks World

Global heuristic:

For each block that has the correct support structure: +1 to every block in the support structure.

For each block that has a wrong support structure: –1 to every block in the support structure.



Blocks World

Hill Climbing: Global Heuristic function