Assignment 2 - Artificial Intelligence

1. Suppose you decide not to keep the explored set (i.e., the place where the explored nodes are saved) in the A* algorithm.
a. If the heuristic used is admissible, will the new algorithm still be guaranteed to find the optimal solution? Explain why (or why not). [2 marks]

Ans.
If we are given that the heuristic is admissible, then we can prove with the help of contradiction that the algorithm will give an optimal solution.
We know that for the A* algorithm:
$f(n) = g(n) + h(n)$

Let us assume that there is an optimal path that has a cost X. And our algorithm gives a non-optimal cost Y. This would mean $X<Y$.
This means that there must have been a node on the path that remained unexpanded because if all the nodes on the optimal path had been expanded, we would have got an optimal cost X.
$g^*(n)$ is the cost of optimal path from start to n
$h^*(n)$ is the cost of the optimal path from n to the nearest goal
For our algorithm: $f(n) = Y$
$f(n) > X$
$f(n) = g(n) + h(n)$
$f(n) = g^*(n) + h(n)$ since n is on an optimal path
$f(n) <= g^*(n) + h^*(n)$ because of admissibility, $h(n)<=h^*(n)$
$f(n) <= X$ (because $X = g^*(n) + h^*(n)$)

We can see a contradiction here. This means that the A* algorithm must return the optimal cost and optimal path and cannot return any other suboptimal path.

b. Will the new algorithm be complete? Explain. [1 marks]

Ans.
No, the new algorithm will not be complete.
This is because for an algorithm to be complete, it must return an optimal path and optimal cost if it exists or declare that such a path does not exist.
In our case, since we are not keeping track of explored nodes, we are going to keep visiting them again and again, hoping to find an optimal path, whereas in reality, we would just be going in a loop. Thus, our algorithm would work in the cases where a path exists. But in cases where there is no path, our algorithm would fail, and it would keep on going, making it non-complete.

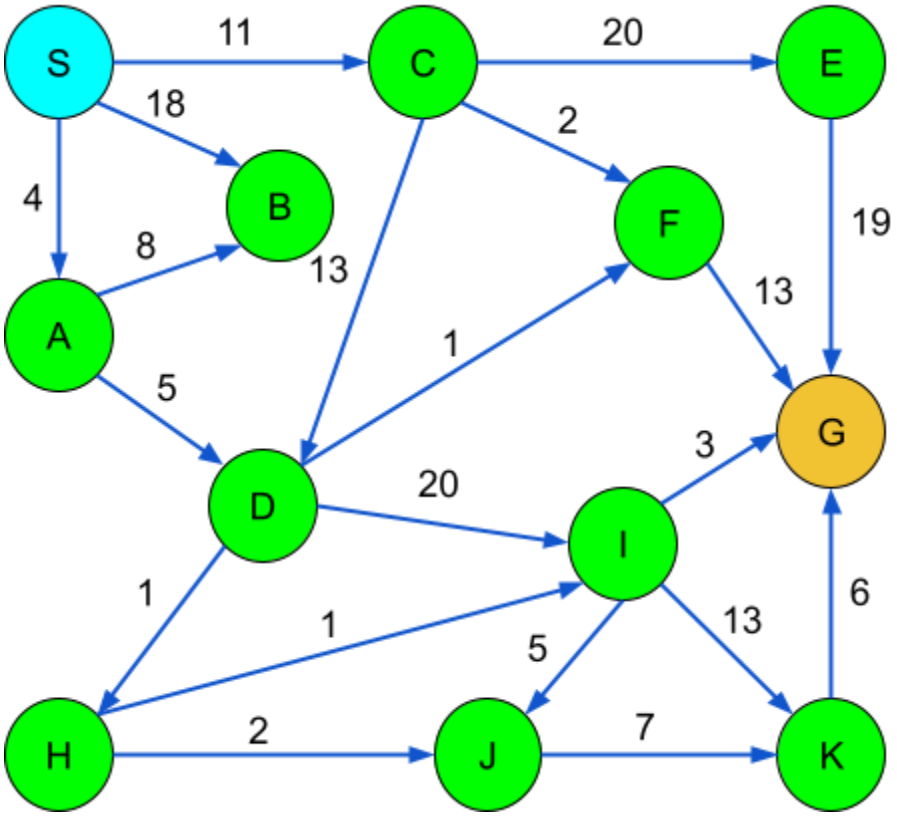
c. Will the new algorithm be faster or not? Explain. [1 marks]

Ans.
The new algorithm would not be faster.
Since we are not keeping track of the explored nodes now, our algorithm would not know which nodes have already been visited. As a result, the nodes will be revisited many times. This would make our algorithm slower overall because in some cases, our algorithm can even get stuck in an infinite loop where it keeps searching for a path to the goal node which may not even exist.

2. Given the graph, find the cost-effective path from S to G. S is the source node, and G is the goal node using A* [2 marks], BFS [2 marks], and Dijkstra algorithm. [2 marks]
Please note: cost written on edges represents the distance between nodes. The cost written on nodes represents the heuristic value. Heuristic for Goal Node G is 0. Distance from K to G is 6.

Ans.

First, we are going to do the Dijkstra algorithm, as it works only on the basis of the distance between two nodes and not any other heuristic.
Below is the graph that we are going to need.



A few things to note here are that we have “S” as the start node and “G” as the goal node. Also, the given graph is a directed graph.
So, we are going to start from S.
Initially, when we are at the start node S, we do not know anything about the other nodes. The only thing we know is that the distance from the current node to S is 0.
So, we will start from S and discover the nodes directly connected to S first.
From the graph, we can see that we can visit three nodes, C, B, and A, from S. So we will write their distances in a priority queue, where our priority will be to expand the nearest node first. Since we also need the path, we are also going to keep track of the node chosen to get to a node in our queue.
For S, we get:

Node	Cost	Path
A	4	From S
B	18	From S
C	11	From S

Arranging the nodes according to priority:

Node	Cost	Path
A	4	From S
C	11	From S
B	18	From S

These are nodes that can be visited from the source node. Now, we will move on to the next node according to our priority queue and choose A node.
From A, we can reach B and D. For D, our distance will be 4+5=9. For B, although we have already visited it once, we will still check if this path is better in terms of the cost. So, through A, the cost for B is 4+8=12, which is less than 18.
We get:

Node	Cost	Path
D	9	From A
C	11	From S
B	12	From A

Again, we will choose the first node and expand it. Here, we have D as the first node. We can see that D can reach three nodes: F, I, and H. So we get:

Node	Cost	Path
F	10	From D
H	10	From D
C	11	From S
B	12	From A
I	29	From D

Now, we have two nodes, F and H, having the same priority, so we will choose randomly between them. I am going to expand F first.
F can visit only one node, G. So, we get:

Node	Cost	Path
H	10	From D
C	11	From S
B	12	From A
G	23	From F
I	29	From D

Now, although we have reached our goal G, we still need to visit and expand our node H as it may have a better path to G (Since G is not at the top currently). So we get:

Node	Cost	Path
I	11	From H
C	11	From S
J	12	From H
B	12	From A
G	23	From F
I	29	From D

Now we have “I” at the top of our priority table, so we will expand “I”. We get:

Node	Cost	Path
C	11	From S
J	12	From H
B	12	From A
G	14	From I
K	24	From I
I	29	From D

Note that from I, we have decreased the cost to G, but still, G is not at the top, so we must go on expanding our priority nodes. Also, we can get to J from I as well, but since the cost of reaching J from I is more than what it already is, we are going to ignore it.

Moving forward, we will expand C. We can visit nodes E and F. But since we have already visited node F at a lower cost before, we will ignore it.
So, we get:

Node	Cost	Path
J	12	From H
B	12	From A
G	14	From I
K	24	From I
I	29	From D
E	31	From C

Now, we will expand node J.
This time, we are going to update the cost to reach node K, as from J, we can reach K at 12+7=19 cost only, which is better than 24.

Node	Cost	Path
B	12	From A
G	14	From I

K	19	From J
I	29	From D
E	31	From C

Now we are left with node B. Since it does not direct to any other node, the cost of reaching goal G through B can be considered infinite. So, we will remove node B from our table.
We get:

Node	Cost	Path
G	14	From I
K	19	From J
I	29	From D
E	31	From C

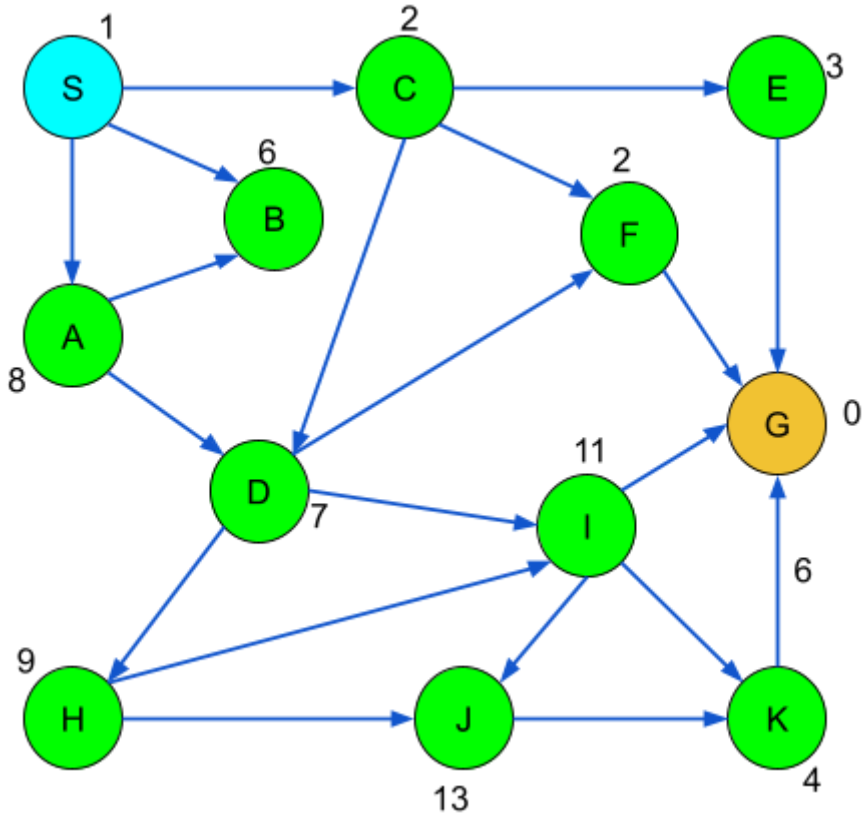
Now we have our node G at the top. Thus we have our solution now.
Therefore, the cost to reach the goal is **14**.
To see the path we need to take to reach G, we will trace the path column we have made in our tables. To do this, we just need to do a simple thing. Whichever node you are at, just find the table which has that node at the top and you will get the node from which you reached this node.

We reach G from I.
We reach I from H.
We reach H from D.
We reach D from A.
We reach A from S.

Therefore, our path is S -> A -> D -> H -> I -> G.

Now we will cover Best first search algorithm.
In this algorithm, the only difference is that instead of taking the distance as the cost, we will take the heuristic as the cost.

This is the graph that we have with the heuristics:



In the best first search algorithm, we start from the start node and look at all the frontier nodes. It then chooses the frontier node having the smallest heuristic value and then expands its children node till it reaches the goal node, when it returns.
Here, we will start from the source node “S” and follow the algorithm till we reach the goal node “G”.
Again, we will make a priority queue and fill heuristic values in it for the frontier nodes.
And as before, since we also need the path we took, we are going to maintain a column for path denoting the path it took to reach that node.

Node	Cost	Path
C	2	From S
B	6	From S
A	8	From S

From the above queue, we can see that node C has the least heuristic value. Thus, we will expand node C and make a queue with its children nodes’ heuristic values.

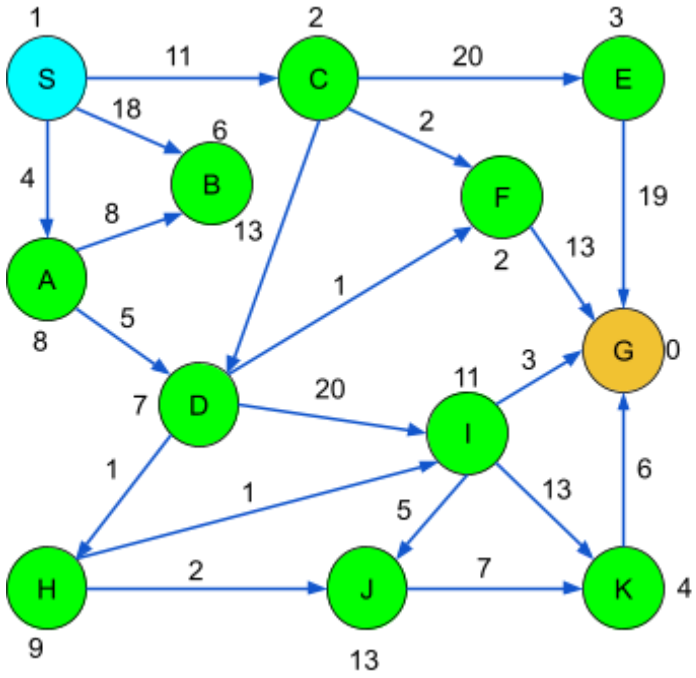
Node	Cost	Path
F	2	From C
E	3	From C
D	7	From C

Now we will expand node F. Since we can only reach one node from F, we will have only one element in the priority queue.

Node	Cost	Path
G	0	From F

Now, since we have reached our goal node, the algorithm will return.
The path chosen by BFS will be -
S -> C -> F -> G

Now, we will cover the A* search algorithm.
This is a complex algorithm that takes into account both the distance between the nodes as well as the heuristic values. The function for the A* search algorithm is:
f(n) = g(n) + h(n)
Thus, we will be taking the sum of the distance between the nodes and the heuristic values as our cost for this algorithm.



Here, S is the source node, and G is the goal node.
We are going to make a priority queue having the cost as the sum of the distance between the nodes and the heuristic values. And since we need to know the path, we are also going to maintain a column that will tell us the path it took to reach that node.
So, we will start from the source node S and choose the first neighbor having the least cost.

Node	Cost	Path
A	12	From S
C	13	From S
B	24	From S

Now, we are going to choose the node with the lowest cost at the top of the queue, i.e., A. We will see that A can reach nodes B and D. But since node B has already been reached, we will check if there is a better path to reach B having a lower cost.
We will see that B can be reached through A by a cost of (4+8) + 6 = 18, which is lower than 24.
So, we will replace the cost for B to 18 and update the path it took.

Node	Cost	Path
C	13	From S
D	16	From A
B	18	From A

Now, we will explore node C.
C can reach three nodes - E, F, and D. Again, we will check if we can reach node D with a lower cost.
The cost to reach node D through C will be = (11+13) +7 = 31. Since this is more than the cost that we already have, we will ignore this.

Node	Cost	Path
F	15	From C
D	16	From A
B	18	From A
E	34	From C

Now, we will expand node F as it is at the top.
We will see that F can reach only node G.

Node	Cost	Path
D	16	From A
B	18	From A
G	26	From F
E	34	From C

Now, we will expand node D. This node can reach nodes F, I, and H.

Node	Cost	Path
F	12	From D
B	18	From A
H	19	From D
G	26	From F
E	34	From C
I	40	From D

Now we will expand node F. Note that although we have already expanded node F once, but since this is a different path, the costs will be different. So we need to take this into consideration.
F can reach only node G. The cost this time will be (4+5+1) + 13 = 23. So, we will update the cost for node G to 23.

Node	Cost	Path
B	18	From A
H	19	From D
G	23	From F
E	34	From C
I	40	From D

Now we will expand node B. But since B does not direct to any node, we will remove B from our queue as it cannot take us to our goal node.

Node	Cost	Path
H	19	From D
G	23	From F
E	34	From C
I	40	From D

Now we will expand node H. H directs to only one node J.
Cost for J will be (4+5+1+2) + 13 = 25.

Node	Cost	Path
G	23	From F
J	25	From H
E	34	From C
I	40	From D

Now, we have arrived at our goal node, and the algorithm will return here.
Thus, the cost to reach the goal is 23.
The path taken to reach the goal node from the source node is:
S -> A -> D -> F -> G

Computational

Comparison between UnformCost Search algorithm and A* search algorithm in terms of performance:

- Uniform Cost Search algorithm generally takes more time, especially in the cases where the graph's size is large. This is because this algorithm explores all the nodes (time complexity = O(V^2)) before giving the final solution.
- A* search algorithm is generally more efficient because it does not need to explore all the nodes to give the best solution. This is because it uses a heuristic to make choices and arrive at a solution. But that makes it dependent on the heuristic chosen. If a complex heuristic is chosen, then the time complexity will increase accordingly and can go even higher than the Uniform Cost search algorithm.

Heuristics chosen:
Admissible: I have taken the distance to the nearest city as the admissible heuristic. It gives the optimal solution.

Inadmissible: I have chosen the number of cities a city is directly connected to as the inadmissible heuristic. It gives a suboptimal solution with a distance that is more than the least distance.