Introduction:

This report will discuss the following grid-based search algorithms –

- Breadth-First Search
- Depth-First Search
- Dijkstra's Algorithm
- A* Algorithm

This report will first cover the basic understanding and implementation of all the algorithms individually. Next this will focus on the similarities and difference among these algorithms. And finally, this algorithm will present the pseudo-code of the algorithm implemented as a part of this assignment and then the results and conclusion obtained by this exercise.

The discussion and argument presented in this report will be in-context of a grid search problem. Here these algorithms are implemented to find a path from **Start** to **Goal** in the grid. This report will focus on the implementation details and performance comparison of these algorithms.

At last, this report includes the list of references used to complete this assignment.

Some important terminology –

Visited node (Cell) – The node (cell) which is a neighbor of some currently **explored** node and the algorithm is currently looking at the property (is it already visited, it is free space to move) of this node.

Explored node (Cell) – The algorithm visits the neighbors of the explored node. Exploring the node literally means looking at the properties of all its neighbors.

Parent node (Cell) – In the above context, if the algorithm is currently exploring node A and then visits the node B from A, then A is the parent of B.

Breadth-First Search (BFS):

Breadth First Search Algorithm is a very commonly used algorithm to find the shortest path between the **Start** and the **Goal** node in an unweighted grid-based map. The BFS algorithm starts at the **Start** node and explores the neighbors of that node first, before moving to the next level. It explores nodes in layers. It generally spans the tree (formed by arranging the nodes in a level-by-level fashion) by its width.

It maintains a FIFO (First-In-First-Out) queue of the nodes that are to be explored in the next step by visiting the neighbor of the node that is currently being explored.

** In the implementation I have used a python list as a FIFO Queue.

One major advantage of BFS algorithm is that it finds the shortest possible path in the unweighted grid (Map) and is very reliable if the desired output is the shortest path.

One major drawback of this algorithm is it explores and visits a lot of nodes which makes it slower as compared to other available algorithms.

The time complexity of BFS can be given by O(V+E), where V is the number of nodes and E is the edges (number of connections between the nodes.)

In context of only the search method, BFS algorithm is **complete** as when applied to infinite grids represented implicitly, BFS will eventually find the **goal**.

Algorithm: 1 **Breadth First Search**

```
bfs(grid, start, goal)

1.  initially mask all the nodes to not visited
2.  initialize a queue
3.  push start in the queue
4.  mark start = visited
5.  while queue is not empty
6.      N = queue.pop(0)    // explore node that was added first in the queue
7.      If N = = goal
8.          return found, path
9.      for each unvisited neighbor n of N
10.         n.parent = N
11.         mark n = visited
12.         push n to queue
```

Depth-First Search (DFS):

Depth First Search Algorithm is a very commonly used algorithm to find a path between two connected nodes like the **Start** and the **Goal** node in an unweighted grid-based map. The DFS algorithm does not necessarily return the shortest path always. DFS starts at the **Start** node and explores the neighbors in the next level, before backtracking (exploring the nodes) of the same level. It explores nodes in depth-wise fashion. It generally spans the tree (formed by arranging the nodes in a level-by-level fashion) along one single branch at a time.

It maintains a LIFO (Last-In-First-Out) stack of the nodes that are to be explored in the next step by visiting the neighbor in the next level of the node that is currently being explored.

** In the implementation I have used a python list as a LIFO stack.

One major advantage of DFS algorithm is that it finds a possible path in the unweighted grid (Map) as quickly as possible since it does not explore all the nodes layer-by-layer and can reach to a depth very quickly.

One major drawback of this algorithm is it does not guarantee to give the most optimal solution (shortest path) and can result in sub-optimal solution if one exists. Therefore, DFS is used if it a good trade-off of getting a solution quicker over getting the best solution.

The time complexity of DFS can be given by O(V+E), where V is the number of nodes and E is the edges (number of connections between the nodes.)

In context of only the search method, DFS algorithm is **not-complete** as when applied to infinite grids represented implicitly, DFS will eventually get lost in parts and never reach the **goal**.

Algorithm: 2 **Depth First Search**

```
dfs(grid, start, goal)

1.  initially mark all the nodes to not visited
2.  initialize a stack
3.  push start in the stack
4.  mark start = visited
5.  while queue is not empty
6.      N = stack.pop(-1)    // explore node that was added last in the queue
7.      If N = = goal
8.          return found, path
9.      for each unvisited neighbor n of N
10.         n.parent = N
11.         mark n = visited
12.         push n to stack
```

Dijkstra’s Algorithm:

Dijkstra’s Algorithm is a very commonly used algorithm to find the shortest path between the **Start** and the **Goal** node in a weighted grid-based map. The Dijkstra’s algorithm starts at the **Start** node and visits the neighbors of that node first and then calculates the cost of exploring that neighbor and keeps track of the cost (cost-to-come) of exploring each of the neighbors. After arranging all the neighbors in the ascending order of cost, it picks the node with lowest cost and explores that node.

It maintains a minimum priority queue of the nodes that are to be explored in the next step by visiting the neighbor of the node that is currently being explored.

** In the implementation I have used a python list as a minimum priority Queue.

It sorts the minimum priority queue based on the g(s) function. That is the distance between the current position and the **Start** node. In other words, we can say that it is based on the cost-to-come.

One major advantage of Dijkstra’s algorithm is that it finds the shortest possible path in the weighted grid (Map) and is very reliable if the desired output is the shortest path.

One major drawback of this algorithm is it explores and visits a lot of nodes which makes it slower as compared to other available algorithms.

This algorithm in its essence is the weighted version of the Breadth-First Search algorithm. Majorly Dijkstra’s algorithm is used to find the shortest distance of all the nodes from the start node but here in this assignment it is implemented as a method to find the shortest possible distance between **Start** and **Goal** in a weighted graph.

** Here the weights are kept to 1 as it was required for this assignment. In this case the Dijkstra’s algorithm behaves exactly like the BFS algorithm.

The time complexity of DFS can be given by O((V+E)log(V)), where V is the number of nodes and E is the edges (number of connections between the nodes.)

Algorithm: 3 **Dijkstra's**

```
dijkstra(grid, start, goal)

1.  initialize a priorityqueue
2.  push start in the priorityqueue
3.  initialize a matrix dist to store the current distance (from start) of all the nodes
4.  initialize the distance of all the nodes to be infinite                //path is unknown
5.  while priorityqueue is not empty
6.      N = priorityqueue.pop(0)          // sort the queue based on minimum dist(node)
7.      If N = = goal
8.          return found, path
9.      for each neighbor n of N
10.         if dist(n) > dist(N) + cost(N,n)          // cost(N,n) is the cost to move to n
11.             dist(n) = dist(N) + cost(N,n)         //cost(N,n) = 1 for the assignment
12.             n.parent = N
13.             push n to priorityqueue
```

A Star (A*) Algorithm:

A* Algorithm is a very commonly used algorithm to find the path between the **Start** and the **Goal** node in a weighted grid-based map. The A* algorithm starts at the **Start** node and visits the neighbors of that node first and then calculates the cost of exploring that neighbor and keeps track of the cost of exploring each of the neighbors. Unlike Dijkstra's algorithm here the cost is a combination of the cost-to-come and the heuristic. Heuristic is a function that approximates the cost of the cheapest possible path. After arranging all the neighbors in the ascending order of cost, it picks the node with lowest cost and explores that node.

It maintains a minimum priority queue of the nodes that are to be explored in the next step by visiting the neighbor of the node that is currently being explored.

** In the implementation I have used a python list as a minimum priority Queue.

It sorts the minimum priority queue based on the $f(s) = g(s) + h(s)$ function. That is the combination of the distance between the current position and the **Start** node, and the estimate of the distance between the current position and the **Goal**.

The A* algorithm does not always promise to give optimal solutions. The only necessary condition for A* algorithm providing optimal solution is that the heuristic function $h(s)$ should be admissible. An admissible heuristic means that it never overestimates the actual cost of traveling from the **Start** to the **Goal**. It always returns a cost lower than the actual cost of reaching the **Goal**.

** In the implementation I have used **Manhattan Distance** as the heuristic.

One major advantage of A* algorithm is that it finds the shortest possible path in the weighted grid (Map) and quickly converges to the optimal solution and is very reliable if the desired output is the shortest path. * This is true only if the heuristic is admissible

One major drawback of this algorithm is that it is difficult to always come up with an admissible heuristic function.

This algorithm in its essence is smart version of the Dijkstra's algorithm as now it has a brain and explores the nodes in a particular opposed to exploring all the available nodes to find the shortest path.

** Here the weights are kept to 1 as it was required for this assignment. In this case the A* algorithm will give the same path as BFS and Dijkstra's algorithm, but the number of modes explored will be less as compared to both the algorithm.

The time complexity of A* start algorithm depends on the heuristic and thus it provides the user with the flexibility to come up with a better time complexity.

Algorithm: 3 **A***

```
dijkstra(grid, start, goal)

1.  initialize a priorityqueue
2.  push start in the priorityqueue
3.  initialize a matrix COST to store the current COST (f = g + h) of all the nodes
4.  initialize the cost of all the nodes to be infinite                //path is unknown
5.  while priorityqueue is not empty
6.      N = priorityqueue.pop(0)        // sort the queue based on minimum COST(node)
7.      If N = = goal
8.          return found, path
9.      for each neighbor n of N
10.         if COST(n) > COST(N) + cost(N,n)        //cost(N,n) is the cost of moving to n
11.             COST(n) = COST(N) + cost(N,n)      // cost(N,n) =1 for assignment
12.             n.parent = N
13.             push n to priorityqueue
```

Implementation:

Pseudo Code of Implementation

```
searchPath(grid,start,goal,type)             // type is the algorithm – BFS,DFS,dijkstra,astar

1.  obtain grid size
2.  initialize all the nodes to not visited
3.  initialize the dist matrix for all the nodes to infinite
4.  start = visited
5.  if astar
6.      dist = g(s) + f(s)                   // cost-to-come  + heuristic (Manhattan distance)
7.  else
8.      dist = g(s)                                                         // cost-to-come
9.  make queue
10. add start to queue
11. while queue is not empty
12.     if BFS
13.         N = queue.pop(0)                                              // pop first added
14.     elif DFS
15.         N = queue.pop(-1)         // pop last added
16.     else sort queue based on dist(N)                         //take node with lowest cost
17.         N = queue.pop(0)
18.     For each neighbor n of N
19.         If ((BFS or DFS) and not visited) or (((dijkstra or astar) and dist(n)>dist(N)+1))
20.             mark n = visited
21.             h  = Manhattan_dist_to_goal(n)
22.             if astar
23.               dist = g(s) + f(s)        // cost-to-come  + heuristic (Manhattan distance)
24.             else
25.               dist = g(s) + f(s)                                        // cost-to-come
26.             dist(n)=dist(N) + 1                                         // since cost is 1
27.             push n to queue
28.
29.
30.
```

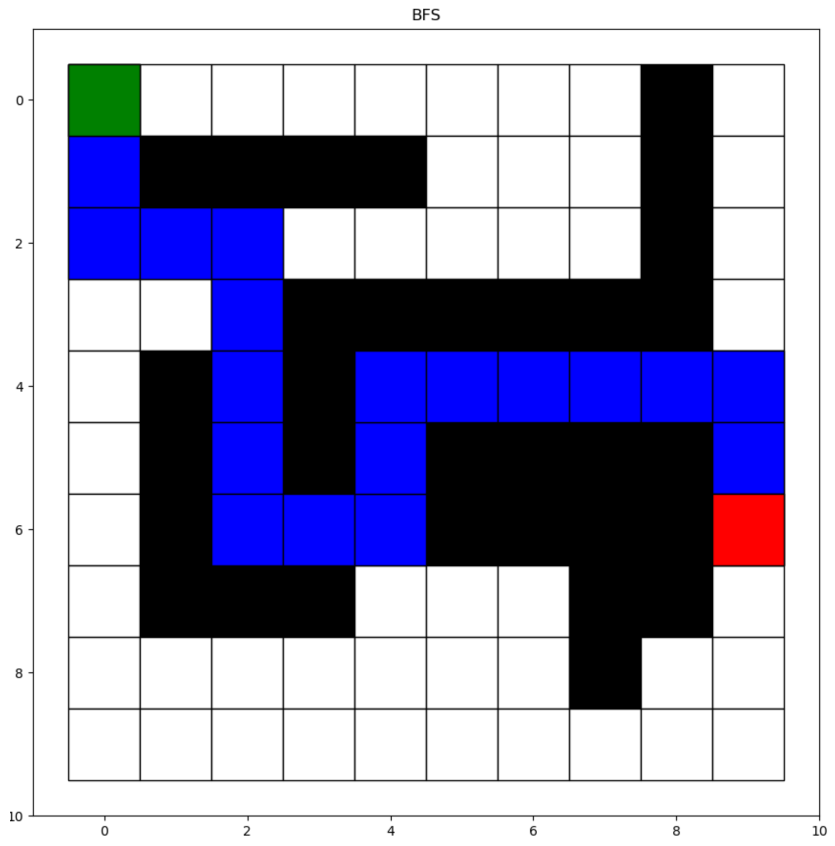
The searchPath(grid, start, goal, type) function takes in grid, goal, start and the type as input and implements that particular algorithm.

Please input type = “BFS” or “DFS” or “dijkstra” or “astar” to check the functionality of that algorithm.

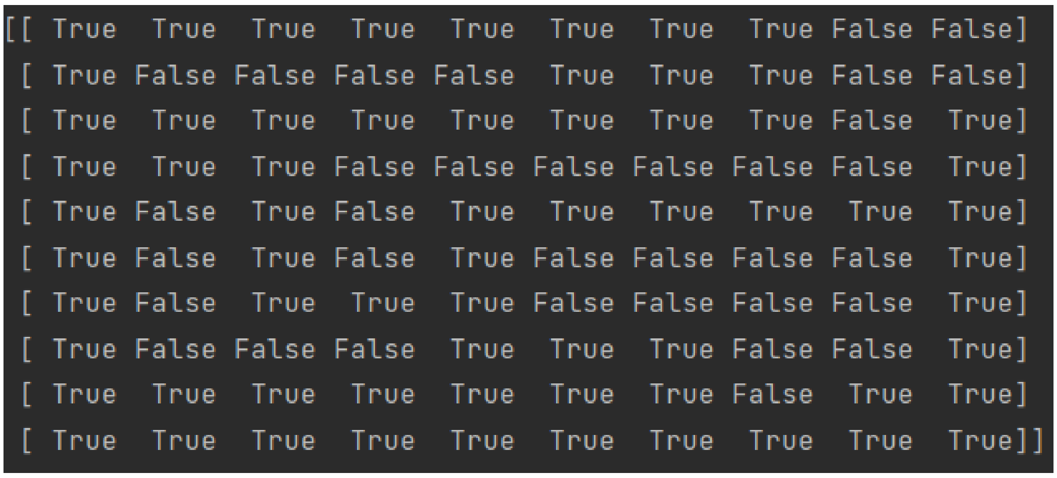
Results:

EXPERIMENT-1

➔ **Breadth First Search**

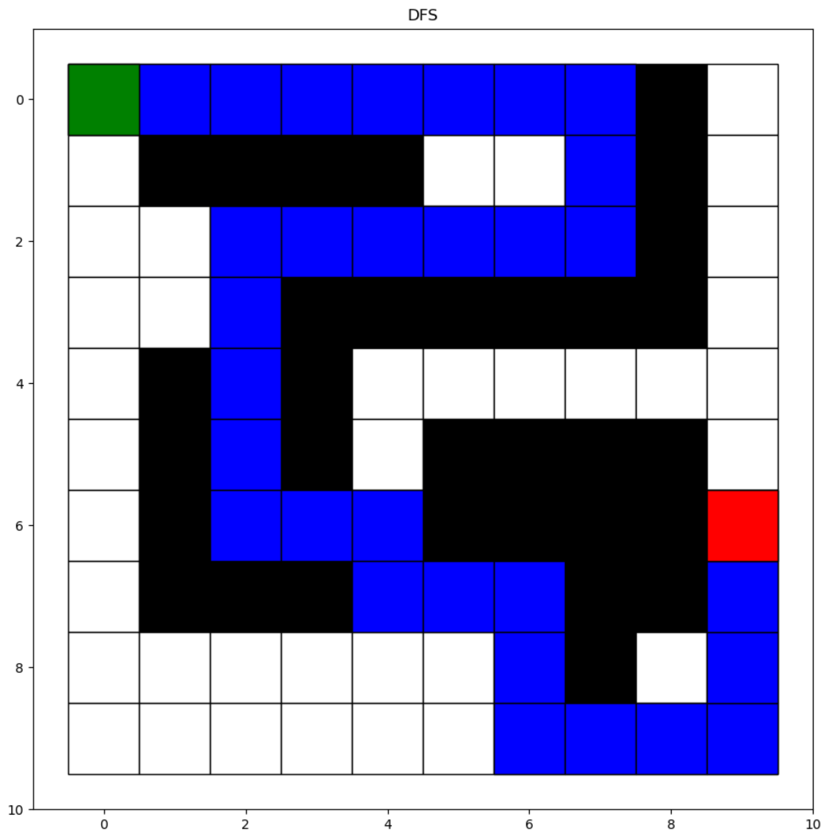


Visited nodes (True represents visited)

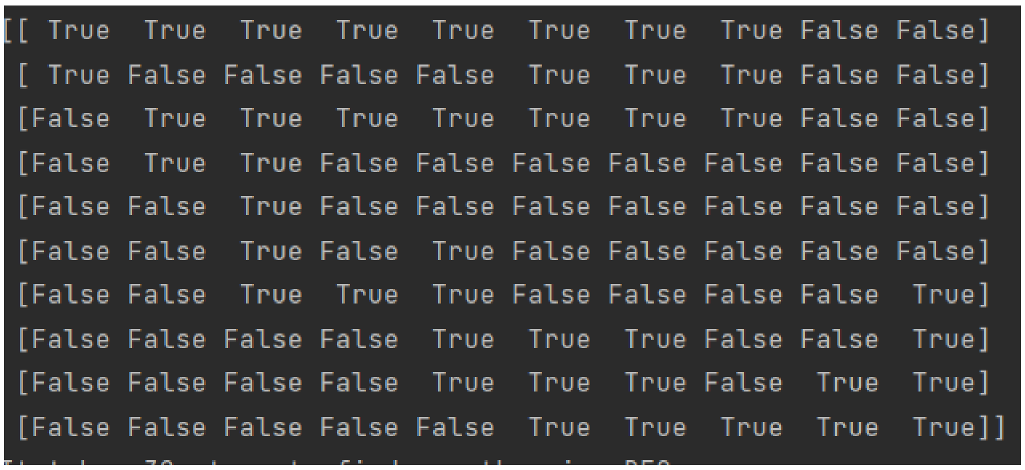


```
[[ True  True  True  True  True  True  True  True False False]
 [ True False False False False  True  True  True False False]
 [ True  True  True  True  True  True  True  True False  True]
 [ True  True  True False False False False False False  True]
 [ True False  True False  True  True  True  True  True  True]
 [ True False  True False  True False False False False  True]
 [ True False  True  True  True False False False False  True]
 [ True False False False  True  True  True False False  True]
 [ True  True  True  True  True  True  True False  True  True]
 [ True  True  True  True  True  True  True  True  True  True]]
```

➔ Depth First Search

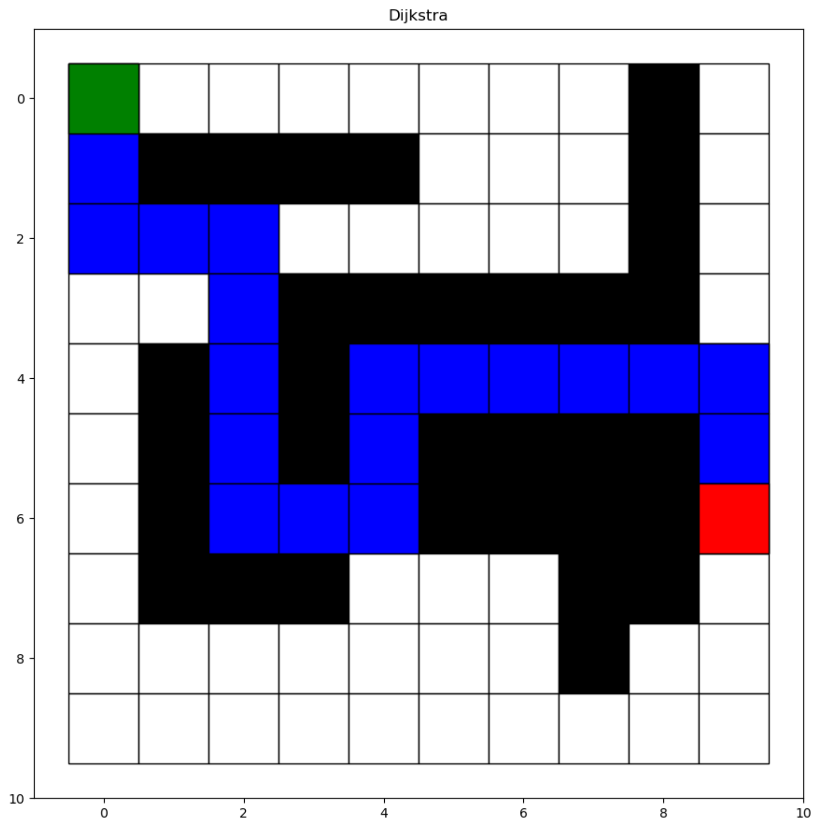


Visited nodes (True represents visited)

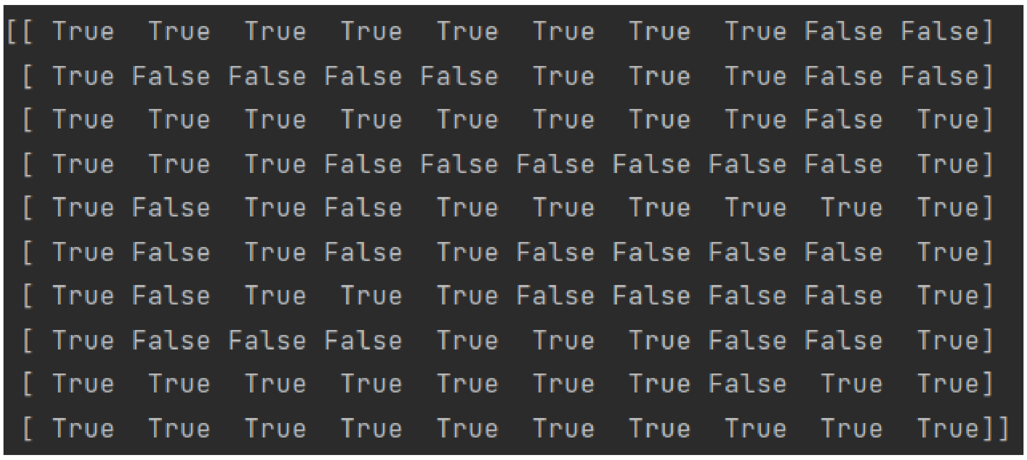


```
[[ True  True  True  True  True  True  True  True False False]
 [ True False False False False  True  True  True False False]
 [False  True  True  True  True  True  True  True False False]
 [False  True  True False False False False False False False]
 [False False  True False False False False False False False]
 [False False  True False  True False False False False False]
 [False False  True  True  True False False False False  True]
 [False False False False  True  True  True False False  True]
 [False False False False  True  True  True False  True  True]
 [False False False False False  True  True  True  True  True]]
[illegible]
```

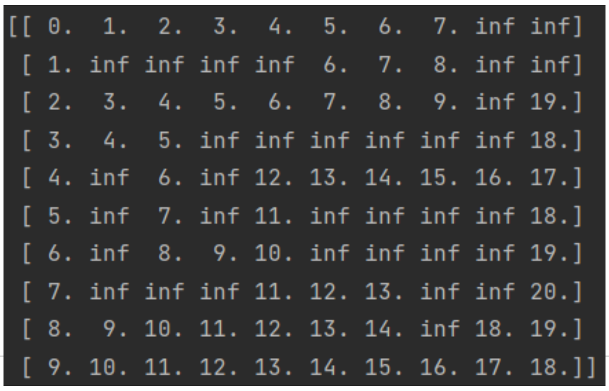
➔ Dijkstra’s Algorithm



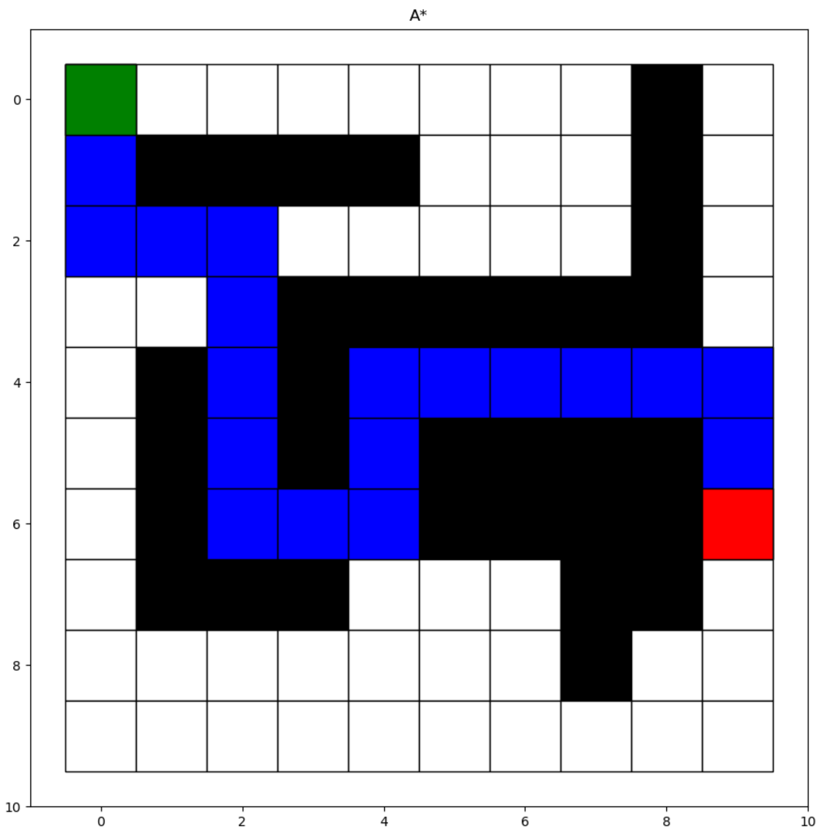
Visited nodes (True represents visited)


```
[[ True  True  True  True  True  True  True  True False False]
 [ True False False False False  True  True  True False False]
 [ True  True  True  True  True  True  True  True False  True]
 [ True  True  True False False False False False False  True]
 [ True False  True False  True  True  True  True  True  True]
 [ True False  True False  True False False False False  True]
 [ True False  True  True  True False False False False  True]
 [ True False False False  True  True  True False False  True]
 [ True  True  True  True  True  True  True False  True  True]
 [ True  True  True  True  True  True  True  True  True  True]]
```

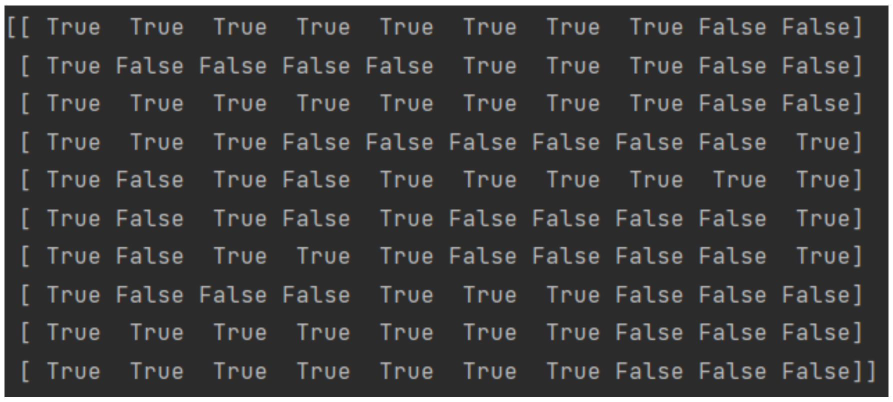
Cost of travel = cost-to-come


```
[[ 0.  1.  2.  3.  4.  5.  6.  7. inf inf]
 [ 1. inf inf inf inf  6.  7.  8. inf inf]
 [ 2.  3.  4.  5.  6.  7.  8.  9. inf 19.]
 [ 3.  4.  5. inf inf inf inf inf inf 18.]
 [ 4. inf  6. inf 12. 13. 14. 15. 16. 17.]
 [ 5. inf  7. inf 11. inf inf inf inf 18.]
 [ 6. inf  8.  9. 10. inf inf inf inf 19.]
 [ 7. inf inf inf 11. 12. 13. inf inf 20.]
 [ 8.  9. 10. 11. 12. 13. 14. inf 18. 19.]
 [ 9. 10. 11. 12. 13. 14. 15. 16. 17. 18.]]
```

➔ A* Algorithm

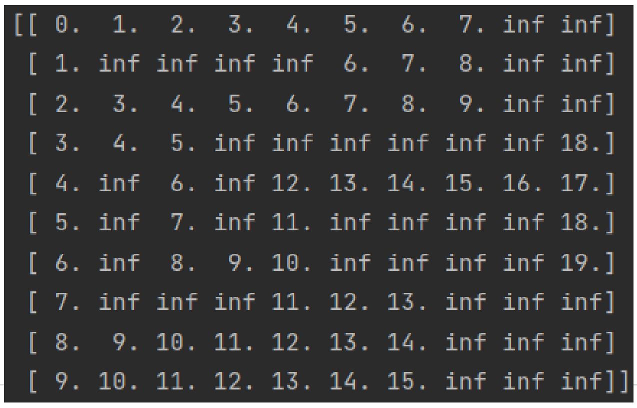


Visited nodes (True represents visited)



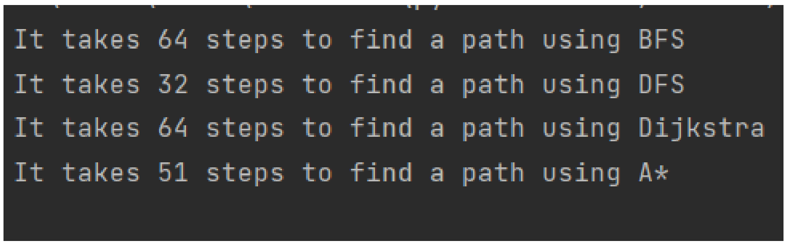
```
[[ True  True  True  True  True  True  True  True False False]
 [ True False False False False  True  True  True False False]
 [ True  True  True  True  True  True  True  True False False]
 [ True  True  True False False False False False False  True]
 [ True False  True False  True  True  True  True  True  True]
 [ True False  True False  True False False False False  True]
 [ True False  True  True  True False False False False  True]
 [ True False False False  True  True  True False False False]
 [ True  True  True  True  True  True  True False False False]
 [ True  True  True  True  True  True  True False False False]]
```

Cost of travel = cost-to-come + Manhattan distance



```
[[ 0.   1.   2.   3.   4.   5.   6.   7.  inf inf]
 [ 1.  inf inf inf inf  6.   7.   8.  inf inf]
 [ 2.   3.   4.   5.   6.   7.   8.   9.  inf inf]
 [ 3.   4.   5.  inf inf inf inf inf inf 18.]
 [ 4.  inf  6.  inf 12. 13. 14. 15. 16. 17.]
 [ 5.  inf  7.  inf 11. inf inf inf inf 18.]
 [ 6.  inf  8.   9. 10. inf inf inf inf 19.]
 [ 7.  inf inf inf 11. 12. 13. inf inf inf]
 [ 8.   9. 10. 11. 12. 13. 14. inf inf inf]
 [ 9. 10. 11. 12. 13. 14. 15. inf inf inf]]
```

The number of steps taken by each algorithm for this Experiment



```
It takes 64 steps to find a path using BFS
It takes 32 steps to find a path using DFS
It takes 64 steps to find a path using Dijkstra
It takes 51 steps to find a path using A*
```

Here, we can see that the DFS algorithm takes less steps as compared to the BFS algorithm and visits less nodes as compared to BFS algorithm (As seen in the visited table). DFS return a sub-optimal solution as we can see from the path formed that there exists a shorter path which the algorithm fails to discover.

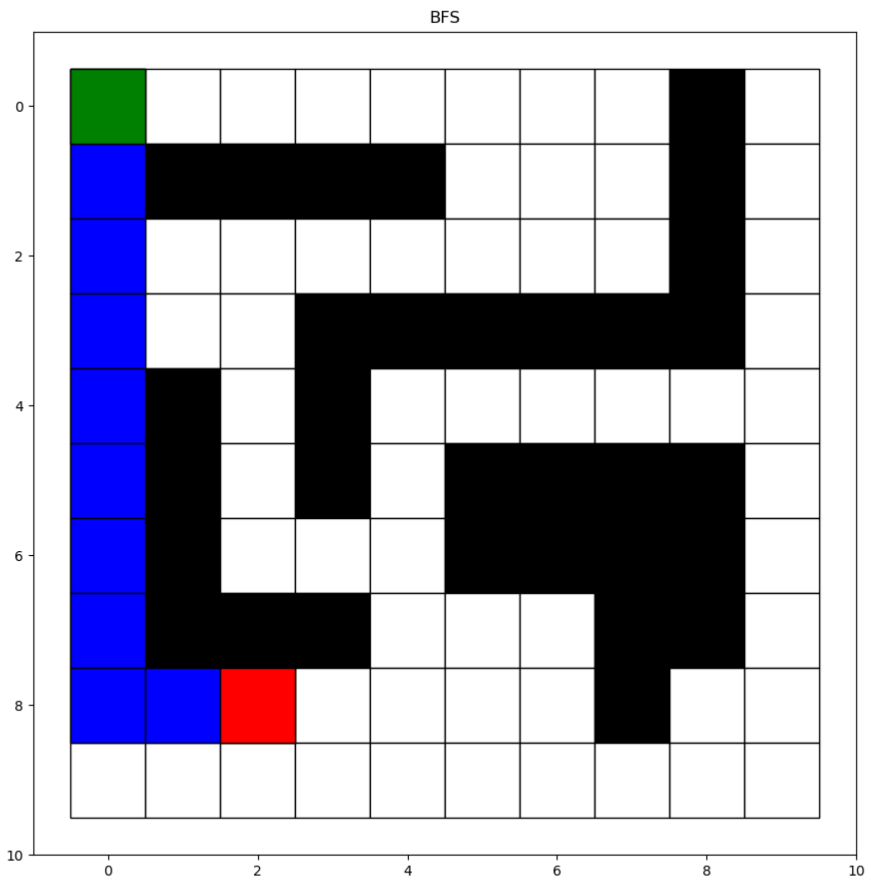
Therefore, we can conclude that DFS gives faster solution as compared to BFS, but it need not be the best possible solution. However, BFS takes more steps to find out the solution but will always return an optimal solution (shortest path).

Here also observe that the Dijkstra's algorithm performs like that of the BFS algorithm, this is because the cost between each node was 1 here. It returns the same path and takes the same number of steps. Hence, Dijkstra's algorithm gives the most optimal solution in case of a weighted graph.

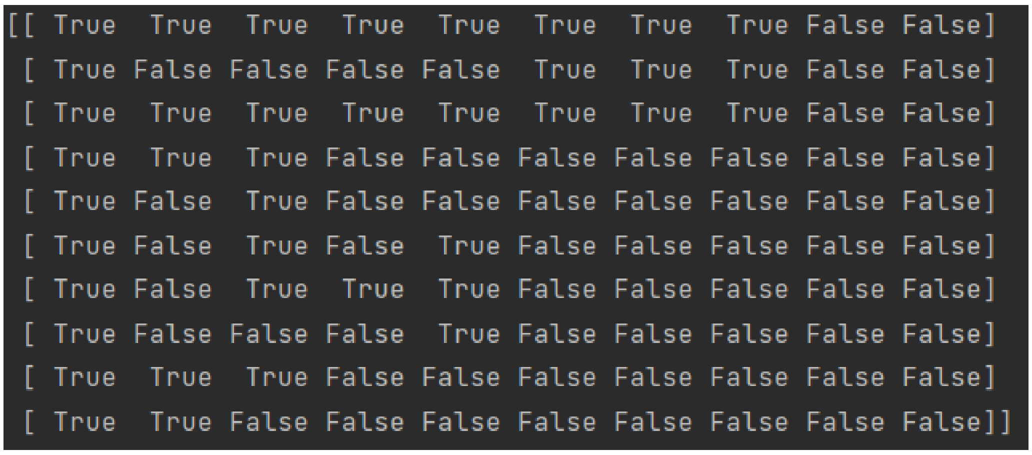
As we can see that A* also returns the same path as BFS and Dijkstra's, i.e. the shortest path. However, it takes a smaller number of steps as compared to Dijkstra's which is evident by the Cost to travel table shown above. It does find the shortest in lesser steps due to an added heuristic function (Manhattan distance between the current node and the goal.)

EXPERIMENT-2

➔ **Breadth First Search**

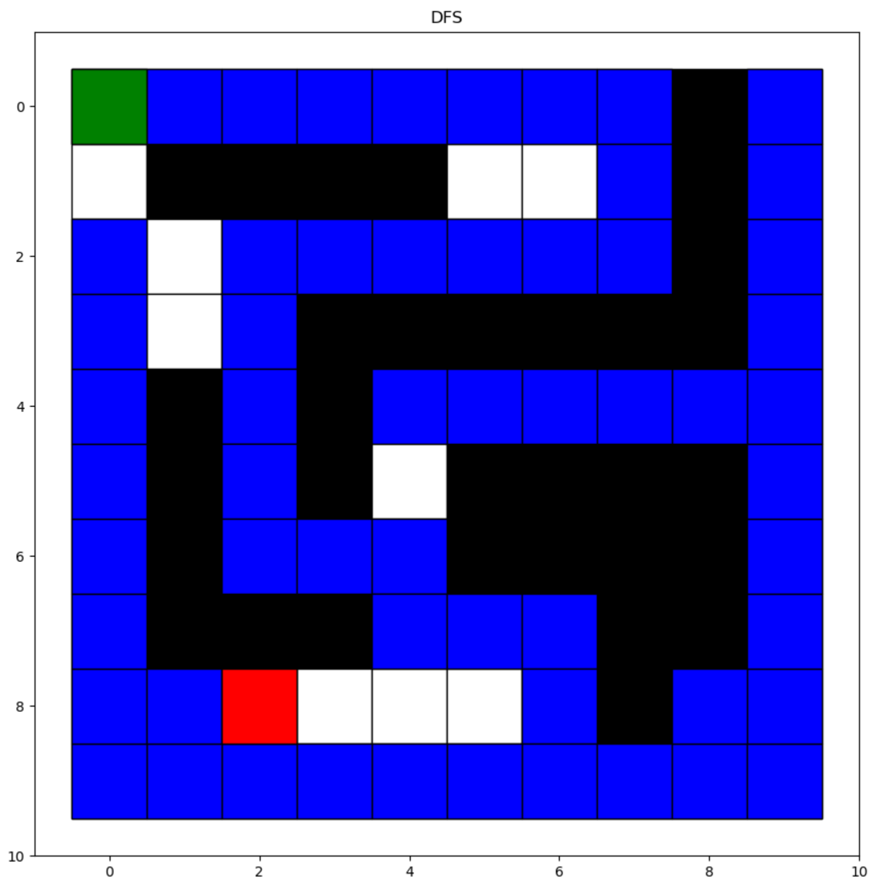


Visited nodes (True represents visited)

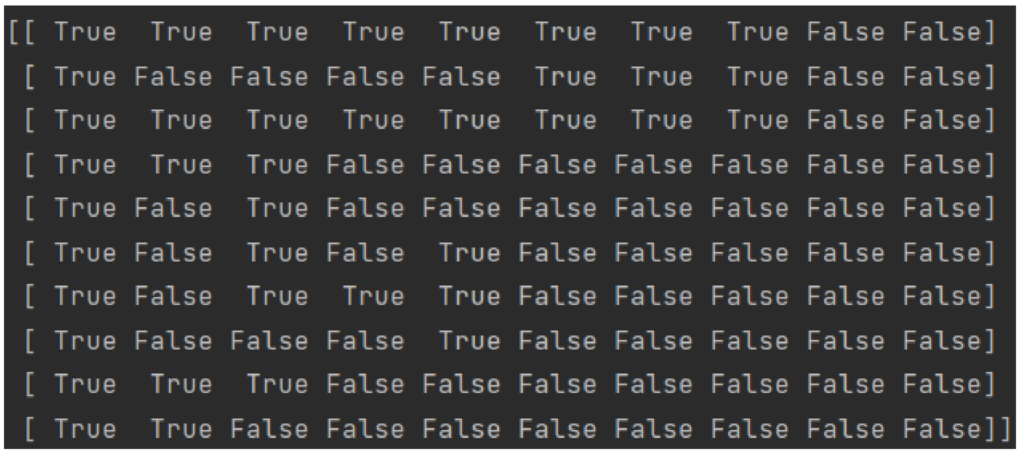


```
[[ True  True  True  True  True  True  True  True False False]
 [ True False False False False  True  True  True False False]
 [ True  True  True  True  True  True  True  True False False]
 [ True  True  True False False False False False False False]
 [ True False  True False False False False False False False]
 [ True False  True False  True False False False False False]
 [ True False  True  True  True False False False False False]
 [ True False False False  True False False False False False]
 [ True  True  True False False False False False False False]
 [ True  True False False False False False False False False]]
```

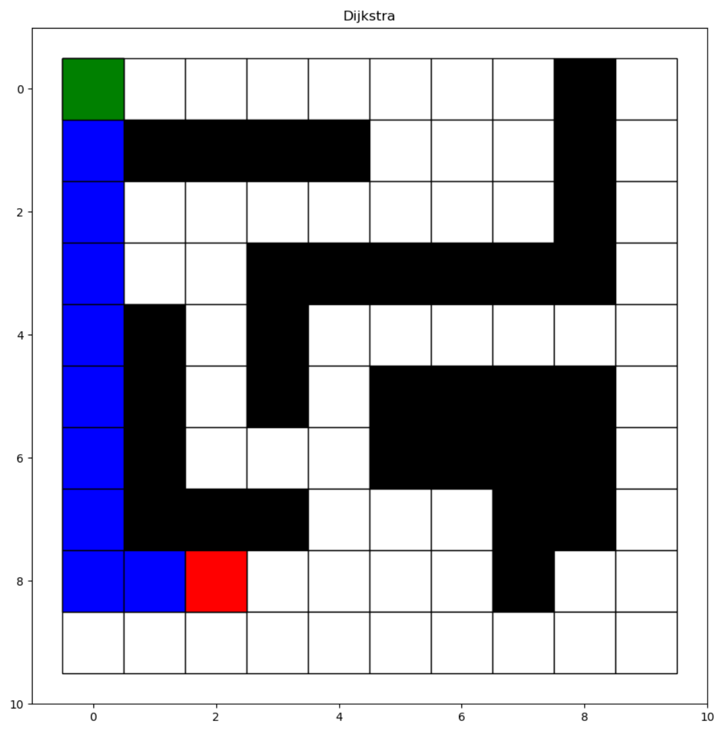
➔ Depth First Search



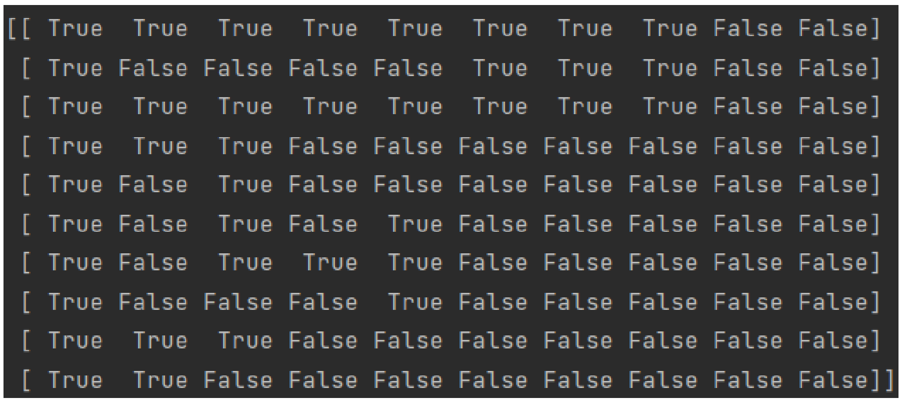
Visited nodes (True represents visited)


```
[[ True  True  True  True  True  True  True  True False False]
 [ True False False False False  True  True  True False False]
 [ True  True  True  True  True  True  True  True False False]
 [ True  True  True False False False False False False False]
 [ True False  True False False False False False False False]
 [ True False  True False  True False False False False False]
 [ True False  True  True  True False False False False False]
 [ True False False False  True False False False False False]
 [ True  True  True False False False False False False False]
 [ True  True False False False False False False False False]]
```

➔ Dijkstra's Algorithm

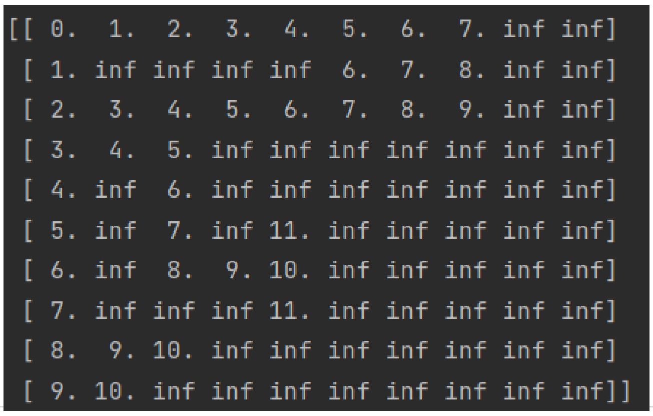


Visited nodes (True represents visited)



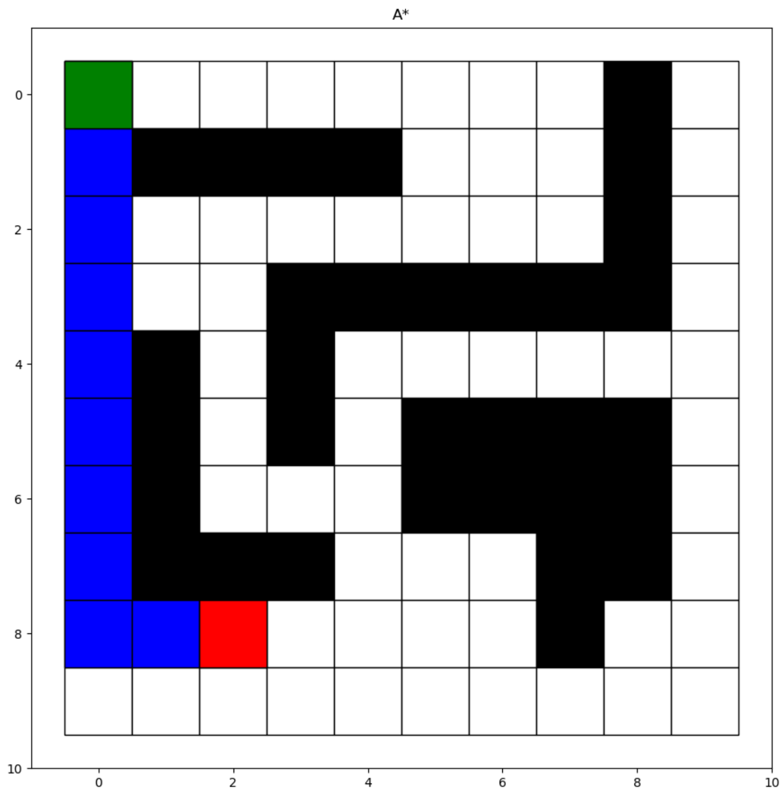
```
[[ True  True  True  True  True  True  True  True False False]
 [ True False False False False  True  True  True False False]
 [ True  True  True  True  True  True  True  True False False]
 [ True  True  True False False False False False False False]
 [ True False  True False False False False False False False]
 [ True False  True False  True False False False False False]
 [ True False  True  True  True False False False False False]
 [ True False False False  True False False False False False]
 [ True  True  True False False False False False False False]
 [ True  True False False False False False False False False]]
```

Cost of travel = cost-to-come

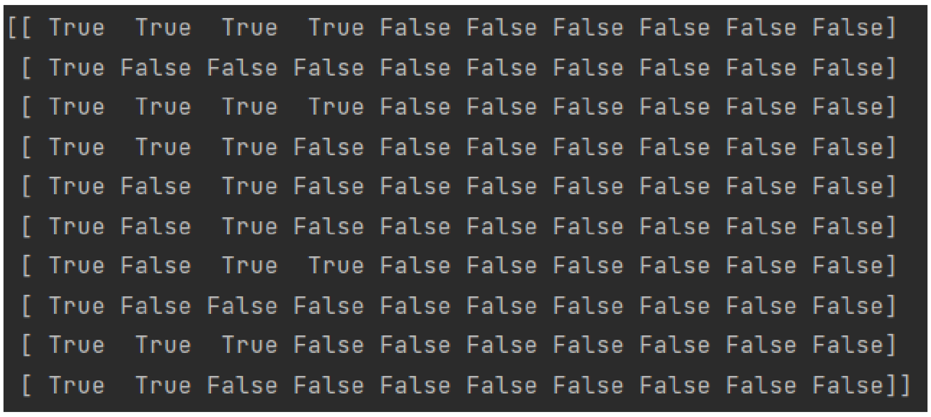


```
[[ 0.  1.  2.  3.  4.  5.  6.  7. inf inf]
 [ 1. inf inf inf inf  6.  7.  8. inf inf]
 [ 2.  3.  4.  5.  6.  7.  8.  9. inf inf]
 [ 3.  4.  5. inf inf inf inf inf inf inf]
 [ 4. inf  6. inf inf inf inf inf inf inf]
 [ 5. inf  7. inf 11. inf inf inf inf inf]
 [ 6. inf  8.  9. 10. inf inf inf inf inf]
 [ 7. inf inf inf 11. inf inf inf inf inf]
 [ 8.  9. 10. inf inf inf inf inf inf inf]
 [ 9. 10. inf inf inf inf inf inf inf inf]]
```

➔ A* Algorithm

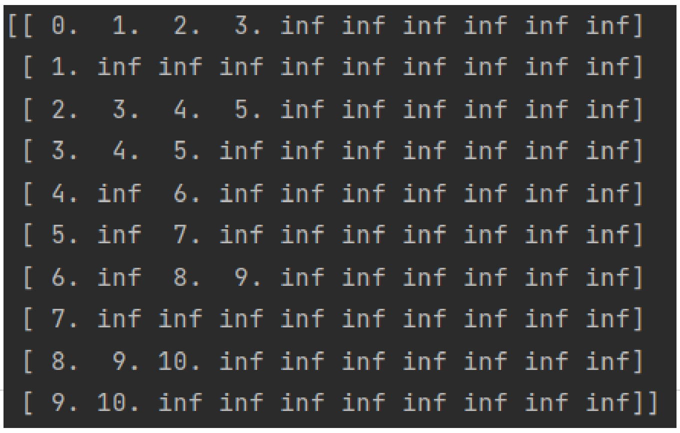


Visited nodes (True represents visited)



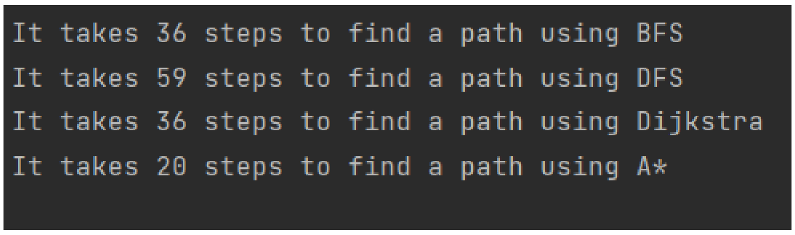
```
[[ True  True  True  True False False False False False False]
 [ True False False False False False False False False False]
 [ True  True  True  True False False False False False False]
 [ True  True  True False False False False False False False]
 [ True False  True False False False False False False False]
 [ True False  True False False False False False False False]
 [ True False  True  True False False False False False False]
 [ True False False False False False False False False False]
 [ True  True  True False False False False False False False]
 [ True  True False False False False False False False False]]
```

Cost of travel = cost-to-come + Manhattan Distance



```
[[ 0.  1.  2.  3. inf inf inf inf inf inf]
 [ 1. inf inf inf inf inf inf inf inf inf]
 [ 2.  3.  4.  5. inf inf inf inf inf inf]
 [ 3.  4.  5. inf inf inf inf inf inf inf]
 [ 4. inf  6. inf inf inf inf inf inf inf]
 [ 5. inf  7. inf inf inf inf inf inf inf]
 [ 6. inf  8.  9. inf inf inf inf inf inf]
 [ 7. inf inf inf inf inf inf inf inf inf]
 [ 8.  9. 10. inf inf inf inf inf inf inf]
 [ 9. 10. inf inf inf inf inf inf inf inf]]
```

The number of steps taken by each algorithm for this Experiment



```
It takes 36 steps to find a path using BFS
It takes 59 steps to find a path using DFS
It takes 36 steps to find a path using Dijkstra
It takes 20 steps to find a path using A*
```

We can draw similar conclusions in this experiment as we did in the experiment 1. One interesting observation to note is that the DFS algorithm performs considerably poorly on this experiment as it starts exploring the nodes deep and deep and does not find the solution until it searches all the nodes on the right side of the graph.

And we can also observe that A* performs very well in such scenarios. Therefore we can conclude that if we have admissible heuristic, A* algorithm can give optimal solution quickly and hence can be very useful.

References:

[1] Breadth-first search algorithm available at: Breadth-first search - Wikipedia

[2] Depth-first search algorithm available at: Depth-first search - Wikipedia

[3] Dijkstra’s Algorithm available at: Dijkstra's algorithm - Wikipedia

[4] A* Algorithm available at: A* search algorithm - Wikipedia

[5] Code inspiration - Shortest distance between two cells in a matrix or grid - GeeksforGeeks

[6] Code inspiration - Check for possible path in 2D matrix - GeeksforGeeks

[7] python class - Corey Schafer - YouTube

End Of Report