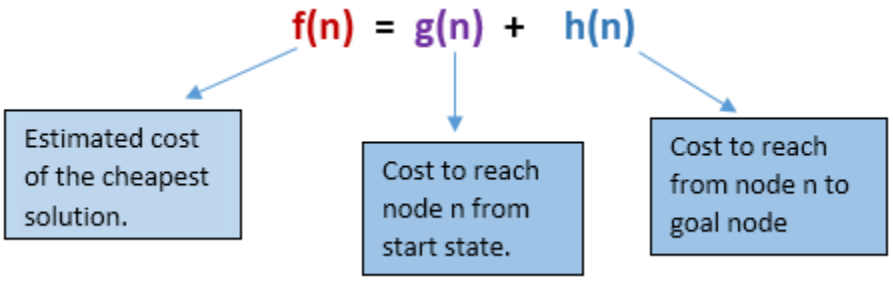
A* Search Algorithm

- **A* is a cornerstone name of many AI systems and has been used since it was developed in 1968 by Peter Hart; Nils Nilsson and Bertram Raphael.**
- **A* search finds the shortest path through a search space to the goal state using the heuristic function. This technique finds minimal cost solutions and is directed to a goal state called A* search. In A*, the * is written for optimality purposes.**
- **The A* algorithm also finds the lowest-cost path between the start and goal state, where changing from one state to another requires some cost.**
- **A* algorithm is similar to Best First Search.**
- **Only difference: Best First Search takes h(n) as evaluation function/heuristic value.**
- **In Best first search h(n)= estimated cost of current node 'n' from the goal node.**
- **A* takes the evaluation function as:**



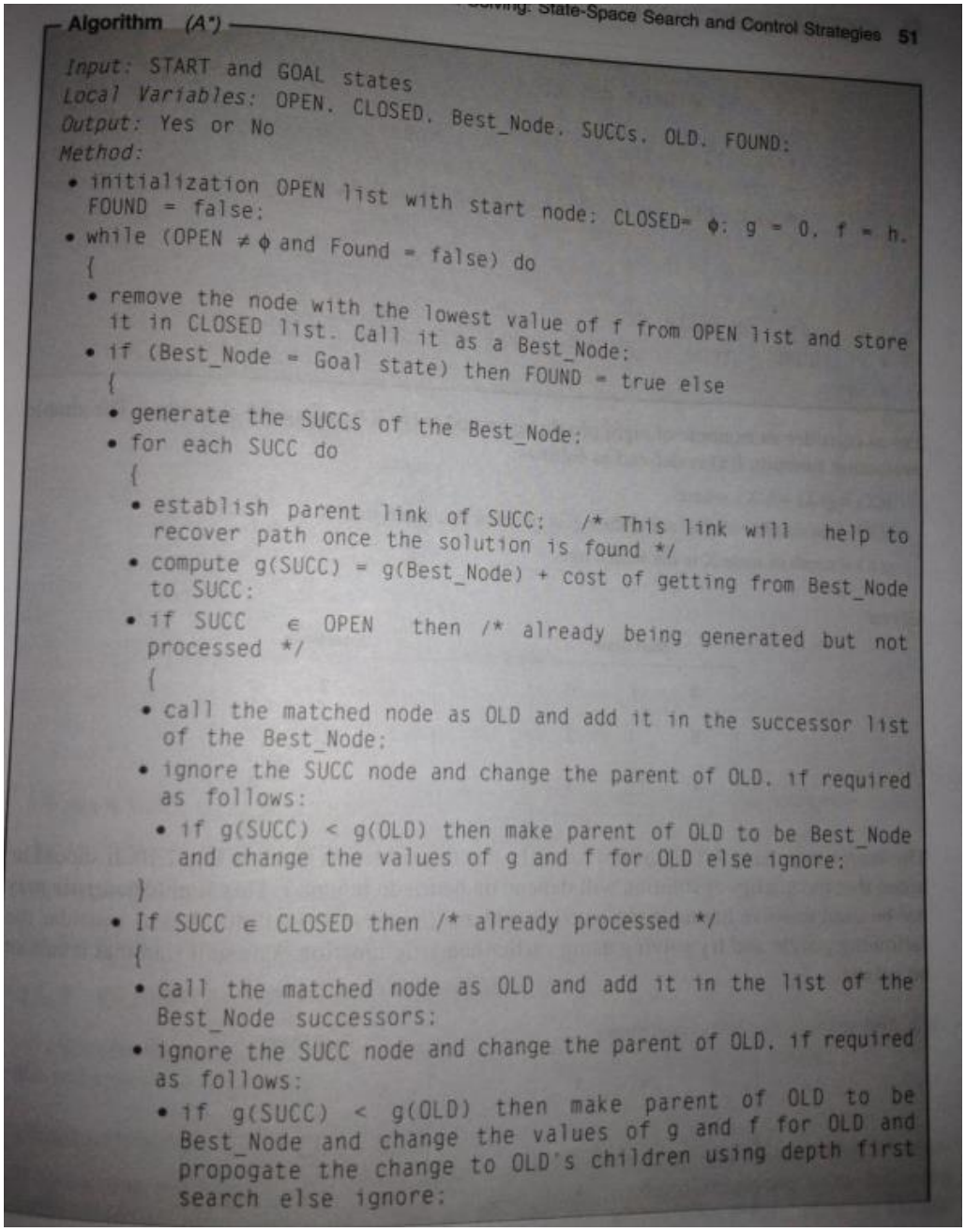
Where

- ***g(n): The actual cost path from the start state to the current state.***
- ***h(n): The actual cost path from the current state to the goal state.***
- ***f(n): The actual cost path from the start state to the goal state***

For the implementation of the A* algorithm, we will use two arrays namely OPEN and CLOSE.

- **OPEN: An array that contains the nodes that have been generated but have not been yet examined.**
- **CLOSE: An array that contains the nodes that have been examined.**

Algorithm:

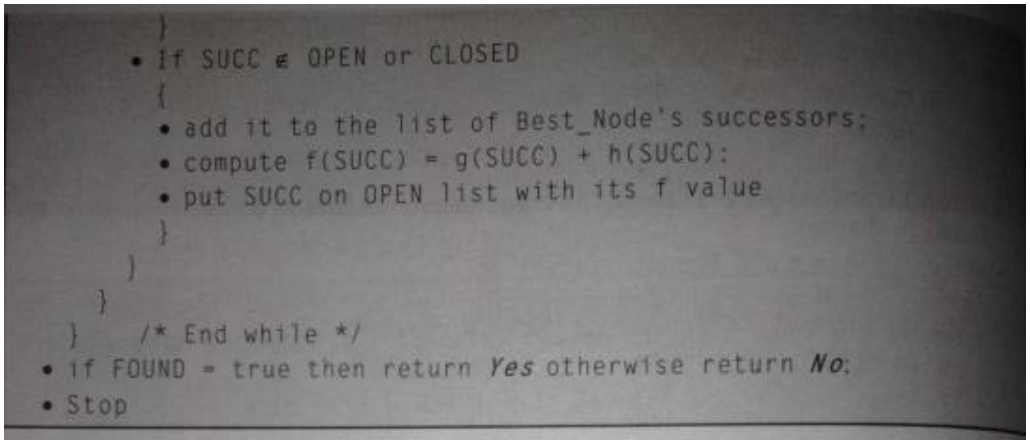


Algorithm *(A*)*

Input: START and GOAL states
Local Variables: OPEN, CLOSED, Best_Node, SUCCs, OLD, FOUND;
Output: Yes or No
Method:

- initialization OPEN list with start node; CLOSED= ϕ; g = 0, f = h, FOUND = false;
- while (OPEN ≠ ϕ and Found = false) do
  {
  - remove the node with the lowest value of f from OPEN list and store it in CLOSED list. Call it as a Best_Node;
  - if (Best_Node = Goal state) then FOUND = true else
    {
    - generate the SUCCs of the Best_Node;
    - for each SUCC do
      {
      - establish parent link of SUCC; /* This link will help to recover path once the solution is found */
      - compute g(SUCC) = g(Best_Node) + cost of getting from Best_Node to SUCC;
      - if SUCC ∈ OPEN then /* already being generated but not processed */
        {
        - call the matched node as OLD and add it in the successor list of the Best_Node;
        - ignore the SUCC node and change the parent of OLD, if required as follows:
          - if g(SUCC) < g(OLD) then make parent of OLD to be Best_Node and change the values of g and f for OLD else ignore;

        }
      - If SUCC ∈ CLOSED then /* already processed */
        {
        - call the matched node as OLD and add it in the list of the Best_Node successors;
        - ignore the SUCC node and change the parent of OLD, if required as follows:
          - if g(SUCC) < g(OLD) then make parent of OLD to be Best_Node and change the values of g and f for OLD and propogate the change to OLD's children using depth first search else ignore;



```
        }
      • If SUCC ∉ OPEN or CLOSED
        {
        • add it to the list of Best_Node's successors;
        • compute f(SUCC) = g(SUCC) + h(SUCC);
        • put SUCC on OPEN list with its f value
        }
      }
    }
  }    /* End while */
• if FOUND = true then return Yes otherwise return No;
• Stop
```

Analysis of the algorithm:

The algorithm is as follows-

Step-01:

- Define a list OPEN.
- Initially, OPEN consists exclusively of a single node, the start node S.

Step-02:

- If the list is empty, return failure and exit.

Step-03:

Remove node n with the smallest value of f(n) from OPEN and move it to list CLOSED.

- If node n is a goal state, return success and exit.

Step-04:

- Expand node n.

Step-05:

- If any successor to n is the goal node, return success and the solution by tracing the path from goal node to S.
- Otherwise, go to Step-06.

Step-06:

For each successor node,

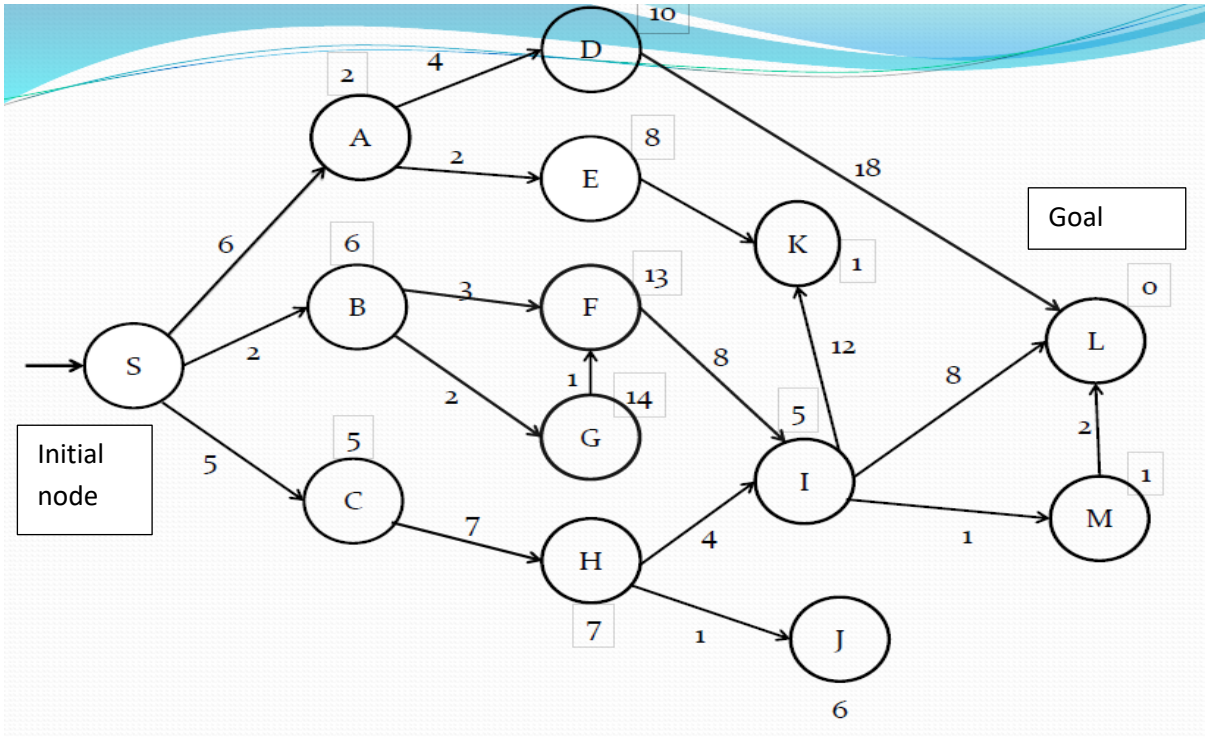
- Apply the evaluation function f to the node.
- If the node has not been in either list, add it to OPEN.

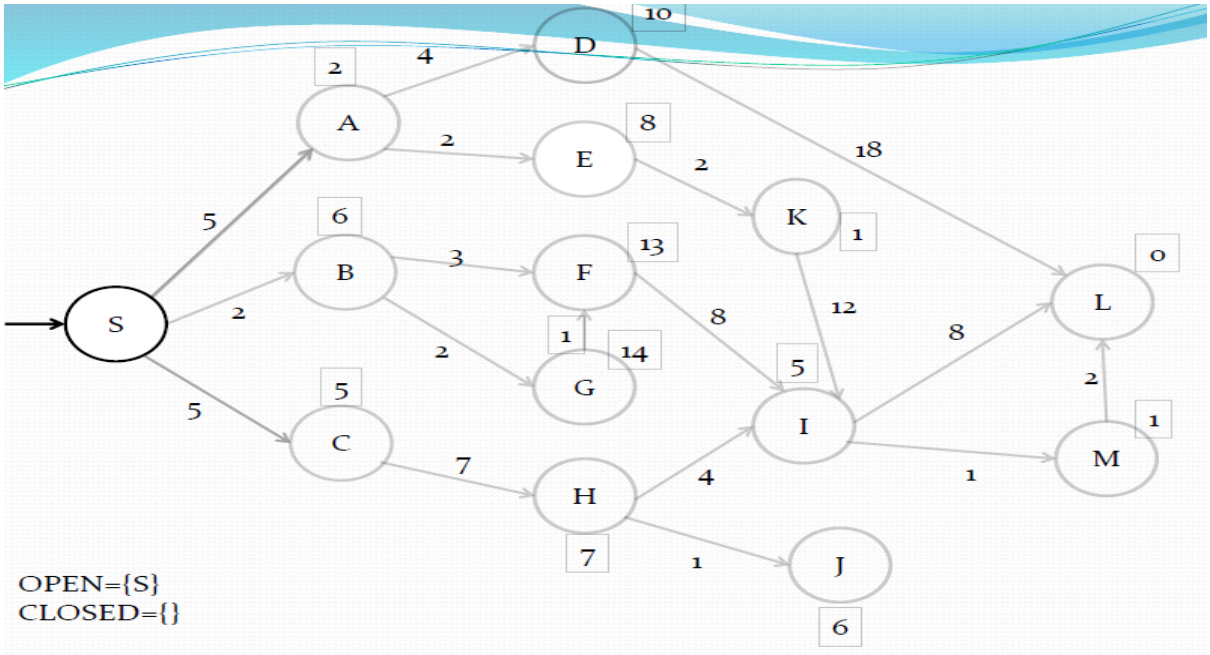
Step-07:

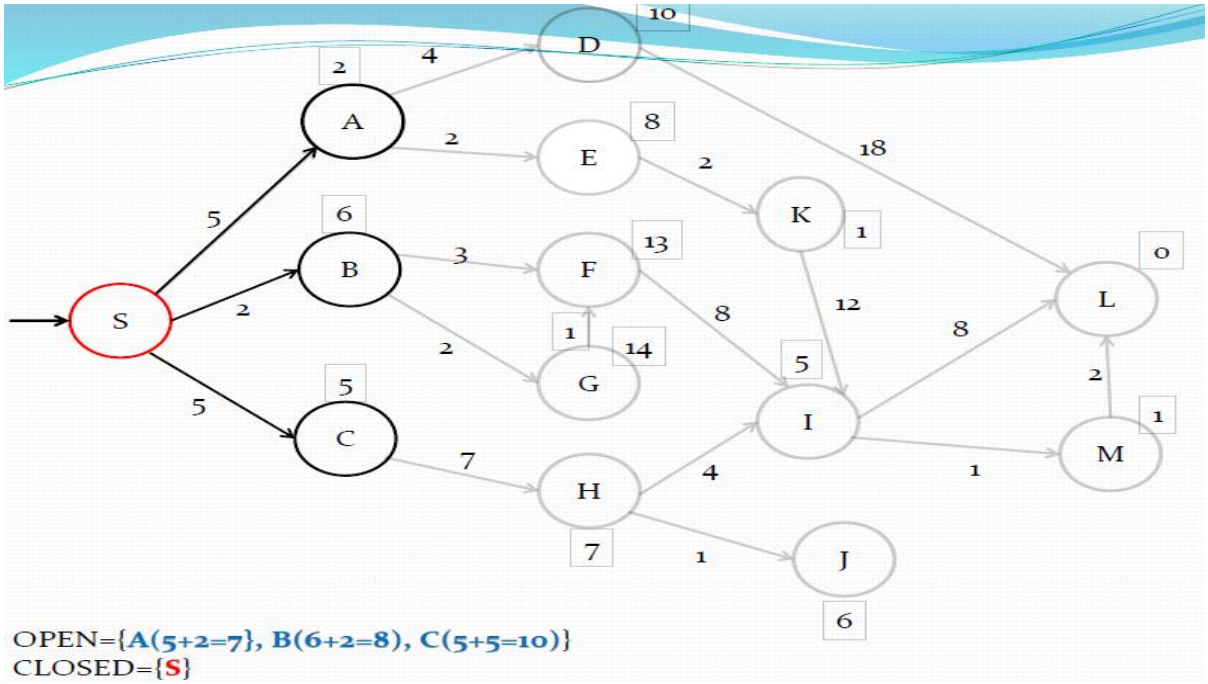
- Go back to Step-02.

PRACTICE PROBLEMS BASED ON A* ALGORITHM-

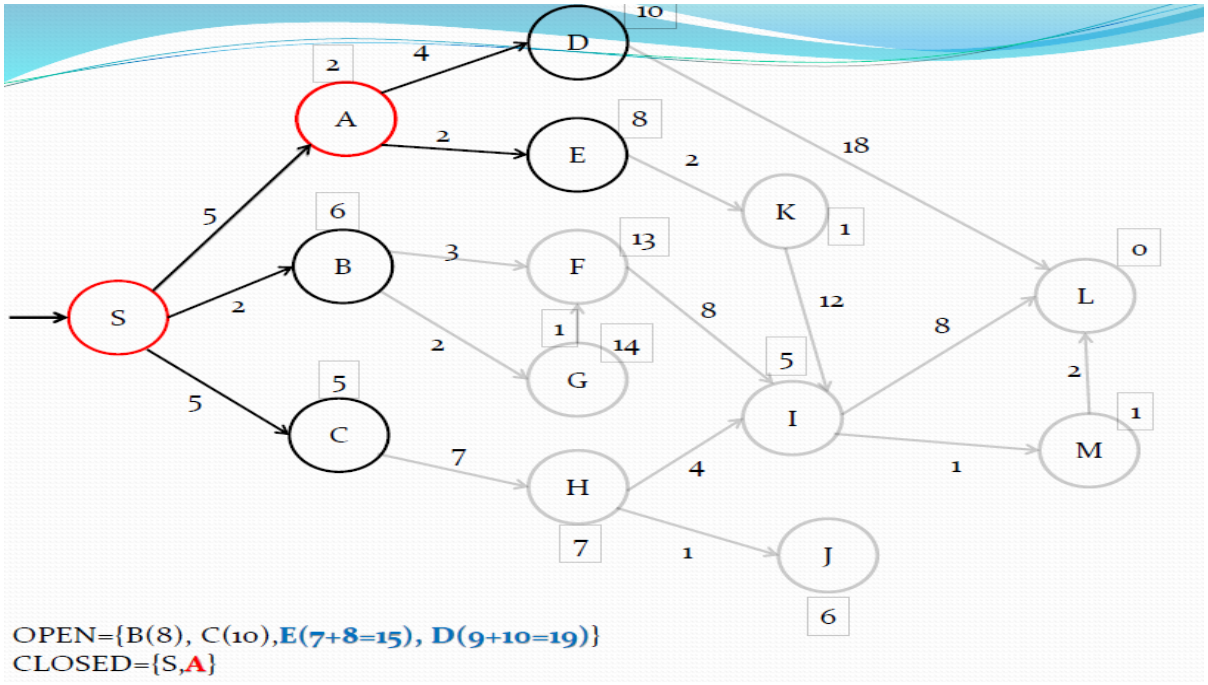
Example:1



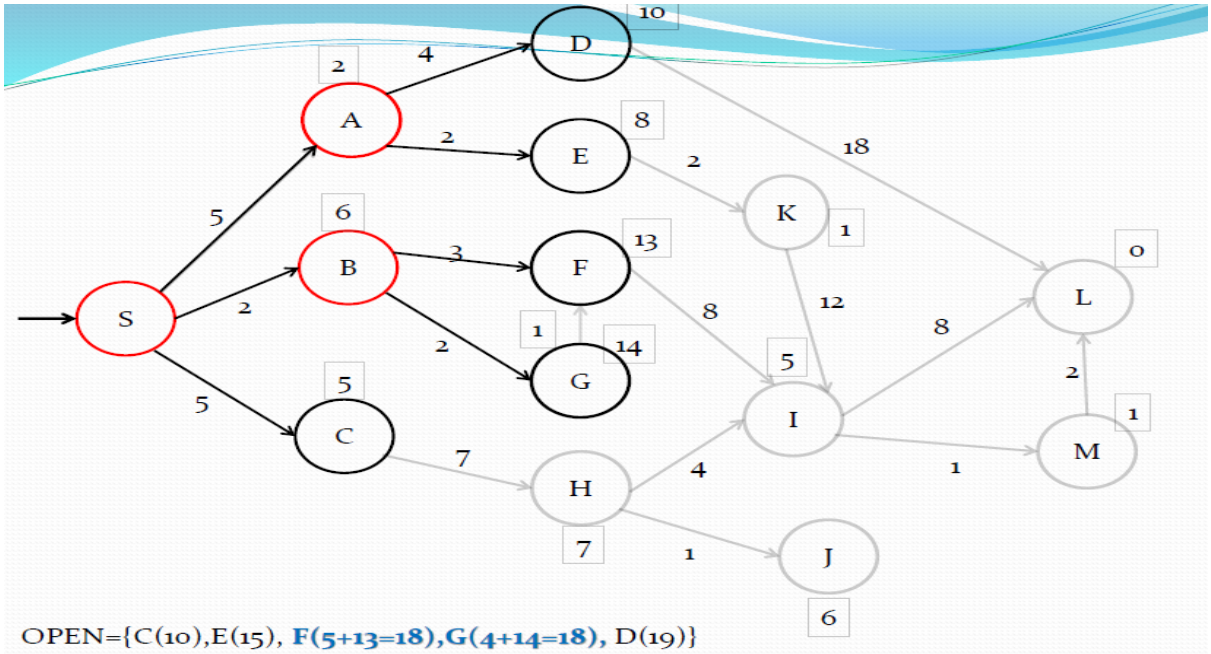




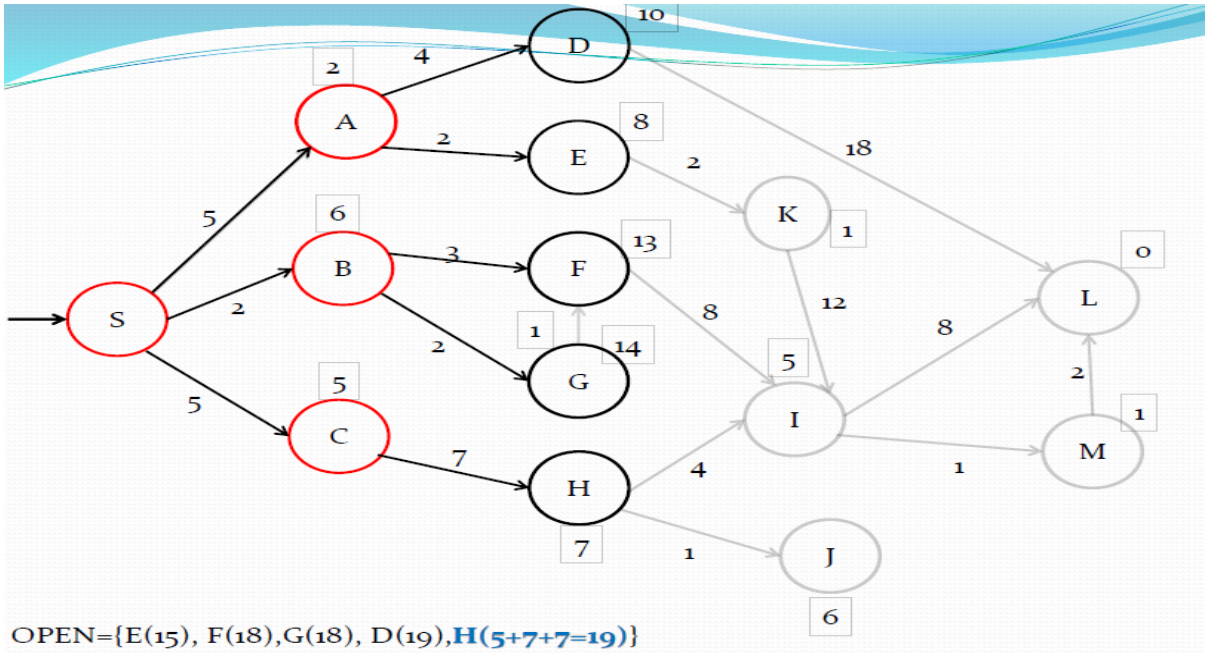
OPEN={**A(5+2=7}, B(6+2=8), C(5+5=10)**}
CLOSED={**S**}



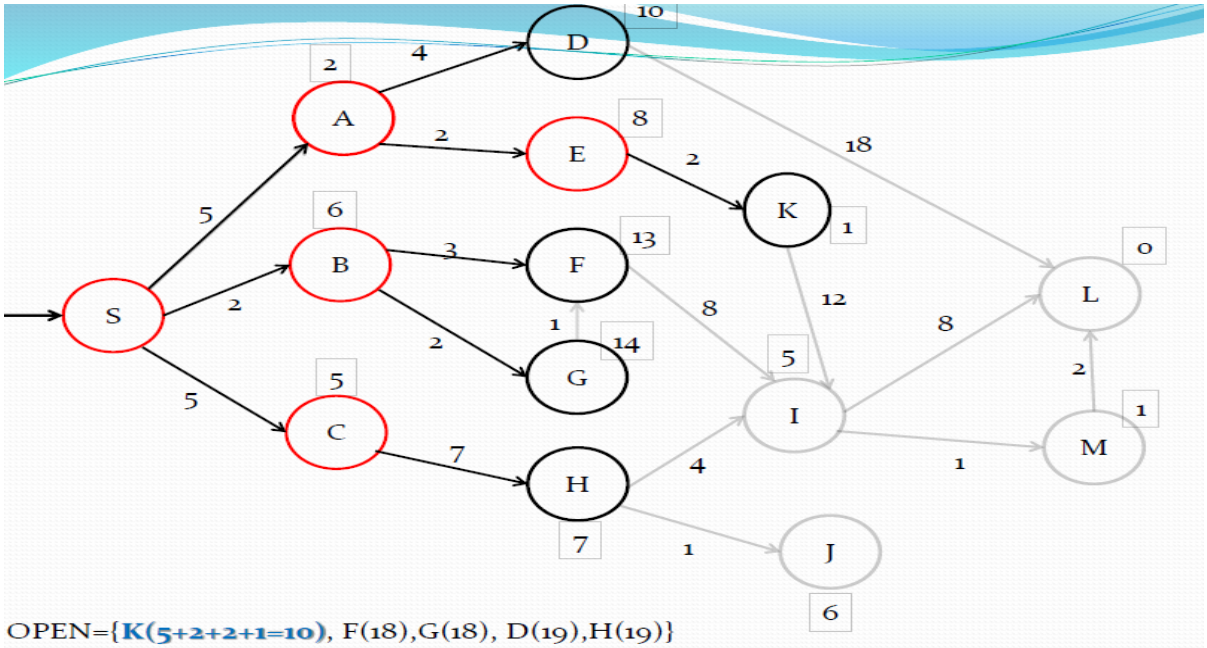
OPEN={B(8), C(10),**E(7+8=15), D(9+10=19)**}
CLOSED={S,**A**}



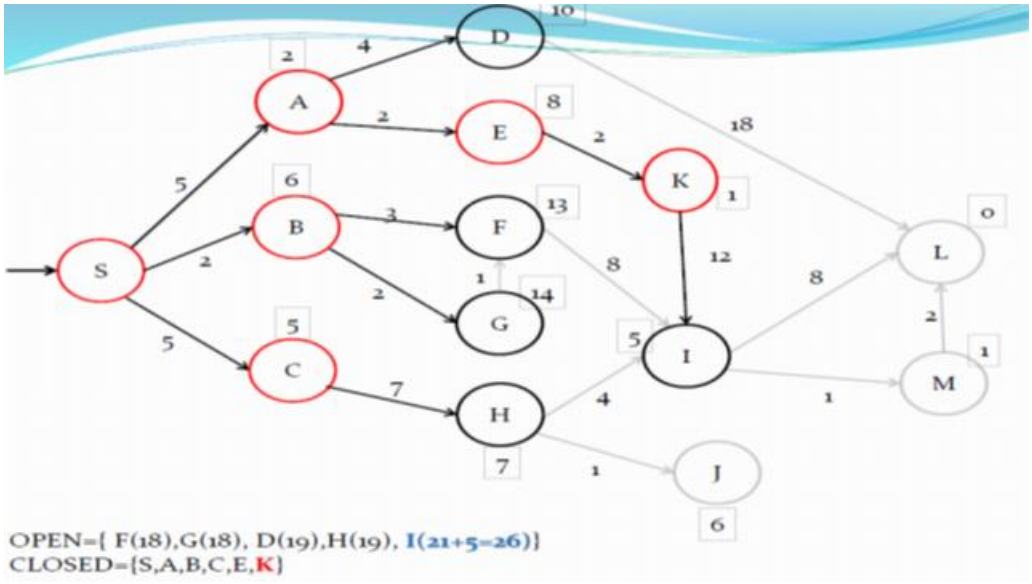
OPEN={C(10),E(15), **F(5+13=18),G(4+14=18),** D(19)}
CLOSED={S,A,**B**}



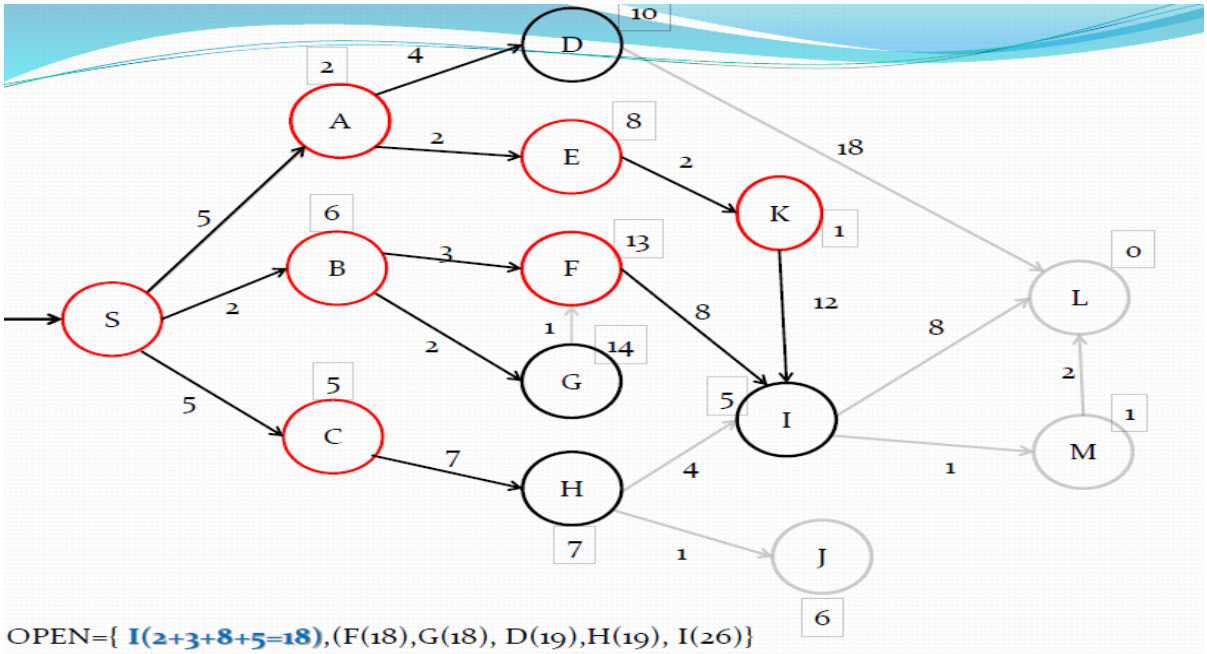
OPEN={E(15), F(18),G(18), D(19),**H(5+7+7=19)**}
CLOSED={S,A,B,**C**}



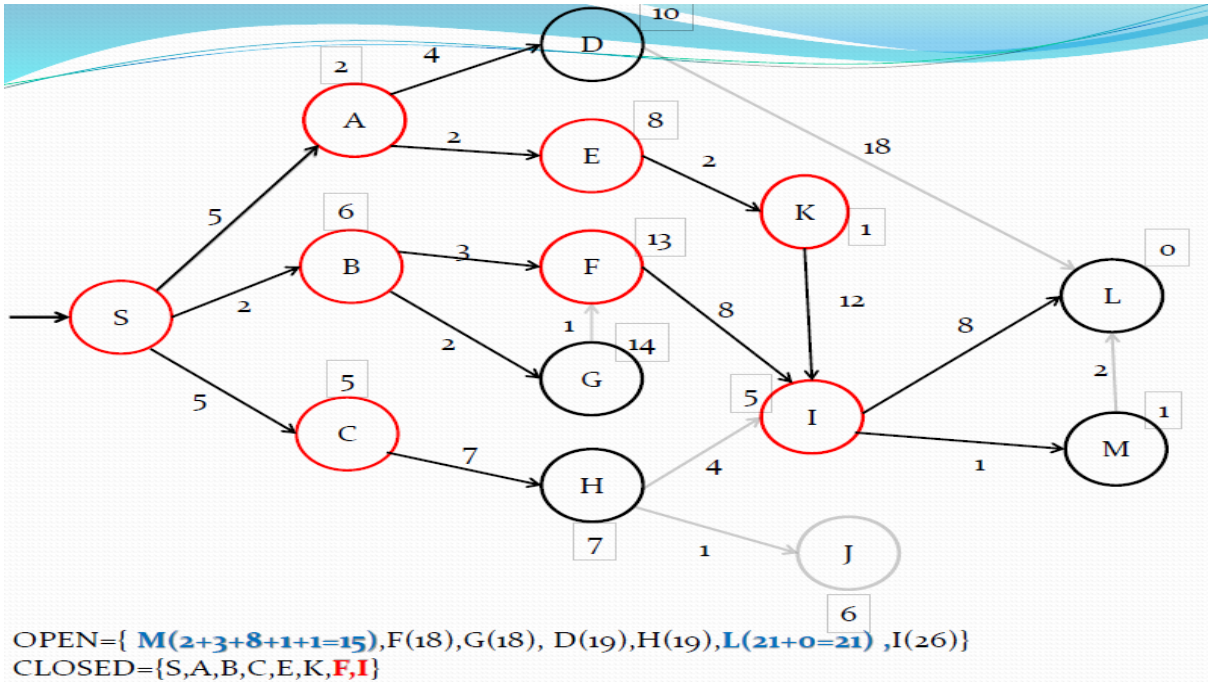
OPEN={**K(5+2+2+1=10)**, F(18),G(18), D(19),H(19)}
CLOSED={S,A,B,C,**E**}



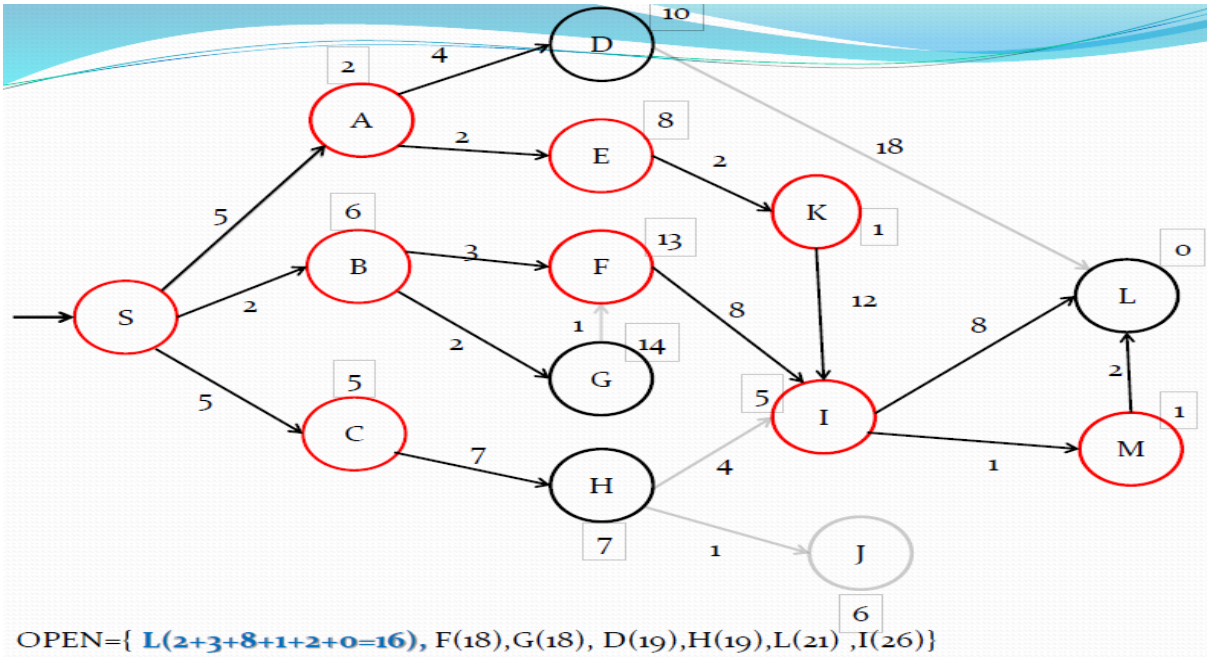
OPEN={ F(18),G(18), D(19),H(19), **I(21+5=26)**}
CLOSED={S,A,B,C,E,**K**}



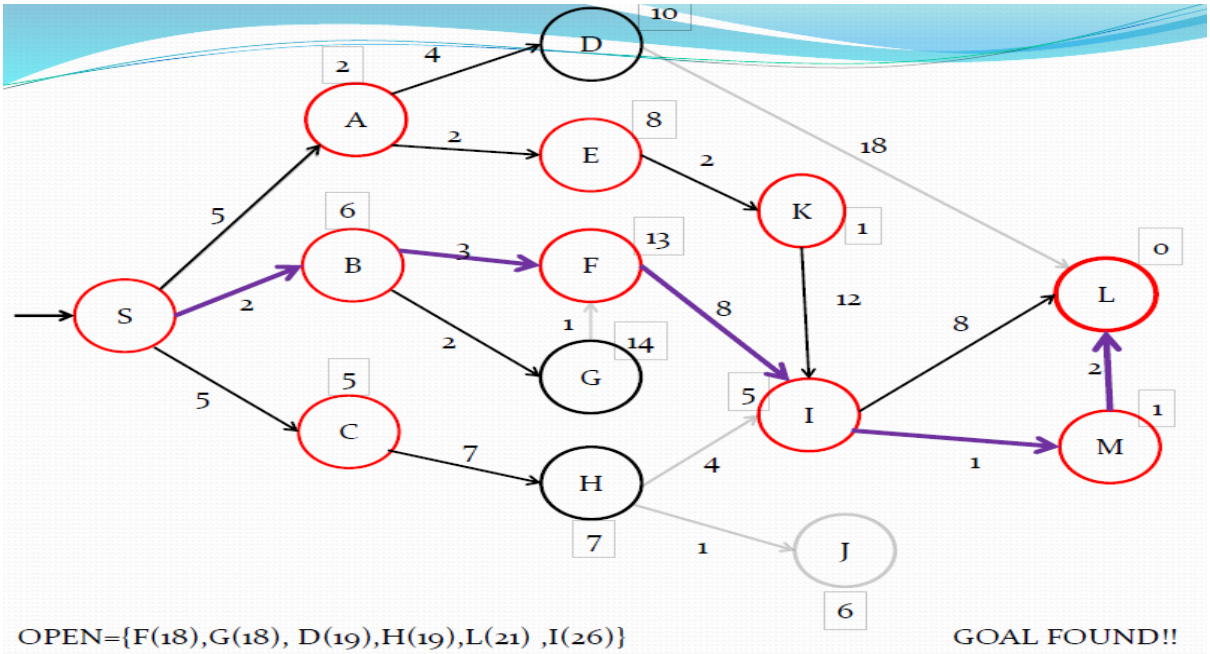
OPEN={ **I(2+3+8+5=18)**,(F(18),G(18), D(19),H(19), I(26)}
CLOSED={S,A,B,C,E,K,**F**}



OPEN={ **M(2+3+8+1+1=15)**,F(18),G(18), D(19),H(19),**L(21+0=21)** ,I(26)}
CLOSED={S,A,B,C,E,K,**F,I**}



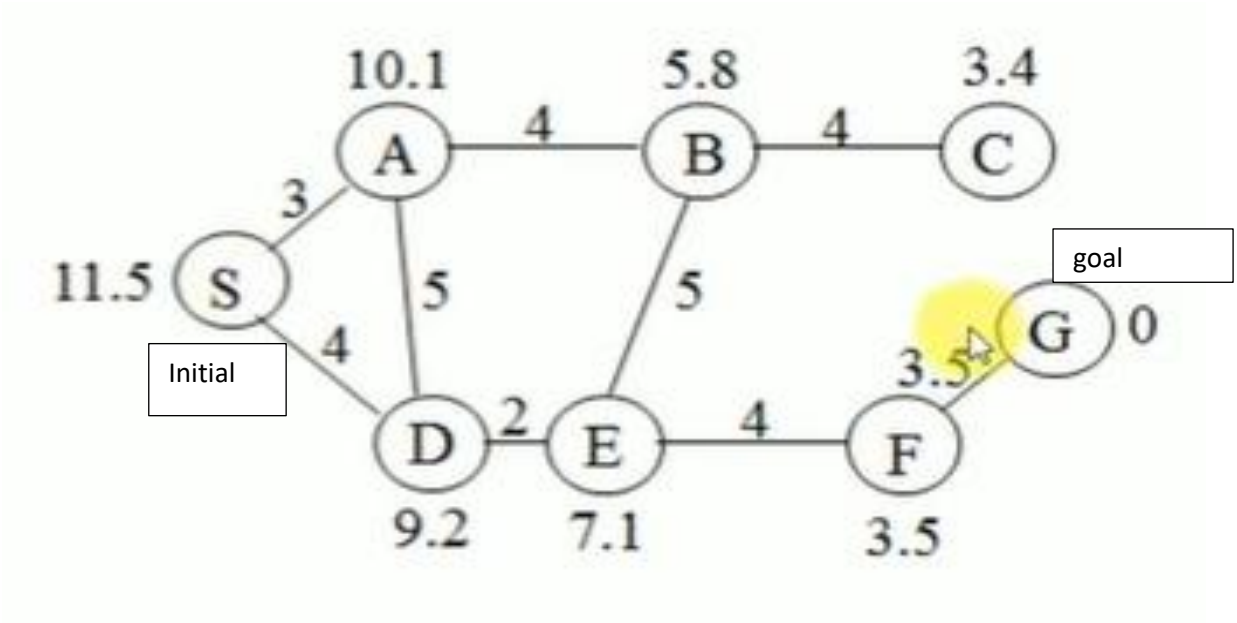
OPEN={ **L(2+3+8+1+2+0=16),** F(18),G(18), D(19),H(19),L(21) ,I(26)}
CLOSED={S,A,B,C,E,K,**F,I,M**}

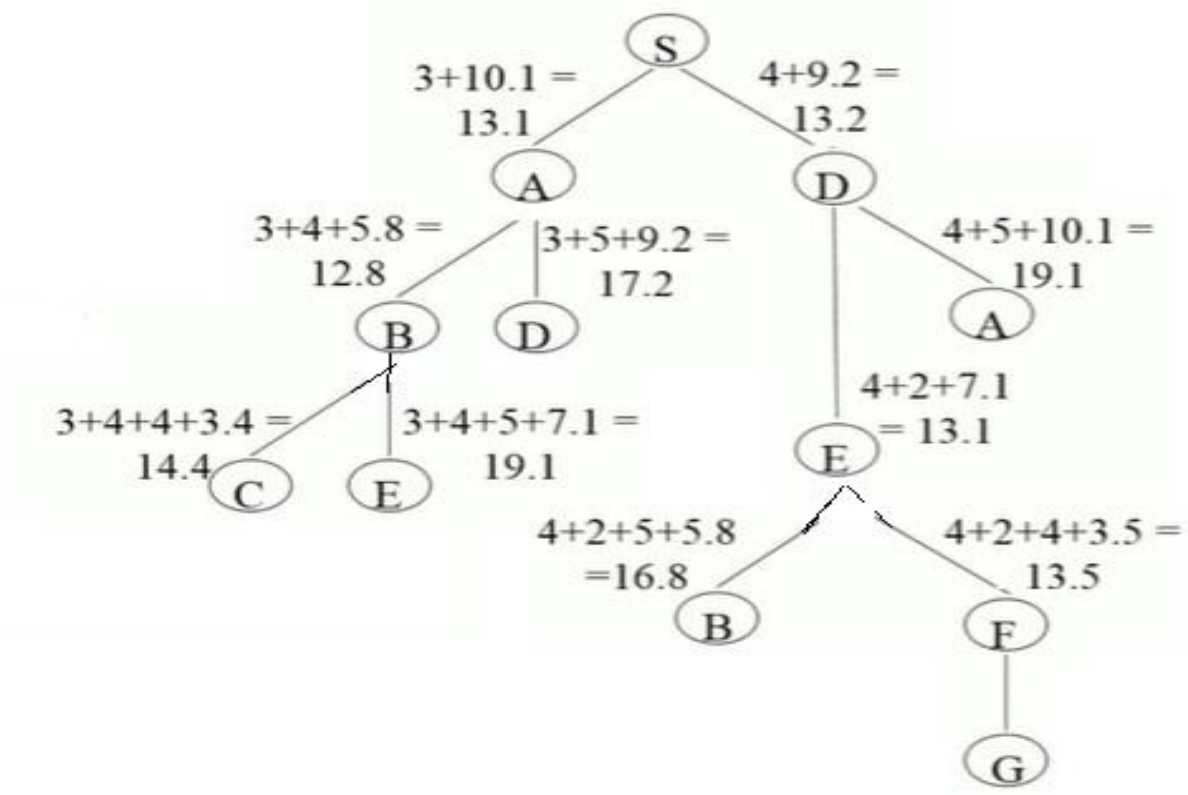


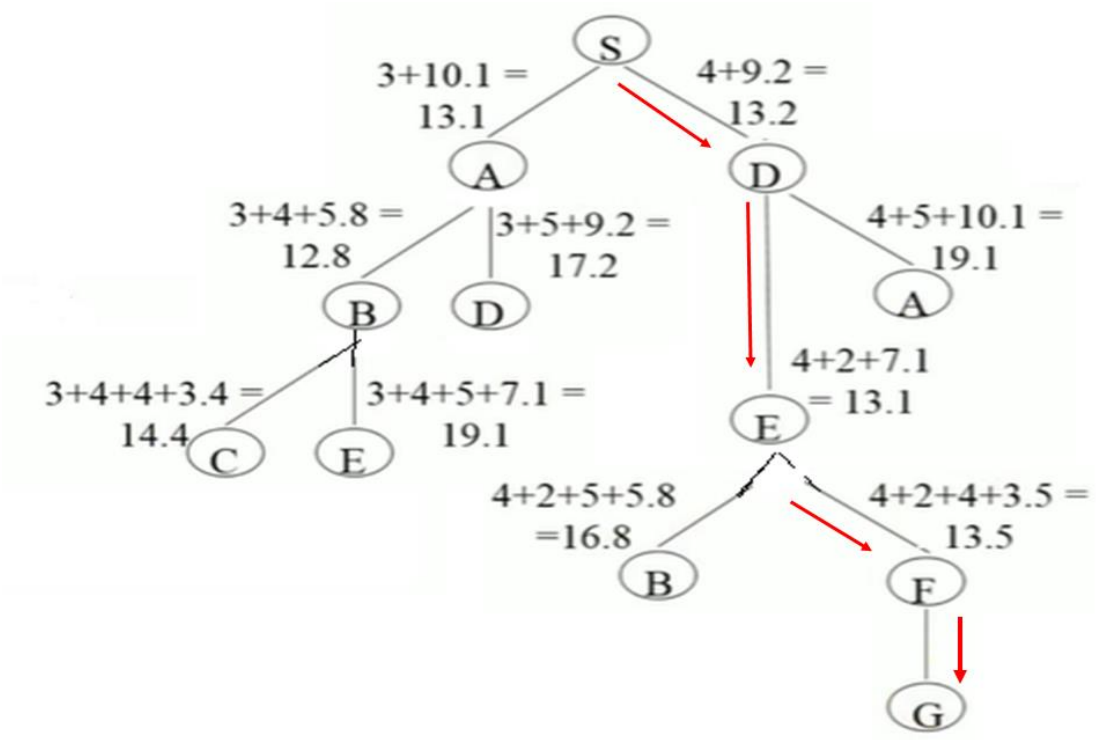
OPEN={F(18),G(18), D(19),H(19),L(21) ,I(26)}
CLOSED={S,A,B,C,E,K,**F,I,M,L**}

GOAL FOUND!!

Example:2

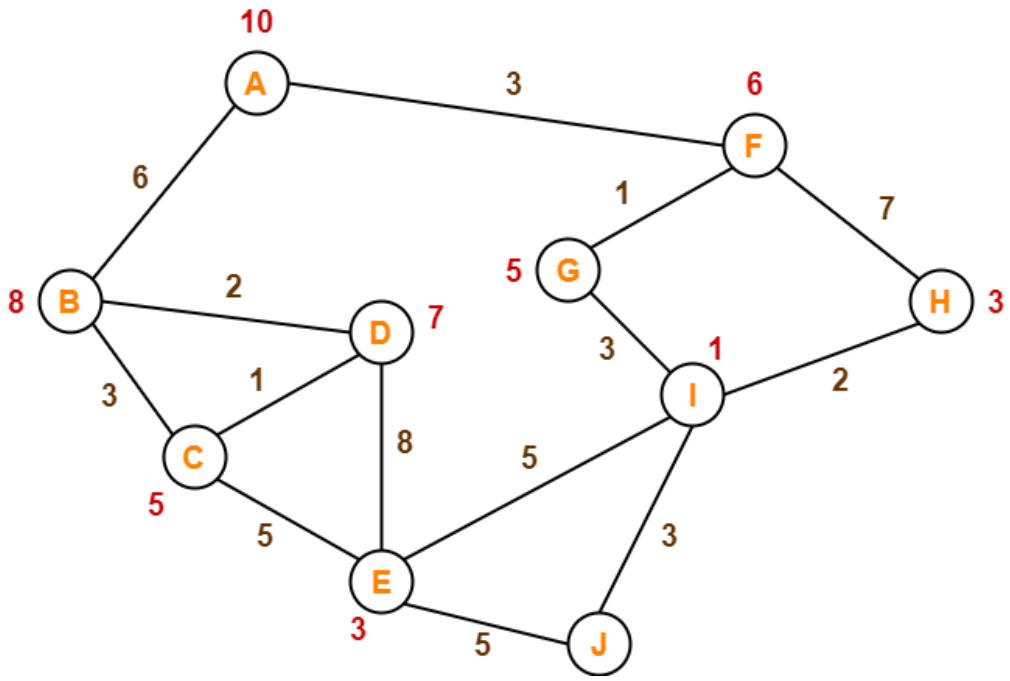

10.1
5.8
3.4
A
4
B
4
C
3
11.5
S
5
5
goal
G
0
4
Initial
3.5
D
2
E
4
F
9.2
7.1
3.5


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



Examples:3

Consider the following graph-



The numbers written on edges represent the distance between the nodes.

The numbers written on nodes represent the heuristic value.

Find the most cost-effective path to reach from start state A to final state J using A* Algorithm.

Solution-

Step-01:

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

A* Algorithm calculates f(B) and f(F).

- f(B) = 6 + 8 = 14
- f(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(G) and f(H).

- f(G) = (3+1) + 5 = 9
- f(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(I).

f(I) = (3+1+3) + 1 = 8

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.

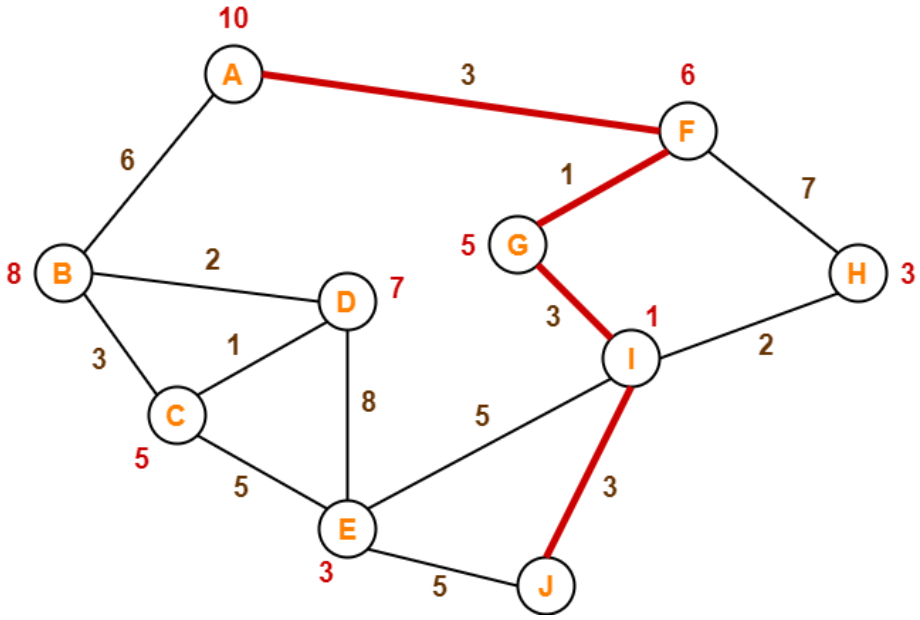
A* Algorithm calculates f(E), f(H) and f(J).

- 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

This is the required shortest path from node A to node J.



Example:4

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

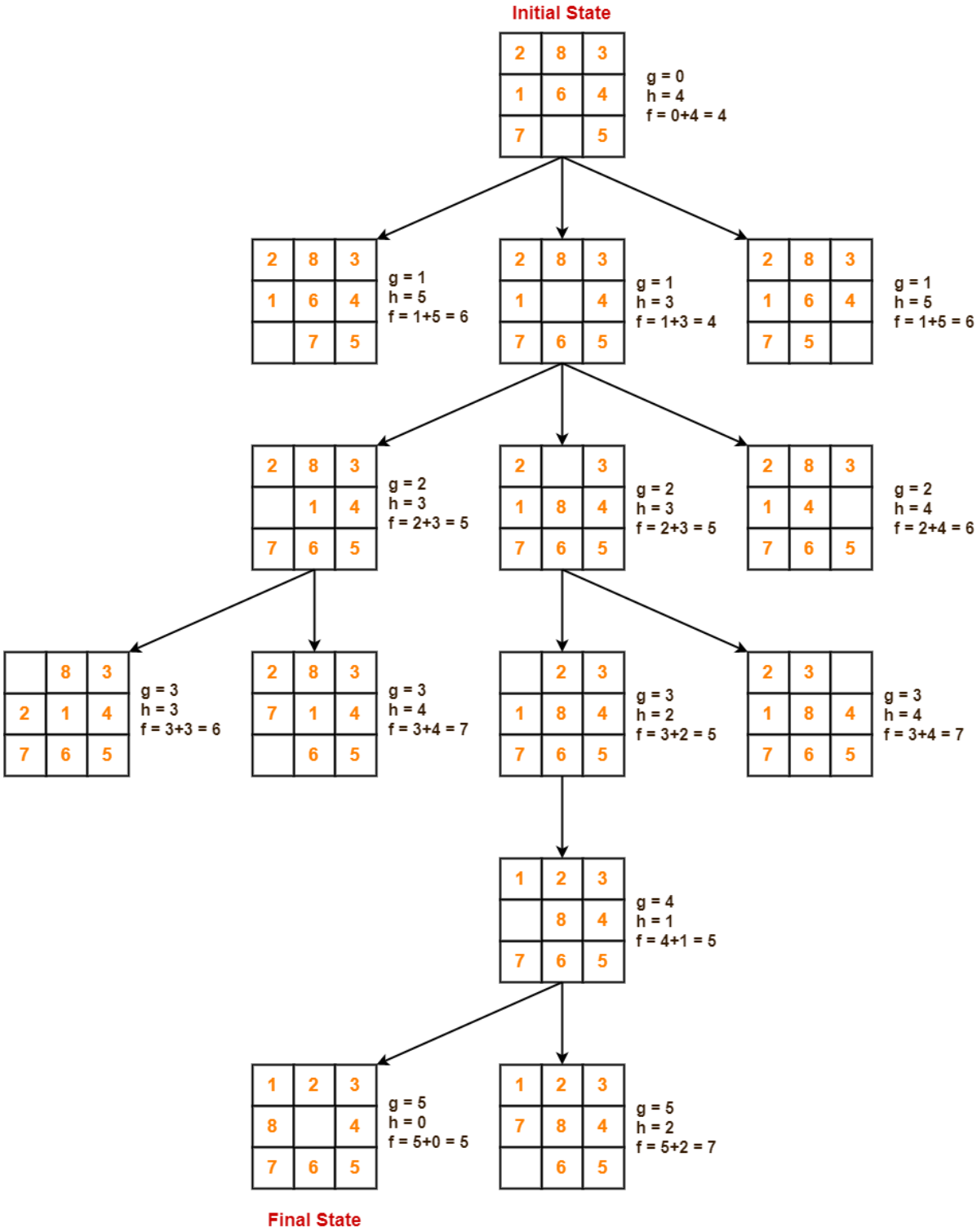
Final State

Find the most cost-effective path to reach the final state from initial state using A* Algorithm.

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

Solution-

- A* Algorithm maintains a tree of paths originating at the initial state.
- It extends those paths one edge at a time.
- It continues until final state is reached.



Important Note-

It is important to note that-

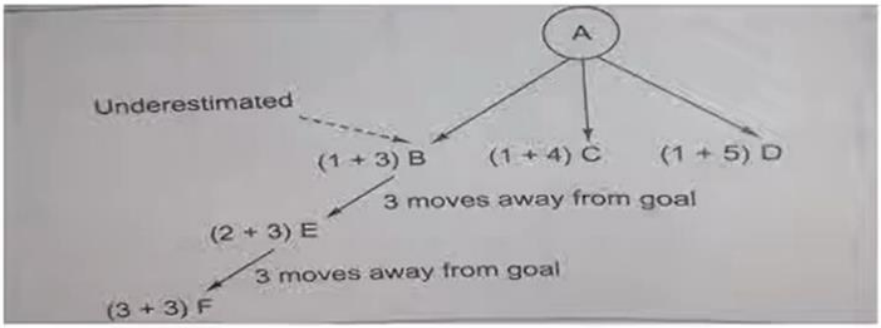
- A* Algorithm is one of the best path finding algorithms.
- But it does not produce the shortest path always.
- This is because it heavily depends on heuristics.

OPTIMAL SOLUTION BY A* ALGORITHM

- A* ALGORITHM FINDS OPTIMAL SOLUTION IF HEURISTIC FUNCTION IS CAREFULLY DESIGNED & IS UNDERESTIMATED.

UNDERESTIMATION: <HEURISTIC VALUE IS LESS THAN ACTUAL VALUE FROM NODE 'N' TO GOAL NODE >

IF '*h*' NEVER OVERESTIMATES ACTUAL VALUE FROM CURRENT TO GOAL, THEN A* ALGORITHM ENSURES TO FIND AN OPTIMAL PATH TO A GOAL (IF EXISTS).



node X to goal node. In Fig. 2.11, start node A is expanded to B, C, and D with *f* values as 4, 5, and 6, respectively. Here we are assuming that the cost of all arcs is 1 for the sake of simplicity. Note that node B has minimum *f* value, so expand this node to E which has *f* value as 5. Since *f* value of C is also 5, we resolve in favour of E, the path currently we are expanding. Now node E is expanded to node F with *f* value as 6. Clearly, expansion of a node F is stopped as *f* value of C is now the smallest. Thus, we see that by underestimating heuristic value, we have wasted some effort but eventually discovered that B was farther away than we thought. Now we go back and try another path and will find the optimal path.

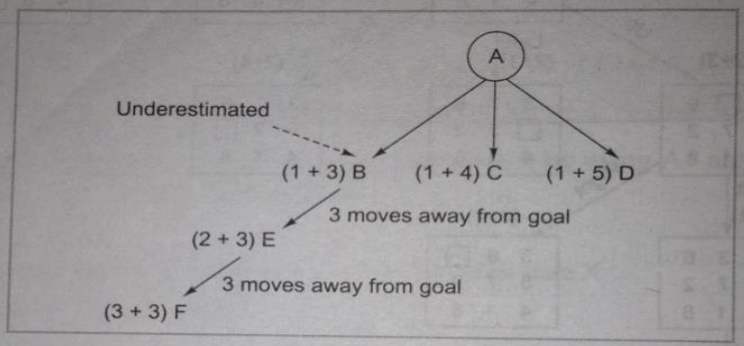
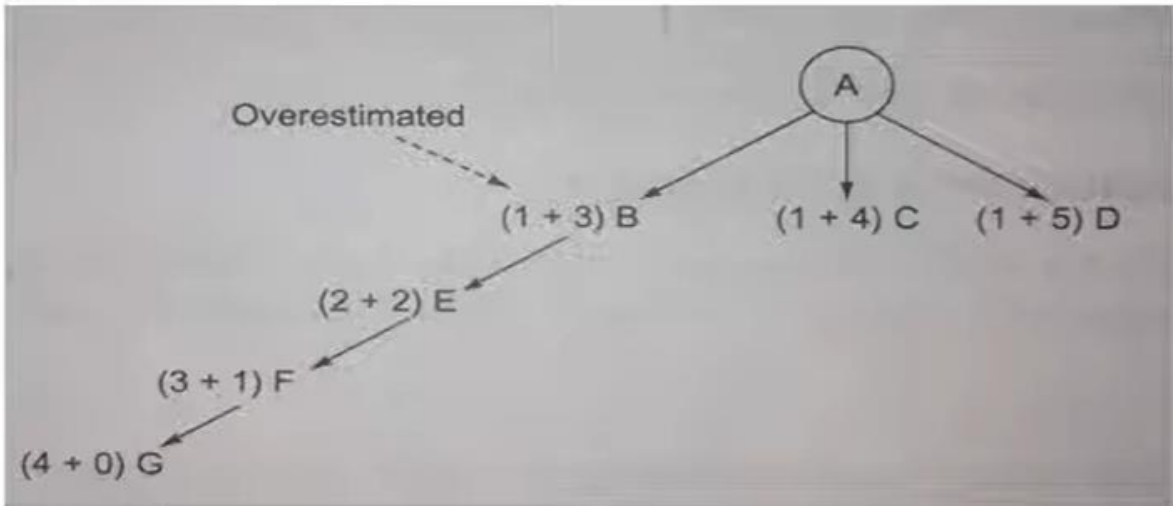


Figure 2.11 Example Search Graph for Underestimation

OVERESTIMATION: <ESTIMATED HEURISTIC VALUE IS GREATER THAN ACTUAL VALUE FROM NODE 'N' TO GOAL NODE >

BY OVERESITMATING '*h*' NOT GUARANTEED TO FIND THE SHORTEST PATH.



Overestimation

Let us consider another situation. Here we are overestimating heuristic value of each node in the graph/tree. We expand B to E, E to F, and F to G for a solution path of length 4. But assume that there is a direct path from D to a solution giving a path of length 2 as *h* value of D is also overestimated. We will never find it because of overestimating h(D).We may find some other worse solution without ever expanding D. So by overestimating *h*, we cannot be guaranteed to find the shortest path.

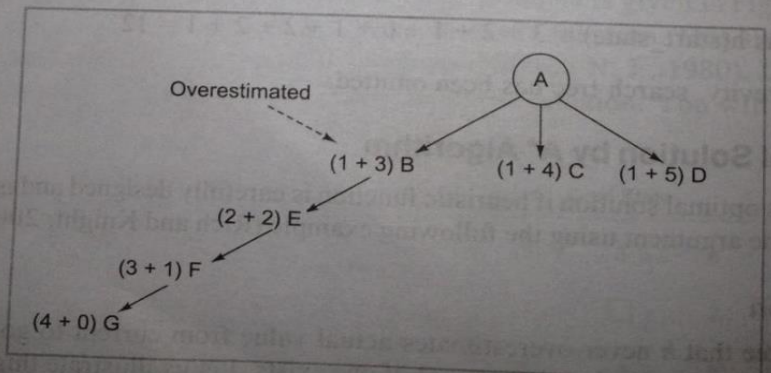


Figure 2.12 Example Search Graph for Overestimation

Admissibility, Monotonicity, Informedness:

Three questions we ask about heuristic algorithms?

- Is it guaranteed to find the shortest path?
- Is it better than another heuristic?
- Does it make steady progress toward a goal?

Admissibility of A*

A search algorithm is *admissible*, if for any graph, it always terminates in an optimal path from start state to goal state, if path exists. We have seen earlier that if heuristic function 'h' underestimates the actual value from current state to goal state, then it bounds to give an optimal solution and hence is called admissible function. So, we can say that A* always terminates with the optimal path in case *h* is an *admissible heuristic function*.

2.5.8 Monotonic Function

A heuristic function *h* is monotone if

1. ∀ states Xi and Xj such that Xj is successor of Xi
   $h(Xi) - h(Xj) \leq$ cost (Xi, Xj) i.e., actual cost of going from Xi to Xj
2. h (Goal) = 0

In this case, heuristic is locally admissible, i.e., consistently finds the minimal path to each state they encounter in the search. The monotone property, in other words, is that search space which is every where locally consistent with heuristic function employed, i.e., reaching each state along the shortest path from its ancestors. With monotonic heuristic, if a state is rediscovered, it is not necessary to check whether the new path is shorter. Each monotonic heuristic function is admissible.

- A cost function *f* is monotone if $f(N) \leq f(succ(N))$
- For any admissible cost function *f*, we can construct a monotonic admissible function.