Authored by: George Engel, Justin Keeling, & Cory Johns

## Overview:

This project was written in python since it is easy to use and allowed us to focus more on the concepts of the project rather than the implementation. The assignment was broken up into parts with each author contributing approximately an equal amount to the project.

Cory covered the MazeNode class, reading in the maze text files, making the node tree, the top-level search function, and printing the results. Justin handled the Depth First and Breadth First search algorithms. George handled the Greedy and A\* search algorithms. All of us extensively planned and communicated each step of the project to each other.

# Main Parts:

Read in maze from text file into 2D array

Before the maze could be searched it first had to be converted into a form that would make the search algorithms easy to make. This was accomplished by first reading in the maze from the text file and directly putting the characters into a 2D array such that each element in the array contained only one character.

Convert 2D array to node tree

To facilitate the search algorithms, the 2D array was converted into a node tree where each node represented a state in the search. Each node contained a list of all adjacent nodes that were traversable and flags to indicate if the node had been visited already.

This node tree was made in the following way:

* Starting from the 2D array of the last step
* Start at element 0, 0
* Make a new node at current location if the maze square is not a wall and the location does not already have a node.
* Then add and connect nodes at (row+1, col) and (row, col+1) if the location is in the maze and the square is not a wall or a node has not been made there already
* Repeat this process at (row, col+1) and wrap to (row+1, 0) if the end of the row has been reached
* This process allows us to hit all the nodes without checking the same node twice, and ensures that the new nodes being made have not been made previously
* Finally, the program will return the node that contains the start point, this will act as the root of the tree and will consequently lose all islands not reachable from the root, as these where not useful anyway.

Run each search algorithm with each maze type

Each search algorithm was then ran from the top level search function that called a parameter function that took the root node and end node as an input. The function then returned a list of the visited nodes as well as the number of nodes expanded / in solution which was then printed to the output file.

Depth first search

In depth first search, it is instantiated by calling the solve maze method with parameter of a start node to begin the maze and end node. The constructor for DFS has a ‘current’ node variable, a lifo queue and a visited list. By implementing methods to check if nodes have been visited, finding connected nodes, changing visited status, and adding/removing from the frontier (lifo queue) I was able to solve the maze using DFS.

One interesting implementation in my code would be retrieving the list of connected nodes from the MazeNode class. I used the copy method, from the list methods, to change the visited status without having to reset all the nodes visited status.

Another interesting implementation is the remove from frontier method. Since this is a lifo queue, I was able to set the current by using get method from import queue, to set my current node. Since it is a lifo I added the current back into the queue, so I could reuse functions which use the current node.

Breadth first search

In BFS, I used a similar approach as DFS. Using all the same methods as described in DFS, I was able to solve the maze. The only difference with these methods is that they took in a node rather than just using current node as in DFS. For a constructor, variables to represent a current node a fifo queue and visited list were used, similarly to DFS.

An interesting implementation in BFS is how to handle what node was the current node. I handled this by making the current node the node most recently removed node from the queue. This then allowed the functions to work properly without redundancy.

Another implementation in which is interesting, is where to change the status of the visited node. Changing the visited status once the node is in the queue while having the current node being the most recently removed node from the queue, I was able to go through the maze using BFS.

Greedy search

For our implementation we chose to use Greedy best first search, which expands the node that is closest to the goal in hopes of finding it quickly. For our implementation, we managed to replicate it’s activity by constructing it as an algorithm class of its own. Much like we did for DFS and BFS, we had it work off a node tree of sorts, so we would have edges to work with from node to node.

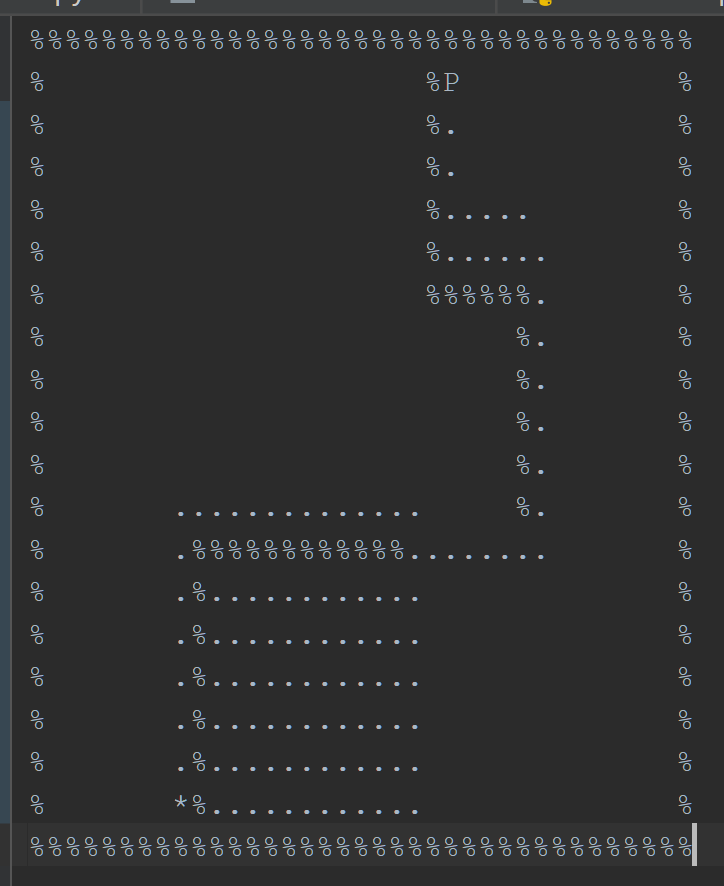
Though in our implementation, they didn’t end up being used much seeing as Greedy first search is simplistic in nature, all it does differently from the other algorithms is iterates through its frontiers to see who has the closest non-visited neighbor to the goal state, and then expands in that direction. In other words, it works off a simple heuristic rather than a pattern. The heuristic being a simple calculation of the Manhattan distance from the unvisited neighbor nodes of our frontier node’s position to the goal node’s position, which is calculated as follows: |X2 - X1| + |Y2 – Y1|.

With a normal Greedy search however, you run the risk of not actually solving the maze, if it where to say hit a dead end. So in order to combat this, we made it remove any nodes on the frontier who had no non-visited neighbors, and marked them as visited themselves. This way we could avoid getting stuck, as well as increasing the algorithm’s efficiency a bit by removing them from the iterative list.

A\* search

Our A\* search function acts much like our Greedy search function, only more reliable and always finds the most optimal solution. It is similar to our Greedy search function in the sense that it is also a heuristic based search function, and in fact is basically the same as the Greedy search function heuristic wise, however it additionally takes in to account the cost of the path from the start to the current node(in Manhattan distance) as well as the path from the current node to the end state to find the minimal cost solution.

As denoted by f(n) = g(n)+h(n), g(n) being the cost to reach the node from the start, and h(n) being the cost to reach the end state from the current node’s position. A star is both complete and optimal, so there is little else we really had to do, as most of its code was simply repurposed from Greedy search.

**Solution:** 

**Path cost: 120**

**Number of Nodes Expanded:300**

## Write-up (Justin Keeling)

For myself, I was responsible for building the DFS and BFS search algorithms to function properly, and to connect the main (which Cory was responsible for) and both search algorithms correctly. This involved much communication. Using Git hub and Git Kraken we were able to communicate and prompt each other for what needs to be updated with more recent code, or bugs that need to be fixed.

While doing the DFS, I found that the most difficult/interesting aspect of this was handling the connected nodes. Initially, I first used a function within MazeNode to get a list of all connecting nodes. I set a variable equal to the function call and proceeded. Little did I know that changing the nodes visited status was prohibiting other search algorithms to function properly. Using the copy method, I was able to copy the list without changing the node’s visited variable directly.

Within BFS, I found that it the most challenging aspect was to correctly backtrack to the correct location to expand another node. I first begun by trying to implement BFS in the same manner as DFS. This approach did not function properly. Using the debugger, I found my error and changed the methods to take in a specific node rather than using the current node, as I did in DFS. To map out my error to the longest. Because of this, the implementation time was unexpectedly longer than I anticipated.

## Write up (Cory Johns)

I was responsible for programing the nodes, the top-level search function, reading the maze from the input file, converting into the node tree structure we used, and the Main class. The most interesting part for me was catering to the requirements George and Justin had while also trying to ensure a uniform process for each search algorithm. Do to the informed vs. uninformed difference between DFS, BFS and A\*, Greedy we arrived at the decision to include both the start and end nodes as inputs to the search functions, even though DFS and BFS would not use the end node.

Nodes where chosen since they are a very natural way to represent the maze, allowing the search algorithms to traverse the maze without needing to track their location or where the walls are. Each node represents a valid state, since walls were not included. Adjacent nodes are connected to each other by a list of local nodes, that was populated when the nodes where made.