Omar Nour Eldin Mohamed Khalil

Table of Contents

First: Project 1

Requirements: This project is for AI applications. You are free to choose any problem that interests you, that can be solved by suitable AI search techniques.

Chosen problem: Solving traffic problems by applying Maze solving algorithms to traffic problems by offering innovative solutions to optimize traffic flow, reduce congestion and improve overall efficiency.

➔Mazes and Traffic have some parallels:

Where in a maze you are often trying to find the most efficient path from one point to another facing various obstacles and dead ends. Similarly, in traffic, drivers are going through roads, intersections, and congestion to reach their destinations, going through several challenges like:

Route Optimization: In both mazes and traffic, the goal is to find the optimal route. In a maze, this might mean identifying the shortest path to the exit, while in traffic, it involves choosing the fastest route to your destination, considering factors like distance, traffic conditions, and road closures.

Obstacles and Dead Ends*:* Mazes are often designed with dead ends and obstacles that force you to backtrack or find alternate routes. Similarly, traffic congestion, accidents, road closures, and construction zones can act as obstacles or dead ends, requiring drivers to reroute and find alternative paths.

Decision Making: Both situations require quick decision-making. In a maze, you must decide which direction to take at each intersection, while in traffic, you must make split-second decisions about lane changes, mergers, and exits.

Efficiency vs. Congestion: In a maze, taking the most direct path may not always be the fastest if it leads to dead ends or blocked passages. Similarly, in traffic, choosing the most direct route may not be the quickest if it's congested or if there are accidents. Drivers often need to balance the desire for efficiency with the reality of traffic conditions.

Optimal Flow: Just as a well-designed maze facilitates smooth navigation, well-designed road systems and traffic management strategies aim to promote optimal traffic flow. This might involve synchronized traffic lights, lane markings, signage, and infrastructure improvements to minimize congestion and delays.

Adaptability: Both maze-solving and driving in traffic require adaptability. As conditions change, whether it's a shifting maze layout or fluctuating traffic patterns, individuals must adjust their strategies accordingly to reach their goals efficiently.

Navigation Skills: Successfully navigating through a maze and through traffic both require good spatial awareness, understanding of directions, and the ability to interpret signage or signals.

1) Problem formulation as a search problem:

Formulating a 2D maze as a search problem involves defining the components required to navigate from a starting point to a goal while avoiding obstacles. Here's how we can break down the problem:

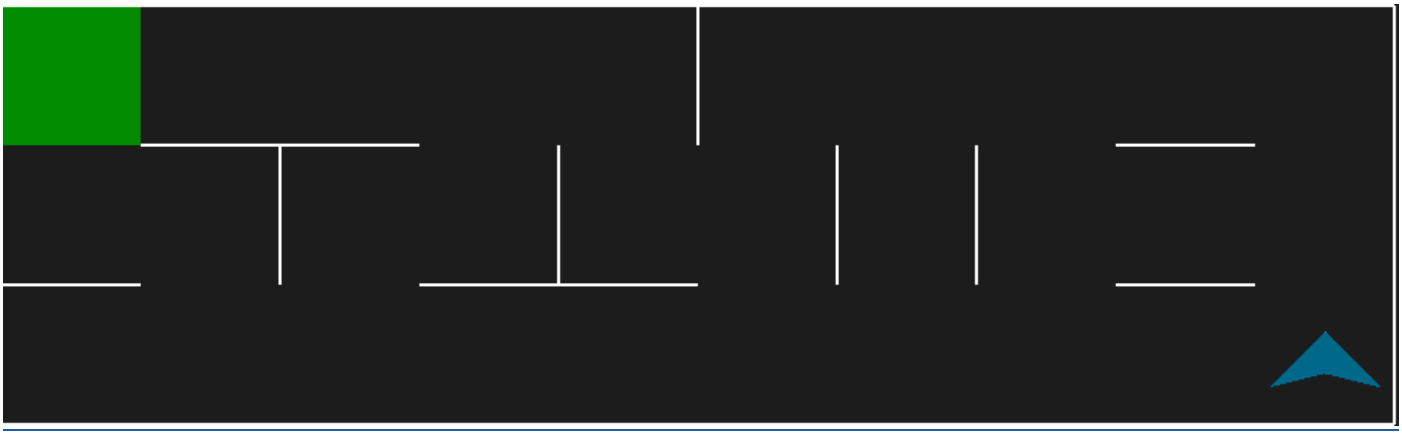

Figure shows an example of the implemented maze environment of 3 rows and 10 columns

State Space: The state space represents all possible configurations of the maze. Each state corresponds to a specific location within the maze. In a 2D maze, each state can be represented by its coordinates (x, y) indicating the position of the agent within the maze.

Initial State*:* This is the starting point of the agent within the maze. It is defined by the coordinates (x_start, y_start) where the agent begins its journey.

Goal State: The goal state represents the destination or target location within the maze. It is defined by the coordinates (x_goal, y_goal) where the agent aims to reach.

Actions: Actions represent the possible moves the agent can make from one state to another. In a 2D maze, typical actions include moving up, down, left, or right. However, depending on the maze layout, some actions may not be available if blocked by obstacles.

Transition Model: The transition model defines the result of taking an action from a given state. It specifies how the agent's position changes when it performs a particular action. For example, if the agent moves up from state (x, y), the resulting state becomes (x, y+1) if the cell above is not blocked.

Cost Function: The cost function assigns a numerical cost or weight to each action taken by the agent. This can represent factors such as distance traveled, time taken, or energy expended. In a basic maze-solving problem, each action typically incurs a unit cost, implying that all moves have equal weight.

Obstacles: Obstacles represent the barriers or impassable areas within the maze. These are cells that the agent cannot traverse. The presence of obstacles affects the agent's ability to reach the goal and may require the search algorithm to find alternative paths.

2) Technical discussion (Algorithm Strategy Biased):

The techniques (algorithms) implemented to reach the goal in the maze:

➔a) BFS (Breadth First Search) Algorithm

➔b) DFS (Depth First Search) Algorithm

➔c) A* Algorithm

➔d) Wall Follower Algorithm

➔e) Dijkstra Algorithm

a) BFS (Breadth First Search):

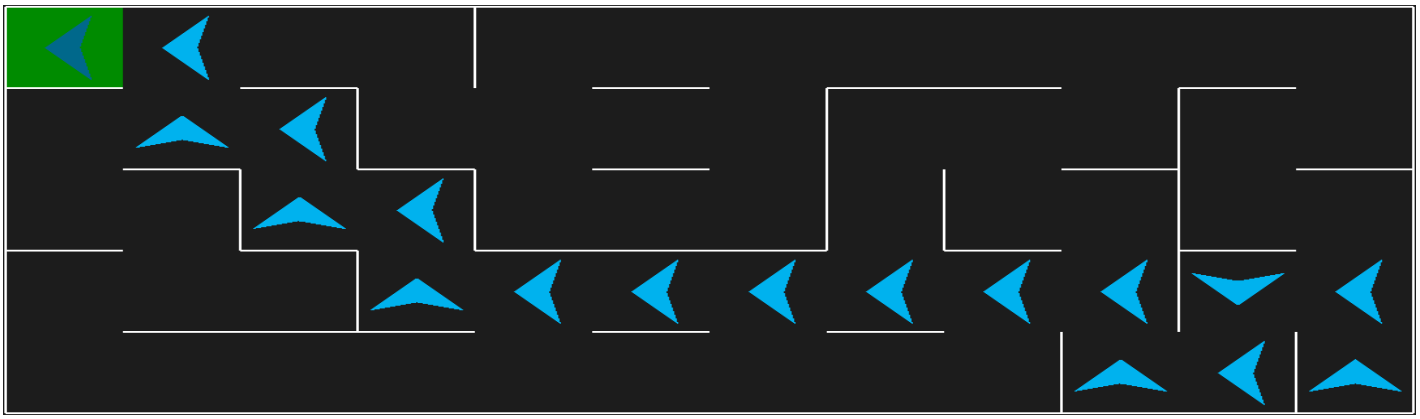

Figure shows an example of the implemented BFS agent in a maze environment of 5 rows and 12 columns

Breadth First Search (BFS) is a graph traversal algorithm that explores all the vertices in a graph at the current depth before moving on to the vertices at the next depth level. It starts at a specified vertex and visits all its neighbors before moving on to the next level of neighbors. BFS is commonly used in algorithms for pathfinding, connected components, and shortest path problems in graphs.

BFS uses **Level Order Traversal** technique which is a method to traverse a Tree such that all nodes present in the same level are traversed completely before traversing the next level. The main idea of level order traversal is to traverse all the nodes of a lower level before moving to any of the nodes of a higher level. This can be done in any of the following ways:

➔*The naive one* (finding the height of the tree and traversing each level and printing the nodes of that level)

➔*Efficiently using a queue.*

```
function BREADTH-FIRST-SEARCH(problem) returns a solution, or failure
    node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
    if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
    frontier ← a FIFO queue with node as the only element
    explored ← an empty set
    loop do
        if EMPTY?(frontier) then return failure
        node ← POP(frontier)  /* chooses the shallowest node in frontier */
        add node.STATE to explored
        for each action in problem.ACTIONS(node.STATE) do
            child ← CHILD-NODE(problem, node, action)
            if child.STATE is not in explored or frontier then
                if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
                frontier ← INSERT(child, frontier)
```

Figure shows BFS on a graph pseucode.

Breadth-first search is an instance of the general graph-search algorithm in which the shallowest unexpanded node is chosen for expansion. This is achieved very simply by using a **FIFO queue** for the frontier. Thus, new nodes (which are always deeper than their parents) go to the back of the queue, and old nodes, which are shallower than the new nodes, get expanded first. There is one slight tweak on the general graph-search algorithm, which is that the goal test is applied to each node when it is generated rather than when it is selected for expansion. This decision is explained below, where we discuss time complexity. Note also that the algorithm, following the general template for graph search, discards any new path to a state already in the frontier or explored set; it is easy to see that any such path must be at least as deep as the one already found. Thus, breadth-first search always **has the shallowest path** to every node on the frontier.

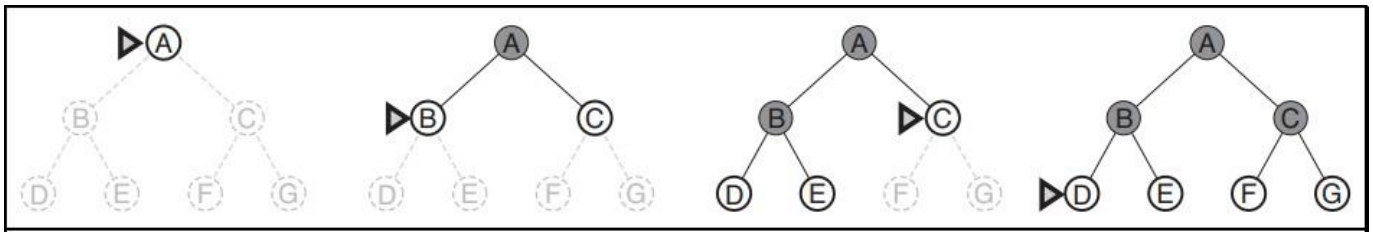

Figure shows BFS on a simple binary tree. At each stage, the node to be expanded next is indicated by the marker.

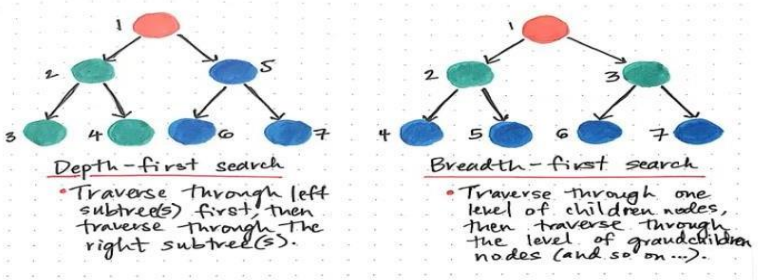

Figure shows a comparison between DFS and BFS

b) DFS (Depth First Search):

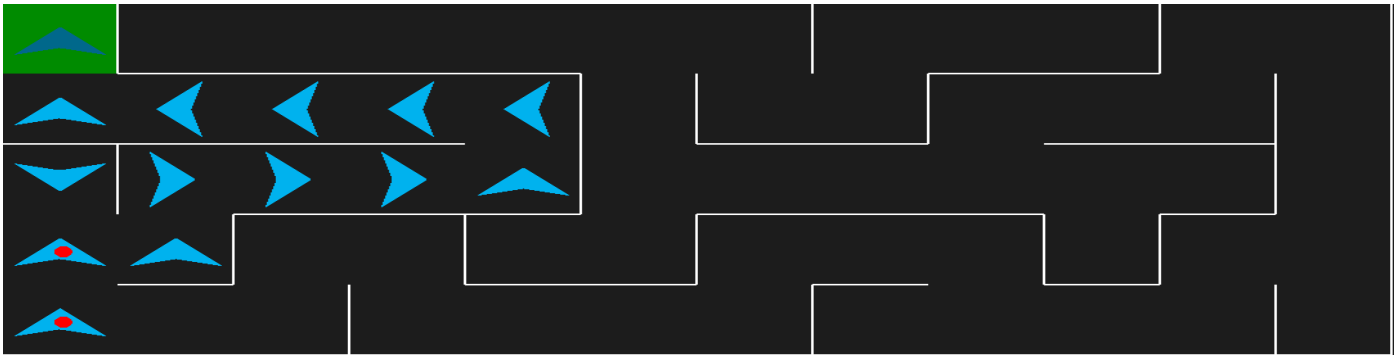

Figure shows an example of the implemented DFS agent in a maze environment of 5 rows and 12 columns

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.

The depth-first search algorithm is an instance of the graph-search algorithm, whereas breadth-first-search uses a **FIFO** queue, depth-first search uses a **LIFO** queue. A LIFO queue means that the most recently generated node is chosen for expansion. This must be the deepest unexpanded node because it is one deeper than its parent—which, in turn, was the deepest unexpanded node when it was selected.

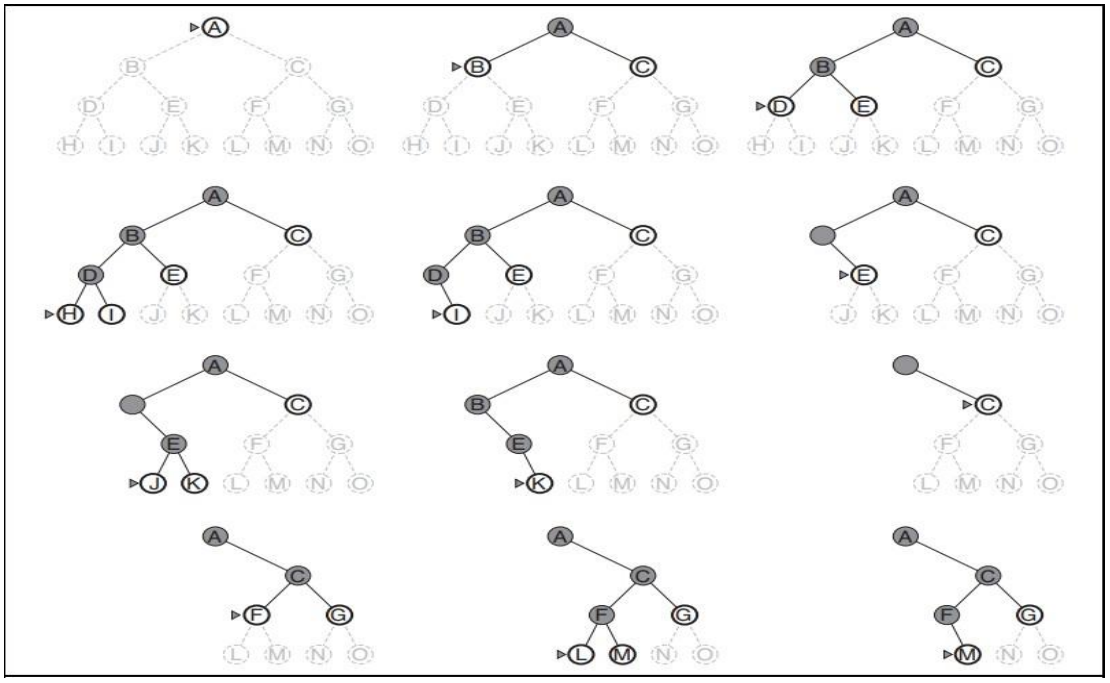

Figure shows a DFS on a binary tree, the unexplored region is shown in light gray, explored nodes with no descendants in the frontier are removed from the memory, nodes at depth 3 have no successors and M is the only goal node.

c) A* search:

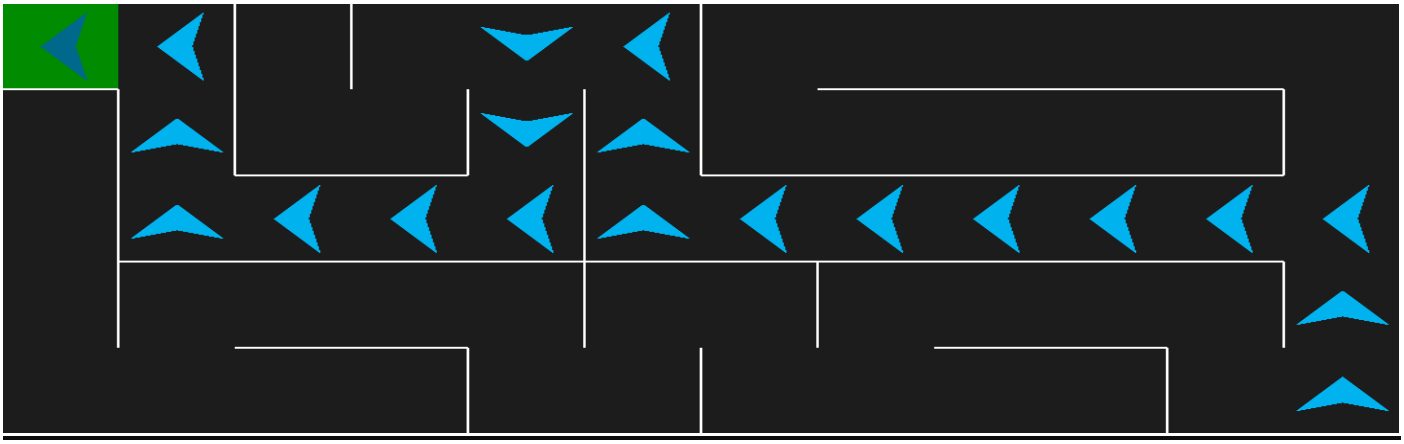

Figure shows an example of the implemented A agent in a maze environment of 5 rows and 12 columns*

A* Search algorithm is one of the best and most popular techniques used in pathfinding 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. And it is also worth mentioning that many games and web-based maps use this algorithm to find the shortest path very efficiently (approximation).

It evaluates nodes by combining g(n), the cost to reach the node, and h(n), the cost to get from the node to the goal: **f(n) = g(n) + h(n)**

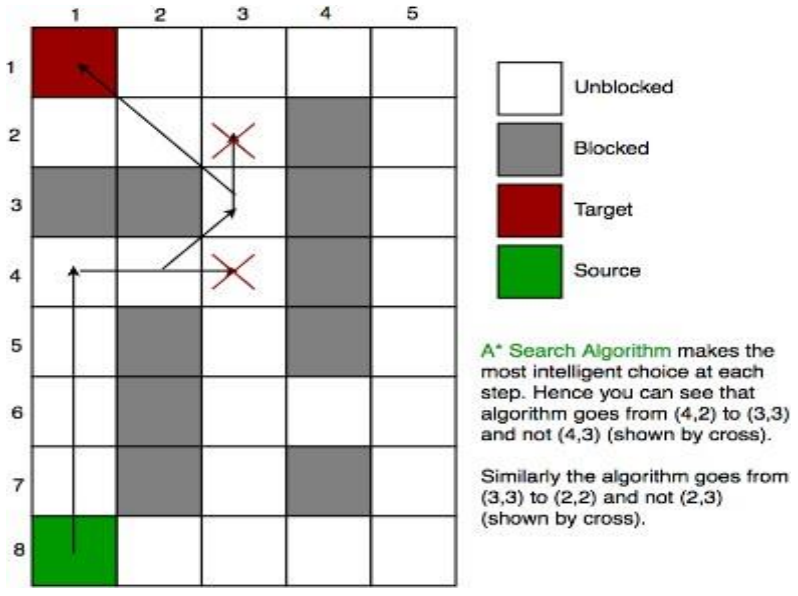

Figure shows how A work*

d) Wall Follower search:

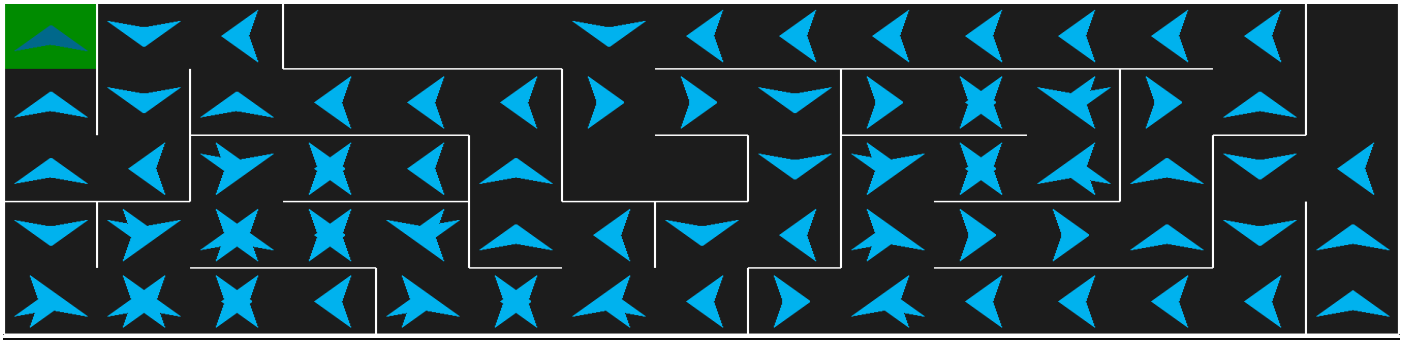

Figure shows an example of the implemented Wall Follower agent in a maze environment of 5 rows and 15 columns

One effective rule for traversing mazes is the Hand on Wall Rule, also known as either the left-hand rule or the right-hand rule. If the maze is simply connected, that is, all its walls are connected together or to the maze's outer boundary, then by keeping one hand in contact with one wall of the maze the solver is guaranteed not to get lost and will reach a different exit if there is one; otherwise, the algorithm will return to the entrance having traversed every corridor next to that connected section of walls at least once. The algorithm is a depth-first in-order tree traversal.

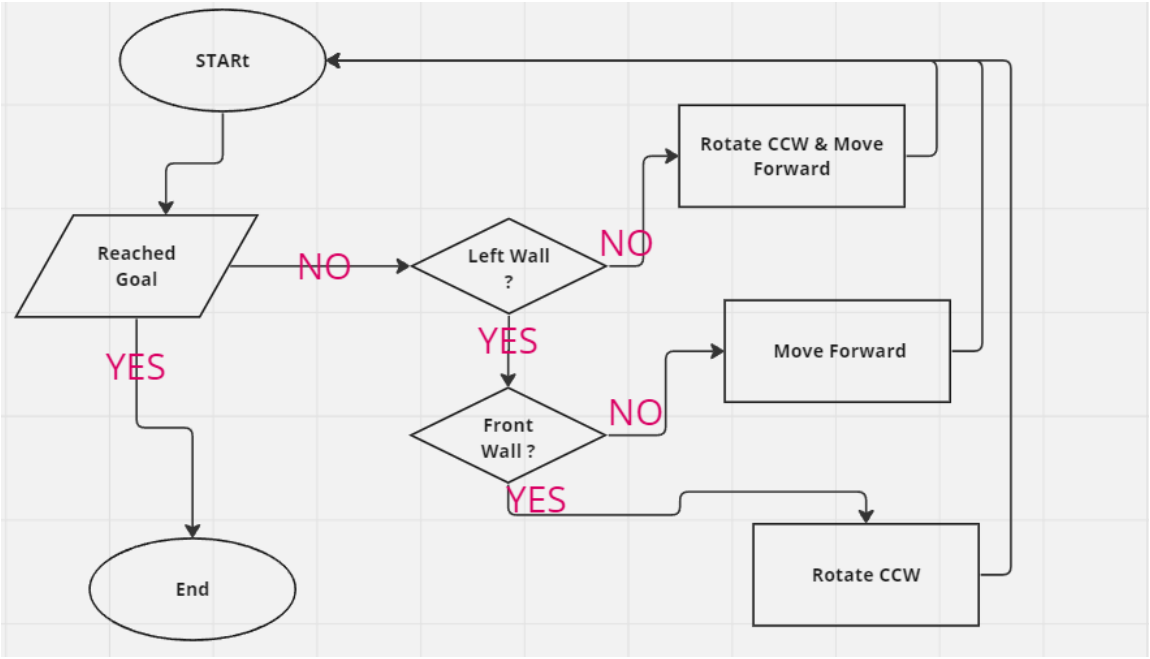

Figure shows Wall Follower Search flow chart

e) Dijkstra search:

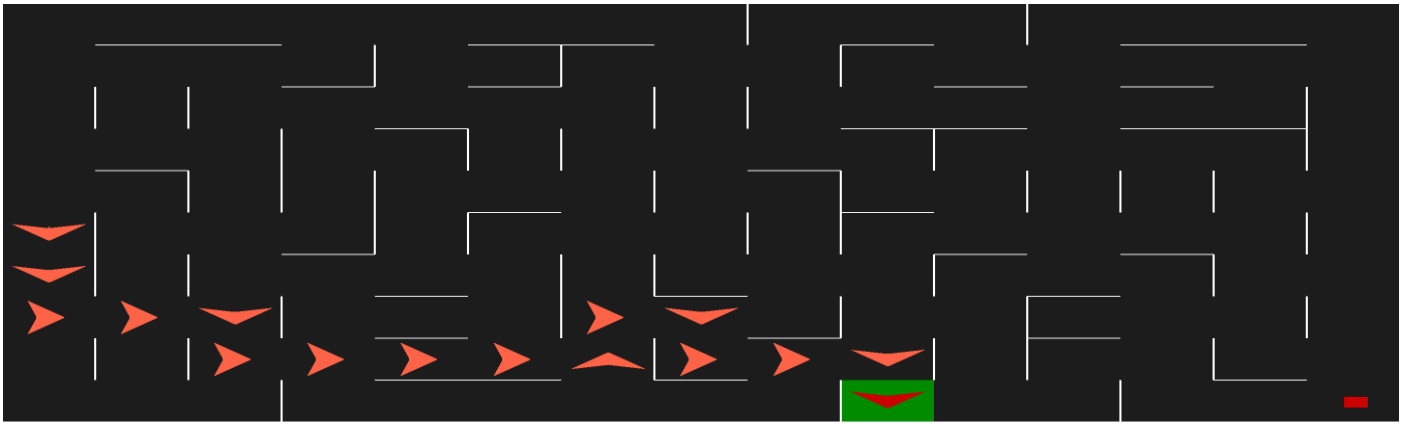

Figure shows an example of the implemented Dijkstra agent in a maze environment of 10 rows and 15 columns

The idea is to generate a SPT (shortest path tree) with a given source as a root. Maintain an Adjacency Matrix with 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, find a vertex that is in the other set (set not yet included) and has a minimum distance from the source.

Algorithm:

- 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.
- Assign a distance value to all vertices in the input graph. Initialize all distance values as **INFINITE**. Assign the distance value as 0 for the source vertex so that it is picked first.
- While **sptSet** doesn't include all vertices
  - Pick a vertex **u** that is not there in **sptSet** and has a minimum distance value.
  - Include u to **sptSet**.
  - Then update the 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 the 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**.

Figure explaining Dijkstra Algorithm

3)Discussion of results:

POC	BFS	DFS	A*	Wall Follower	Dijkstra
PROS	Guarantees finding the shortest path in terms of the number of steps.	Requires less memory compared to BFS	More efficient than BFS and DFS especially in larger search spaces	Simple to implement and does not require maintaining a search tree.	Guarantees finding the shortest path in terms of total cost, not just the number of steps.
CONS	Can require a lot of memory to store all nodes at each level of the search tree	Does not necessarily find the shortest path can get stuck in infinite loops	Requires an admissible heuristic to ensure optimality, and the quality of the solution heavily depends on the quality of the heuristic.	May not find the shortest path, can get stuck in certain mazes, and does not work well in mazes with open areas or multiple paths.	Can be slower than A* if there is no heuristic guiding the search
Search Time	O(V + E)	O(V + E)	O(V log V)	Highly dependent on maze configuration	O((V + E) log V) With priority queue:O(V^2)
Memory Complexity	O(V + E)	O(V)	O(h)	does not require maintaining a search tree, so its memory complexity is minimal. It only needs memory for storing the current position.	O(E + V)
Optimality	Yes	NO	Yes, if the heuristic is admissible	NO	Yes

V➔number of vertices, E➔number of edges

4)Results:

➔If finding the shortest path is paramount and memory resources are not a concern, **BFS or A*** with an appropriate heuristic may be the best choices.

➔If memory resources are limited and finding an optimal solution is not critical, **DFS or Wall Follower** may be suitable.

➔If the goal is to find the shortest path from the start node to all other reachable nodes, **Dijkstra's Algorithm** is the appropriate choice.

➔Overall, the best search algorithm for a 2D maze depends on the specific requirements and constraints of