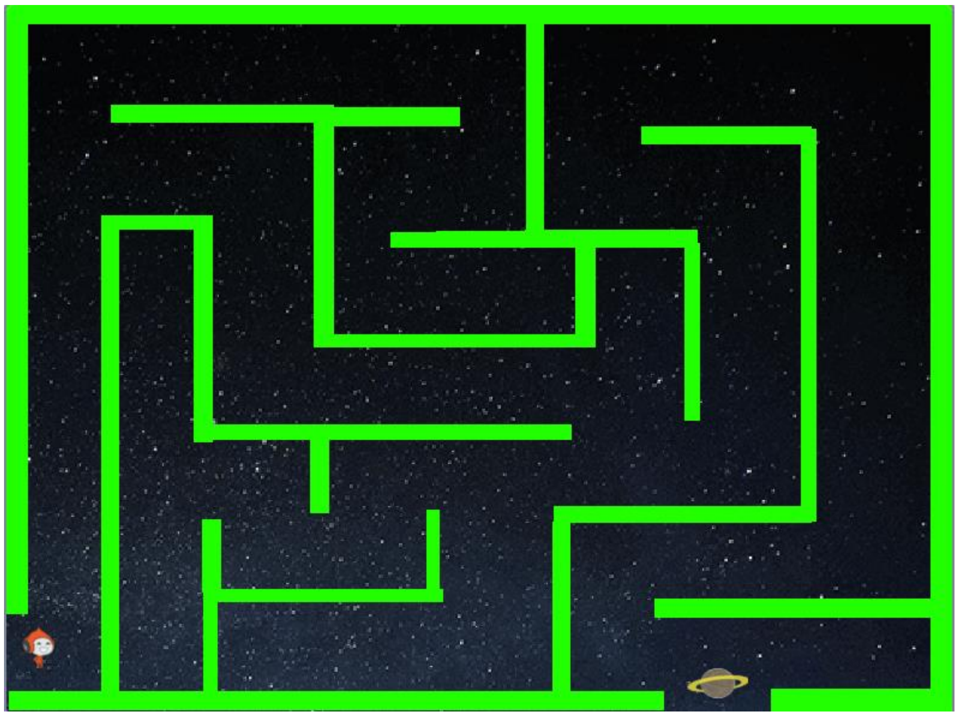
Puzzle Game

ARTIFICAL INTELLIGENCE

COURSE CODE: CS-323

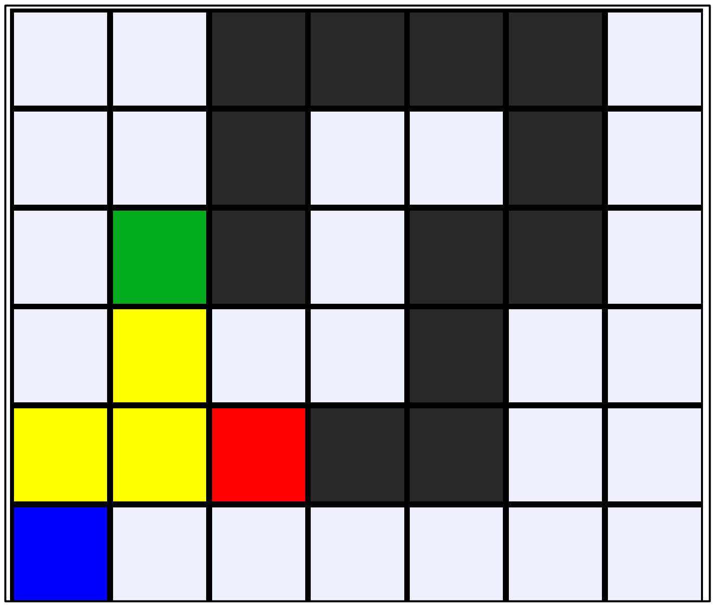


GROUP MEMBERS:

MIR SHAHNAWAZ	CS-19131
MEHMEER HUSSAIN	CS-19144
ALAUDIN TAHA	CS-19305

OBJECTIVE OF PROJECT:

Our project is all about a Puzzle game in which there is a **starting point (A)** and an **ending point (B)** a goal node. Here in this Project we have two different Scenarios and purpose of this project is to analyze the optimal path from starting to an ending point. Here we used two AI algorithms DFS (DEPTH-FIRST SEARCH) and BFS (BREADTH-FIRST SEARCH) and now we analyze which one is more optimal with the help of this project.



🟦 **Represents the starting point (A)**

🟥 **Represents the useless path (expanded nodes where goal is not present)**

🟨 **Represents the optimal path (Shortest Path) to the Goal node**

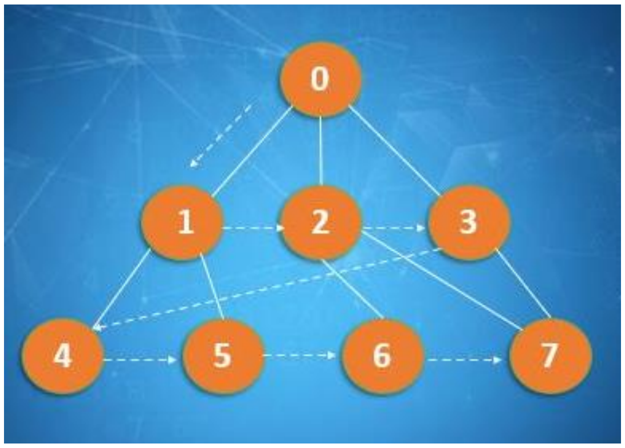
🟩 **Represents the ending point (B) the Goal**

□ **Represents the Walls (BARRIERS)**

■ **Represents empty cells**

ALGORITHMS USED:

BFS (BREADTH-FIRST SEARCH)



BREADTH-FIRST SEARCH is a classic and elegant algorithm for finding a shortest path. A breadth-first search reaches outward from the entry location in a radial fashion until it finds the exit. The first paths examined take one hop from the entry. If any of these reach the exit location, you're done. If not, the search expands to those paths that are two hops long. At each subsequent step, the search expands radially, examining all paths of length three, then of length four, etc., stopping at the first path that reaches the exit.

Breadth-first search is typically implemented using a queue. The queue stores partial paths that represent possibilities to explore. The paths are processed in order of increasing length. The first paths enqueued are all length one, followed by the length two paths, and so on. Given the FIFO handling of the queue, all shorter paths are dequeued before the longer paths make their way to the front of queue, ready for their turn to be processed.

At each step, the algorithm considers the current path at the front of the queue. If the current path ends at the exit, it is a complete solution. If not, the algorithm takes the current path and extends it to reach locations that are one hop further away in the maze, and enqueue those extended paths to be examined later. Since a path is represented as a Stack of Grid Locations, putting these paths into a queue for processing means you'll have a nested ADT, a Queue<Stack>. A nested container type looks a little scary at first, but it is just the right tool for this job.

1. Create a queue of paths. A path is a stack of grid locations.

2. Create a length-one path containing just the entry location. Enqueue that path.

   - In our mazes, the entry is always the bottom-left corner and exit in the upper-right.

3. While there are still more paths to explore:

   - Dequeue path from queue.
   - If this path ends at exit, this path is the solution
   - If path does not end at exit:
     ♠ For each viable neighbor from path end, make copy of path, extend by adding neighbor and enqueue it.
     ♠ A viable neighbor is a location that is a valid move and that has not yet been visited.

A couple of notes to keep in mind as you're implementing BFS:

You may make the following assumptions:

• The maze is well-formed. It is non-empty, fully connected, and entrance at lower-left and exit at upper-right are both open corridors.

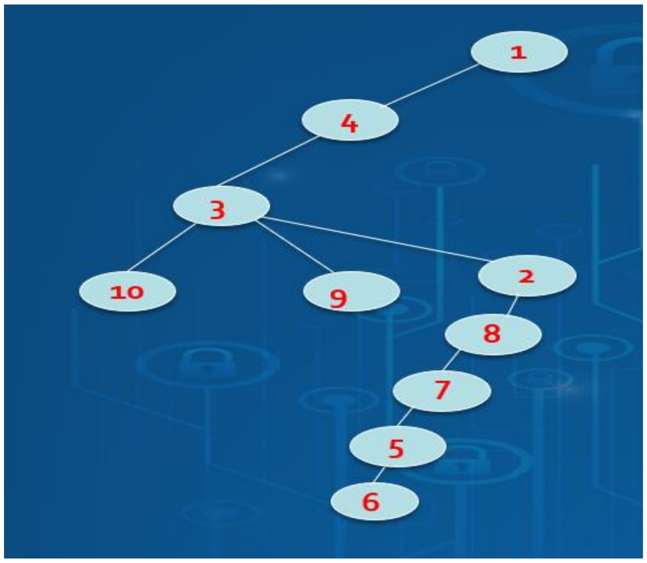
• The maze has a solution.

• You should avoid repeatedly revisiting the same location in the maze or creating a path with a cycle, lest the search get stuck in an infinite loop. For example, if the current path leads from location r0c0 to r1c0, you should not extend the path by moving back to location r0c0.

• You shouldn't use check Solution within you solve implementation, but you should use it to write tests that confirm the validity of paths found by solve

. • A strategy that will help with both visualizing and debugging solve it to add calls to our provided graphics functions, explained below. We have provided some existing tests that put your solve functions to the test. You should take advantage of the extra maze files we've provided you and add 2-3 more tests that verify the correct functionality of the solve function.

DEPTH-FIRST SEARCH (DFS):



DEPTH-FIRST SEARCH is an algorithm for traversing or searching tree or graph data structures. In simple words using some traversing technique, we can visit all the nodes of a graph in a certain order that we define. Here, the word backtrack means that when you are moving forward and there are no more nodes along the current path, you move backwards on the same path to find nodes to traverse. All the nodes will be visited on the current path till all the unvisited nodes have been traversed after which the next path will be selected.

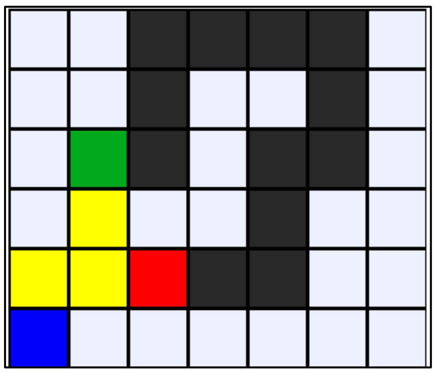
This recursive nature of DFS can be implemented using stacks. The basic idea is as follows:

Pick a starting node and push all its adjacent nodes into a stack. Pop a node from stack to select the next node to visit and push all its adjacent nodes into a stack.

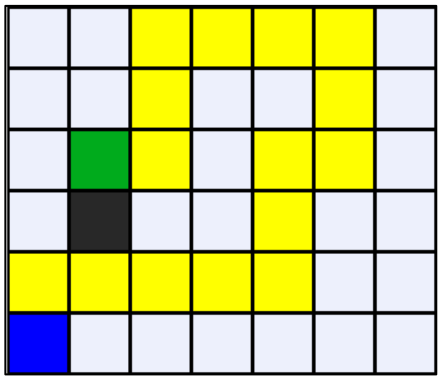
Repeat this process until the stack is empty. However, ensure that the nodes that are visited are marked. This will prevent you from visiting the same node more than once. If you do not mark the nodes that are visited and you visit the same node more than once, you may end up in an infinite loop.

WHICH ALGORITHM IS OPTIMAL?

BFS



DFS



Hence **BFS explored 6 states while DFS explored 17 states** for the same scenario.

So,

BFS is the best algorithm as it gives optimal and shortest path. This can be easily seen from **recursive nature of DFS.** It visits the **'deeper' nodes** or you can say farther from source nodes first. It goes as far as it can from the source vertex and then returns back to unvisited adjacent nodes of visited vertices.

On the other hand **BFS** always visits nodes in increasing No of their distance from the source. It first visits all nodes at same 'level' of the graph and then goes on to the next level.

Tools used:
We implemented both algorithm BFS and DFS on python and more specifically we used Pillow (PIL) library for implementation for our algorithms