Name: Muhammad Ahmed Javed

Student ID: 103171624

Instructions:

The given program operates DOS command-line interface which can be brought up in Windows 10 by typing cmd into the search box at the start button. Below is how to run the given program from DOS command line:

```
> RobotNavigation <filename> <method>
```

For instance: RobotNavigation RobotNav-test .txt DFS would give the result for depth first search.

Introduction:

Robot route issues as displayed in Figure 1 can be summed up into four classifications which are 1) limitation, 2) way arranging and 3) movement control and 4) mental planning. Among these issues, it very well may be contended that way arranging is the main issue in the route cycle. Way arranging empowers the choice and recognizable proof of an appropriate way for the robot to cross in the work area region. The two fundamental parts for worldwide or deliberative way arranging are 1) robot portrayal of the world in arrangement space (C-space) and 2) the calculation execution. These two parts are interrelated and enormously impact each other in the process to decide an ideal way for the robot to navigate in the work area inside an ideal time.

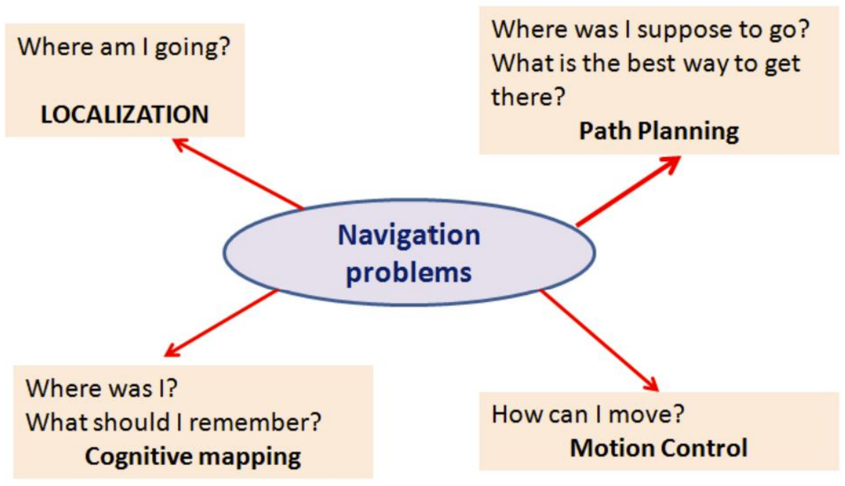


Fig. 1 Robot navigation problems.

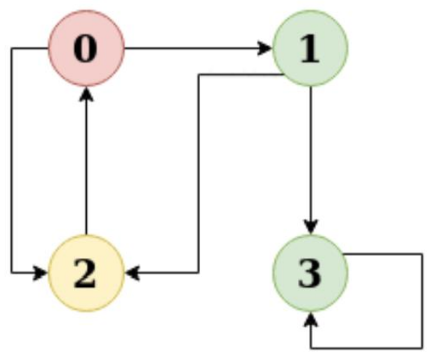
Searching Algorithms:

1. **Depth-First Search**

Depth-first search is an algorithm for traversing or searching tree or graph data structures. The algorithm starts at the root node (selecting some arbitrary node as the root node in the case of a graph) and explores as far as possible along each branch before backtracking. So the basic idea is to start from the root or any arbitrary node and mark the node and move to the adjacent unmarked node and continue this loop until there is no unmarked adjacent node. Then backtrack and check for other unmarked nodes and traverse them. Finally, print the nodes in the path.

Create a recursive function that takes the index of the node and a visited array.

- Mark the current node as visited and print the node.
- Traverse all the adjacent and unmarked nodes and call the recursive function with the index of the adjacent node.



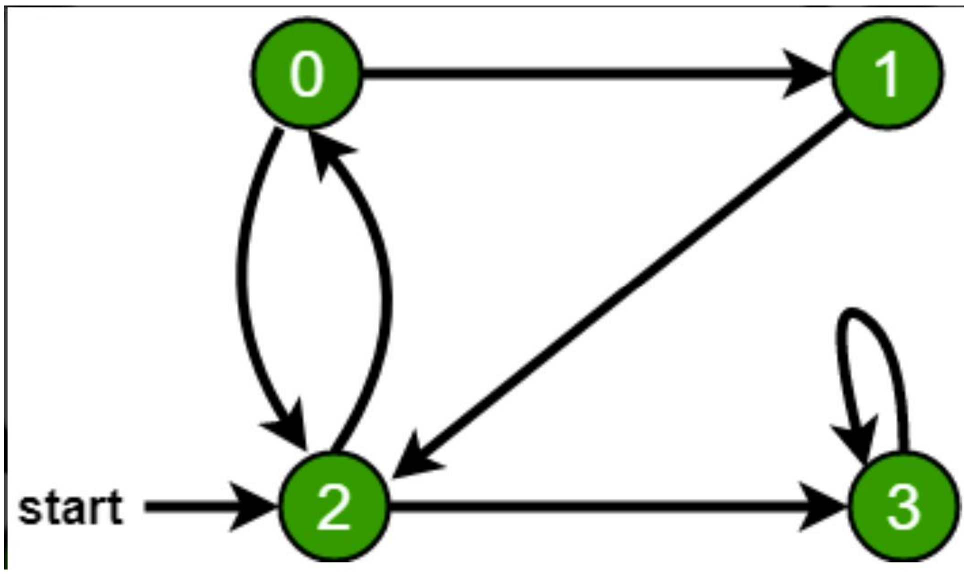
Green is unvisited node.
Red is current node.
Orange is the nodes in the recursion stack.

---

2. **Breadth-First Search**

Breadth-First Traversal (or Search) for a graph is similar to Breadth-First Traversal of a tree. The only catch here is, unlike trees, graphs may contain cycles, so we may come to the same node again. To avoid processing a node more than once, we use a Boolean visited array. For simplicity, it is assumed that all vertices are reachable from the starting vertex.

For example, in the following graph, we start traversal from vertex 2. When we come to vertex 0, we look for all adjacent vertices of it. 2 is also an adjacent vertex of 0. If we don’t mark visited vertices, then 2 will be processed again and it will become a non-terminating process. A Breadth-First Traversal of the following graph is 2, 0, 3, 1.



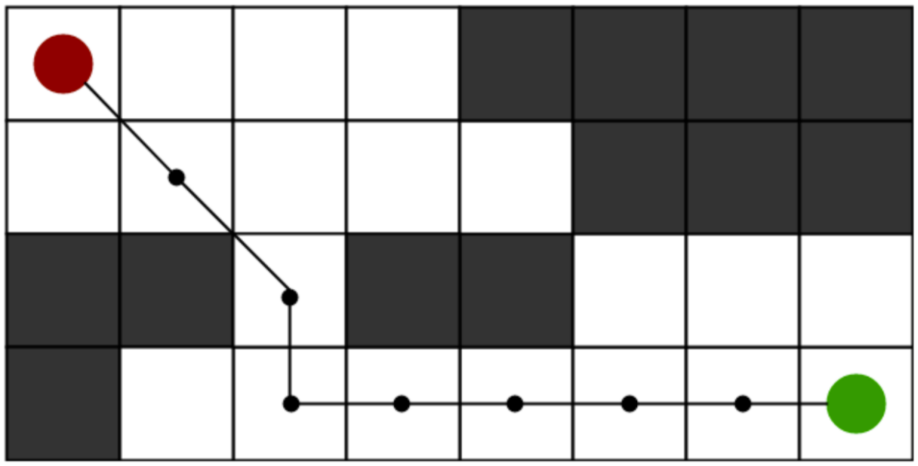
3. Greedy-Best-First Search

In BFS and DFS, when we are at a node, we can consider any of the adjacent as next node. So both BFS and DFS blindly explore paths without considering any cost function. The idea of Best First Search is to use an evaluation function to decide which adjacent is most promising and then explore. Best First Search falls under the category of Heuristic Search or Informed Search. We use a priority queue or heap to store costs of nodes which have lowest evaluation function value. So the implementation is a variation of BFS, we just need to change Queue to PriorityQueue.

- Worst case time complexity for Best First Search is O(n * Log n) where n is number of nodes. In worst case, we may have to visit all nodes before we reach goal. Note that priority queue is implemented using Min(or Max) Heap, and insert and remove operations take O(log n) time.
- Performance of the algorithm depends on how well the cost or evaluation function is designed.

4. A* Search

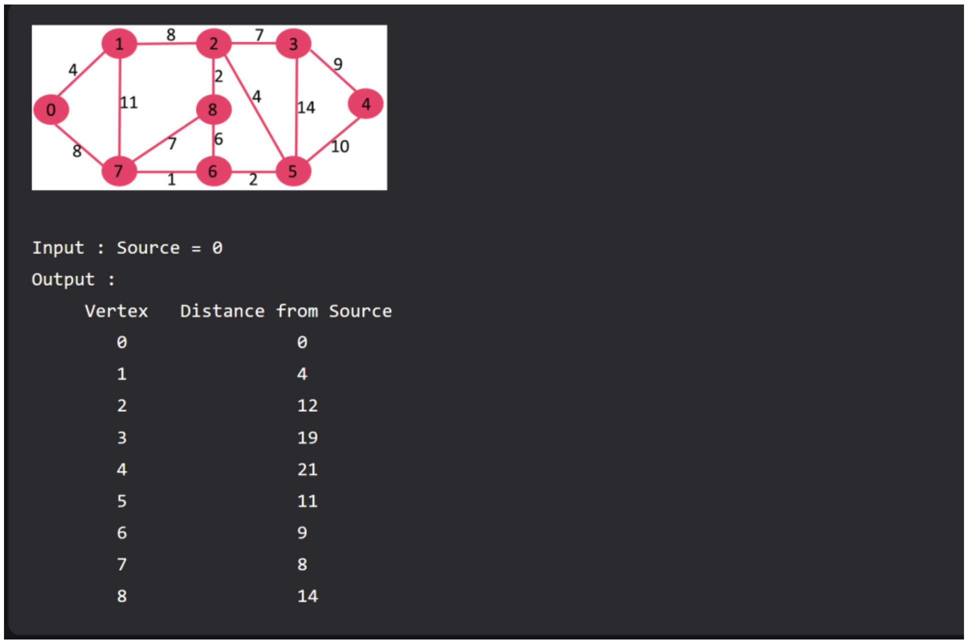
To approximate the shortest path in real-life situations, like- in maps, games where there can be many hindrances.
We can consider a 2D Grid having several obstacles and we start from a source cell (colored red below) to reach towards a goal cell (colored green below)



A* Search algorithm is one of the best and popular technique used in path-finding and graph traversals. Informally speaking, A* Search algorithms, unlike other traversal techniques, it has “brains”. What it means is that it is really a smart algorithm which separates it from the other conventional algorithms. This fact is cleared in detail in below sections. And it is also worth mentioning that many games and web-based maps use this algorithm to find the shortest path very efficiently (approximation).

Although being the best path finding algorithm around, A* Search Algorithm doesn’t produce the shortest path always, as it relies heavily on heuristics / approximations to calculate – h.

5. **Dijkstra's Search**



```
Input : Source = 0
Output :
     Vertex   Distance from Source
        0                 0
        1                 4
        2                 12
        3                 19
        4                 21
        5                 11
        6                 9
        7                 8
        8                 14
```

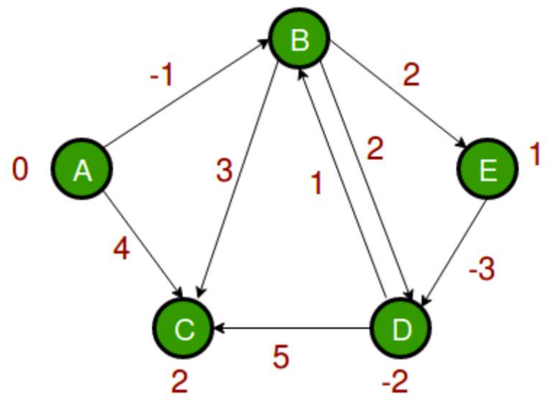
Given a graph and a source vertex in the graph, find the shortest paths from the source to all vertices in the given graph.
Dijkstra's algorithm is very similar to Prim's algorithm for minimum spanning tree. Like Prim's MST, we generate a *SPT (shortest path tree)* with a given source as a root. We maintain two sets, one set contains vertices included in the shortest-path tree, other set includes vertices not yet included in the shortest-path tree. At every step of the algorithm, we find a vertex that is in the other set (set of not yet included) and has a minimum distance from the source.

1)The code calculates the shortest distance but doesn't calculate the path information. We can create a parent array, update the parent array when distance is updated (like prim's implementation) and use it to show the shortest path from source to different vertices.
2) The code is for undirected graphs, the same Dijkstra function can be used for directed graphs also.
3) The code finds the shortest distances from the source to all vertices. If we are interested only in the shortest distance from the source to a single target, we can break the for loop when the picked minimum distance vertex is equal to the target (Step 3.a of the algorithm).
4) Time Complexity of the implementation is O(V^2). If the input graph is represented using adjacency list, it can be reduced to O(E log V) with the help of a binary heap. Please see Dijkstra's Algorithm for Adjacency List Representation for more details.
5) Dijkstra's algorithm doesn't work for graphs with negative weight cycles. It may give correct results for a graph with negative edges but you must allow a vertex can be visited multiple times and that version will lose its fast time complexity. For graphs with negative weight edges and cycles, Bellman–Ford algorithm can be used, we will soon be discussing it as a separate post.

6. Bellman-Ford Algorithm

Given a graph and a source vertex *src* in graph, find shortest paths from *src* to all vertices in the given graph. The graph may contain negative weight edges.
We have discussed Dijkstra's algorithm for this problem. Dijkstra's algorithm is a Greedy algorithm and time complexity is O((V+E)LogV) (with the use of Fibonacci heap). *Dijkstra doesn't work for Graphs with negative weights, Bellman-Ford works for such graphs. Bellman-Ford is also simpler than Dijkstra and suites well for distributed systems. But time complexity of Bellman-Ford is O(VE), which is more than Dijkstra.*

A	B	C	D	E
0	∞	∞	∞	∞
0	-1	∞	∞	∞
0	-1	4	∞	∞
0	-1	2	∞	∞
0	-1	2	∞	1
0	-1	2	1	1
0	-1	2	-2	1



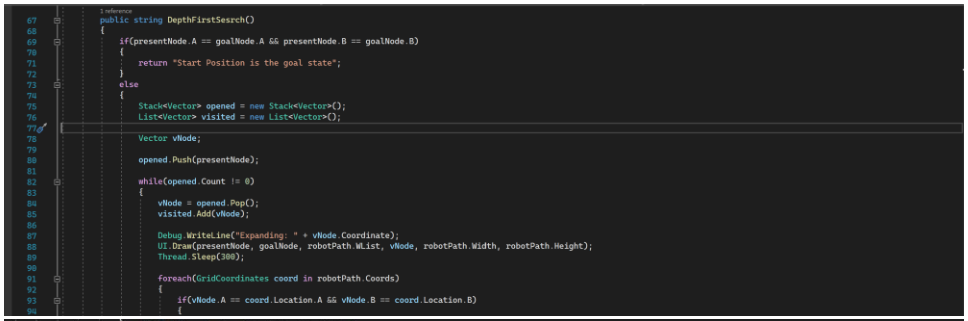
The first iteration guarantees to give all shortest paths which are at most 1 edge long. We get the following distances when all edges are processed second time (The last row shows final values). The second iteration guarantees to give all shortest paths which are at most 2 edges long. The algorithm processes all edges 2 more times. The distances are minimized after the second iteration, so third and fourth iterations don't update the distances.

1) Negative weights are found in various applications of graphs. For example, instead of paying cost for a path, we may get some advantage if we follow the path.
2) Bellman-Ford works better (better than Dijkstra's) for distributed systems. Unlike Dijkstra's where we need to find the minimum value of all vertices, in Bellman-Ford, edges are considered one by one.
3) Bellman-Ford does not work with undirected graph with negative edges as it will declared as negative cycle.

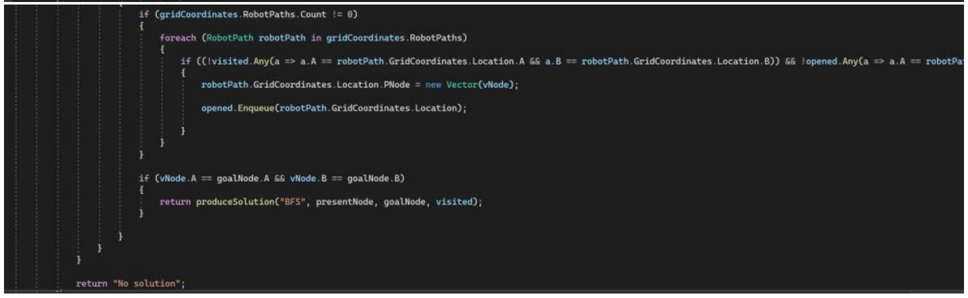
The standard Bellman-Ford algorithm reports the shortest path only if there are no negative weight cycles. Modify it so that it reports minimum distances even if there is a negative weight cycle.

Implementation:

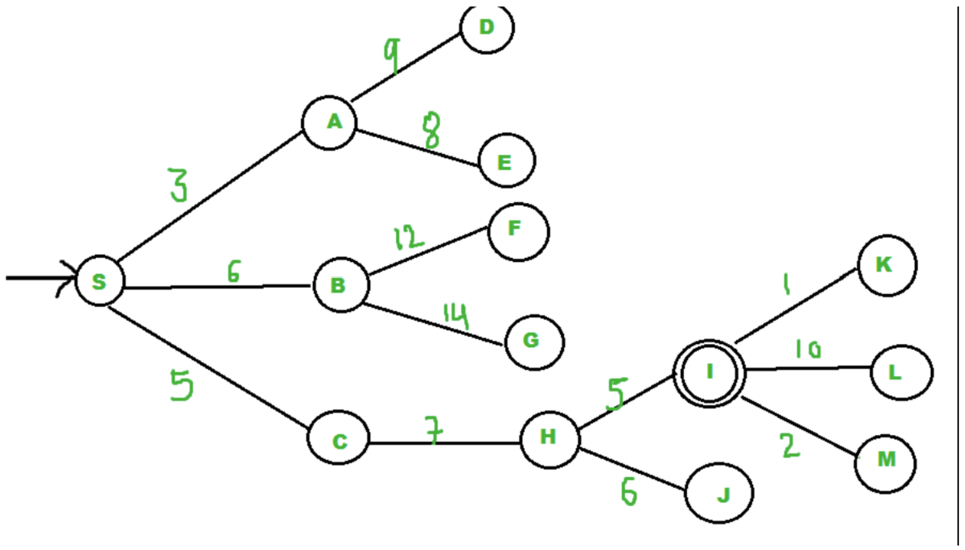
1. **Depth-First Search**



```
1 reference
public string DepthFirstSesrch()
{
    if(presentNode.A == goalNode.A && presentNode.B == goalNode.B)
    {
        return "Start Position is the goal state";
    }
    else
    {
        Stack<Vector> opened = new Stack<Vector>();
        List<Vector> visited = new List<Vector>();

        Vector vNode;

        opened.Push(presentNode);

        while(opened.Count != 0)
        {
            vNode = opened.Pop();
            visited.Add(vNode);

            Debug.WriteLine("Expanding: " + vNode.Coordinate);
            UI.Draw(presentNode, goalNode, robotPath.WList, vNode, robotPath.Width, robotPath.Height);
            Thread.Sleep(300);

            foreach(GridCoordinates coord in robotPath.Coords)
            {
                if(vNode.A == coord.Location.A && vNode.B == coord.Location.B)
                {
```



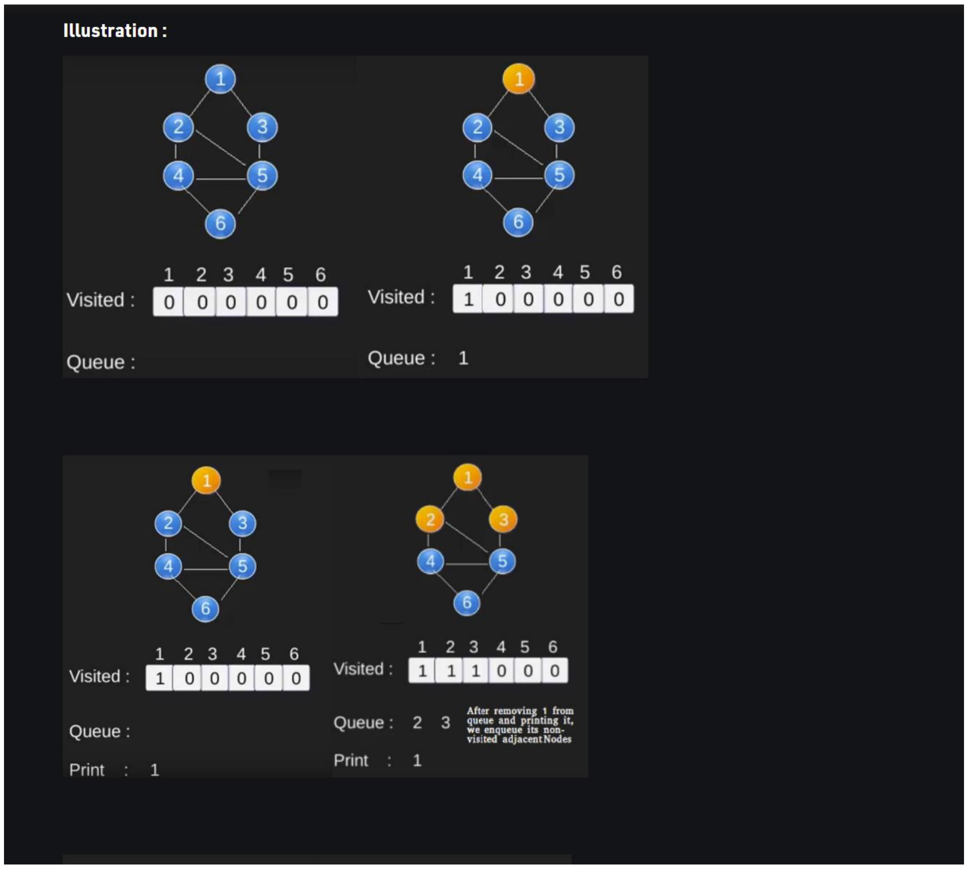
```
        if (gridCoordinates.RobotPaths.Count != 0)
        {
            foreach (RobotPath robotPath in gridCoordinates.RobotPaths)
            {
                if ((!visited.Any(a => a.A == robotPath.GridCoordinates.Location.A && a.B == robotPath.GridCoordinates.Location.B)) && !opened.Any(a => a.A == robotPa
                {
                    robotPath.GridCoordinates.Location.PNode = new Vector(vNode);

                    opened.Enqueue(robotPath.GridCoordinates.Location);

                }
            }
        }

        if (vNode.A == goalNode.A && vNode.B == goalNode.B)
        {
            return produceSolution("BFS", presentNode, goalNode, visited);
        }

    }
    }
}

return "No solution";
```

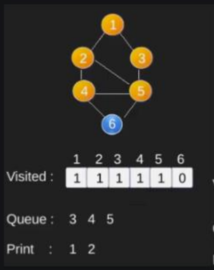


- Create a recursive function that takes the index of the node and a visited array.
- Mark the current node as visited and print the node.
- Traverse all the adjacent and unmarked nodes and call the recursive function with the index of the adjacent node.
- Run a loop from 0 to the number of vertices and check if the node is unvisited in the previous DFS, call the recursive function with the current node.
- **Time complexity:** O(V + E), where V is the number of vertices and E is the number of edges in the graph.
- **Space Complexity:** O(V), since an extra visited array of size V is required.

2. **Breadth-First Search**

Time Complexity: O(V+E) where V is a number of vertices in the graph and E is a number of edges in the graph.



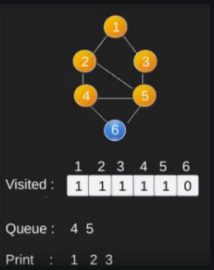


Visited :

1	2	3	4	5	6
1	1	1	1	1	0

Queue : 3 4 5

Print : 1 2

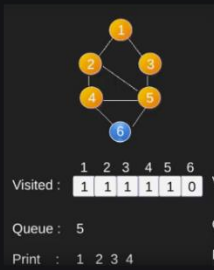


Visited :

1	2	3	4	5	6
1	1	1	1	1	0

Queue : 4 5

Print : 1 2 3

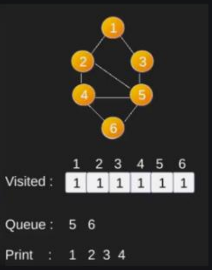


Visited :

1	2	3	4	5	6
1	1	1	1	1	0

Queue : 5

Print : 1 2 3 4

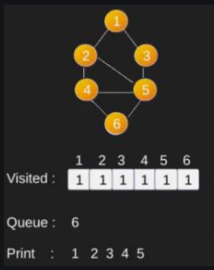


Visited :

1	2	3	4	5	6
1	1	1	1	1	1

Queue : 5 6

Print : 1 2 3 4

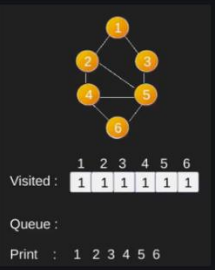


Visited :

1	2	3	4	5	6
1	1	1	1	1	1

Queue : 6

Print : 1 2 3 4 5

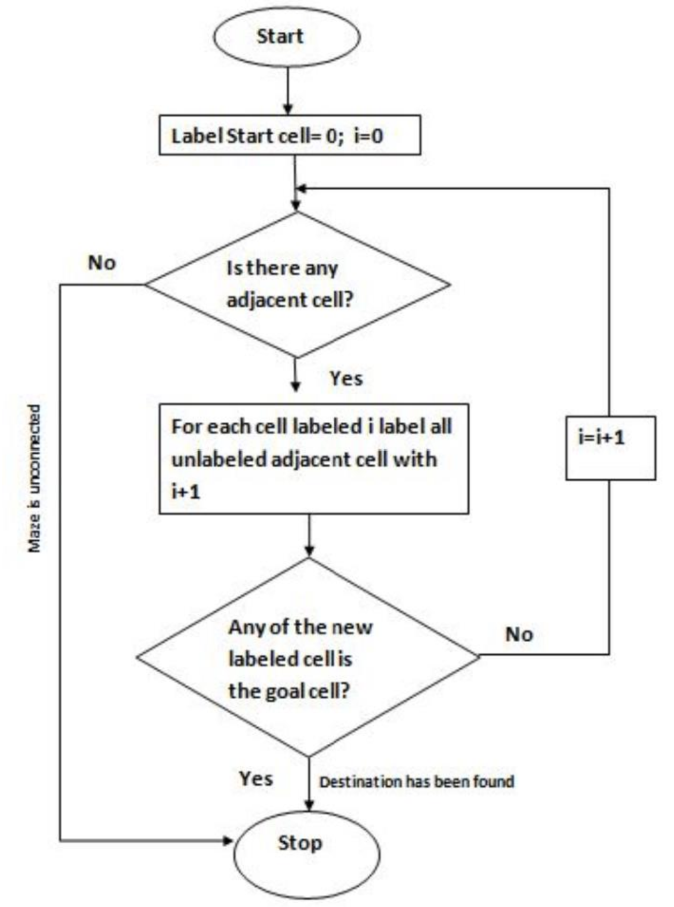


Visited :

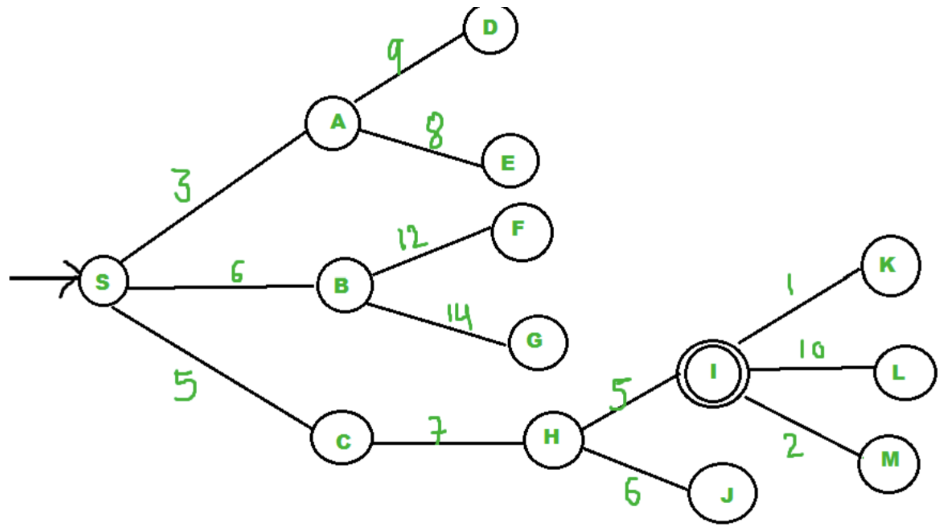
1	2	3	4	5	6
1	1	1	1	1	1

Queue :

Print : 1 2 3 4 5 6


Start
Label Start cell= 0; i=0
Is there any adjacent cell?
No
Yes
For each cell labeled i label all unlabeled adjacent cell with i+1
i=i+1
Any of the new labeled cell is the goal cell?
No
Yes
Destination has been found
Maze is unconnected
Stop

3. Greedy-Best-First Search



```
First search.

pq initially contains S
We remove s from and process unvisited
neighbors of S to pq.
pq now contains {A, C, B} (C is put
before B because C has lesser cost)

We remove A from pq and process unvisited
neighbors of A to pq.
pq now contains {C, B, E, D}

We remove C from pq and process unvisited
neighbors of C to pq.
pq now contains {B, H, E, D}

We remove B from pq and process unvisited
neighbors of B to pq.
pq now contains {H, E, D, F, G}

We remove H from pq.  Since our goal
"I" is a neighbor of H, we return.
```

4. **A* Search**

```
 / A* Search Algorithm
1.  Initialize the open list
2.  Initialize the closed list
    put the starting node on the open
    list (you can leave its f at zero)

3.  while the open list is not empty
    a) find the node with the least f on
       the open list, call it "q"

    b) pop q off the open list

    c) generate q's 8 successors and set their
       parents to q

    d) for each successor
        i) if successor is the goal, stop search
```

```
         ii) else, compute both g and h for successor
           successor.g = q.g + distance between
                               successor and q
           successor.h = distance from goal to
           successor (This can be done using many
           ways, we will discuss three heuristics-
           Manhattan, Diagonal and Euclidean
           Heuristics)

           successor.f = successor.g + successor.h

         iii) if a node with the same position as
             successor is in the OPEN list which has a
            lower f than successor, skip this successor

         iV) if a node with the same position as
             successor  is in the CLOSED list which has
             a lower f than successor, skip this successor
             otherwise, add  the node to the open list
      end (for loop)

     e) push q on the closed list
     end (while loop)
```

5. **Dijkistra's Search**

Algorithm
1) Create a set *sptSet* (shortest path tree set) that keeps track of vertices included in the shortest-path tree, i.e., whose minimum distance from the source is calculated and finalized. Initially, this set is empty.
2) Assign a distance value to all vertices in the input graph. Initialize all distance values as INFINITE. Assign distance value as 0 for the source vertex so that it is picked first.
3) While *sptSet* doesn't include all vertices
….**a)** Pick a vertex u which is not there in *sptSet* and has a minimum distance value.
….**b)** Include u to *sptSet*.
….**c)** Update distance value of all adjacent vertices of u. To update the distance values, iterate through all adjacent vertices. For every adjacent vertex v, if the sum of distance value of u (from source) and weight of edge u-v, is less than the distance value of v, then update the distance value of v.

6. **Bellman Ford Algorithm**

Following are the detailed steps.
Input: Graph and a source vertex *src*
Output: Shortest distance to all vertices from *src*. If there is a negative weight cycle, then shortest distances are not calculated, negative weight cycle is reported.
1) This step initializes distances from the source to all vertices as infinite and distance to the source itself as 0. Create an array dist[] of size |V| with all values as infinite except dist[src] where src is source vertex.
2) This step calculates shortest distances. Do following |V|-1 times where |V| is the number of vertices in given graph.
…..**a)** Do following for each edge u-v
………………If dist[v] > dist[u] + weight of edge uv, then update dist[v]
………………….dist[v] = dist[u] + weight of edge uv
3) This step reports if there is a negative weight cycle in graph. Do following for each edge u-v
……If dist[v] > dist[u] + weight of edge uv, then "Graph contains negative weight cycle"
The idea of step 3 is, step 2 guarantees the shortest distances if the graph doesn't contain a negative weight cycle. If we iterate through all edges one more time and get a shorter path for any vertex, then there is a negative weight cycle
How does this work? Like other Dynamic Programming Problems, the algorithm calculates shortest paths in a bottom-up manner. It first calculates the shortest distances which have at-most one edge in the path. Then, it calculates the shortest paths with at-most 2 edges, and so on. After the i-th iteration of the outer loop, the shortest paths with at most i edges are calculated. There can be maximum |V| – 1 edges in any simple path, that is why the outer loop runs |v| – 1 times. The idea is, assuming that there is no negative weight cycle, if we have calculated shortest paths with at most i edges, then an iteration over all edges guarantees to give shortest path with at-most (i+1) edges (Proof is simple, you can refer this or MIT Video Lecture)

Bugs/Missing:

Custom Search methods are missing. One of them is not working properly.

Research:

Conclusion:

A* search is optimal if the heuristic is admissible. Admissible makes that whichever node you expand, it makes sure that the current estimate is always smaller than the optimal, so path about to expand maintains a chance to find the optimal path.

Acknowledgement:

https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/

https://www.geeksforgeeks.org/depth-first-search-or-dfs-for-a-graph/

https://www.geeksforgeeks.org/best-first-search-informed-search/

https://www.geeksforgeeks.org/a-search-algorithm/

These links helped me explain the search and learn methods to apply this code in my project

Referneces:

Dijkstra’s shortest path algorithm | GeeksforGeeks

http://theory.stanford.edu/~amitp/GameProgramming/

https://www.geeksforgeeks.org/greedy-algorithms-set-7-dijkstras-algorithm-for-adjacency-list-representation/