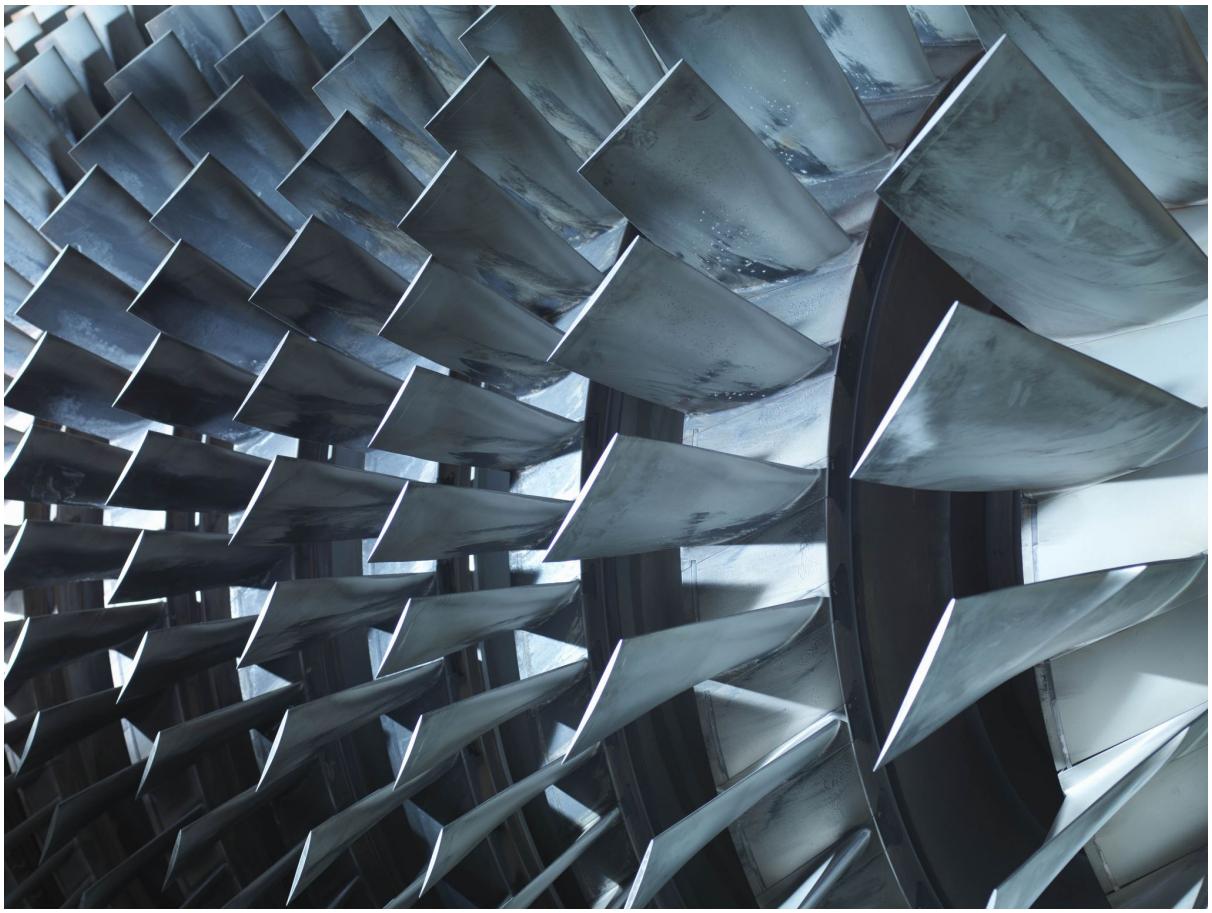
COS30019 – Introduction to Artificial Intelligence

Assignment 1 – Tree-Based Search
Option (B) – Robot Maze Navigation
By Jake Varrese – 102578350



Contents

Introduction 3

Instructions 3

How to execute command-line interface: 3

Search Algorithms 4

Depth-First Search 4

Breadth-First Search 4

Greedy Best-First Search 5

A-Star (A*) Search 5

Uniform-cost Search 6

Which search Algorithm is the best? 6

Depth-First Search Analysis 6

Breadth-First Search Analysis 7

Greedy Best First Search Analysis 7

A* Search Analysis 7

Uniform-cost Search 7

Overview 8

Implementation 9

UML Diagram 9

How each search algorithm was implemented 10

Depth-First Search Implementation 10

Breadth-First Search Implementation 11

Greedy-Best-First Search Implementation 12

Uniform-cost Search 13

A* Search Implementation 14

Features, Bugs and Missing Content 15

Features in the Program 15

Bugs found in the Program 15

Missing content in Program 15

Research 16

Graphical User Interface 16

Testing Randomly Generated Mazes w/ and without Variation of Costs 16

Results from Testing Random Generated Mazes 17

Conclusion 18

Acknowledgments 18

SwinGame – GUI Class Library 18

Swinburne University Faculty 18

References 18

Introduction

For my assignment, I have chosen to implement my knowledge of a variety of search algorithms and C# programming to create a robot maze navigator. As the name suggests, my implementation of different search algorithms tests the viability and performance of each of the algorithms as a robot attempts to navigate the maze to reach one of the goal states.

The search algorithms featured in this assignment consist of informed and uninformed methods of reaching a goal state within the maze. The robot traversing the maze using these search methods is operating in a partially-observable environment, having the capabilities to move up, down, left, and right. The robot's environment that traverses is deterministic since the robot's action are the only factor manipulating the state of the world. There is no competition featured and the environment is also static.

Instructions

For the assignment, I have completed both the command-line interface program and the GUI program. Initially, I will demonstrate how to use the command-line interface program for the RobotMazeSearch.

How to execute command-line interface:

As indicated in the assignment guidelines, the program is to be launched with command-line arguments. Below is the format the program uses for the command-line interface:

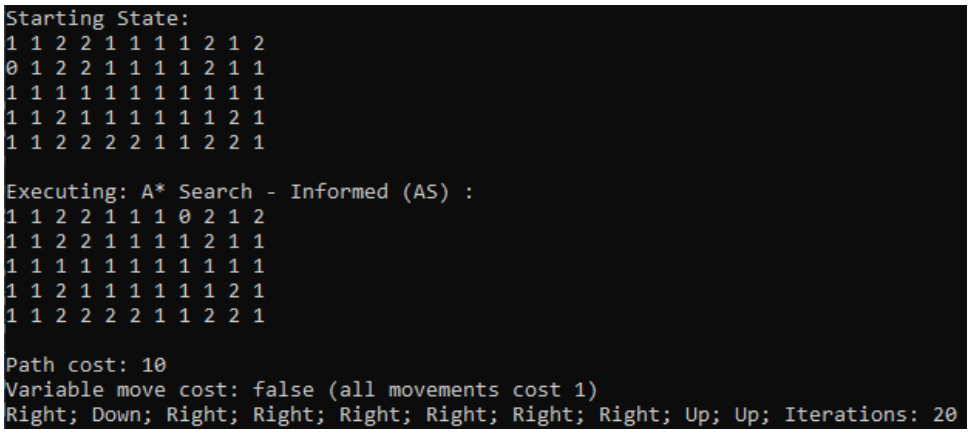
```
> <filename> <search method> <variable cost mode>
```

Filename: The provided filename for the assignment is ***mapconfig.txt*** which provides the configuration for the maze solving programming. **CAN ALSO BE** 0 **FOR TEST MODE.**

Search method: BFS, DFS, GBFS, AS, CUS1 are the options for different search methods to execute for the maze solving programming. (enter search method argument in capital letters)

Variable cost mode: 0 for OFF and 1 for ON. The variable cost option will change the traversal cost of different movements made. When off, all moves made in any direction cost 1. When on, moves have different costs (different costs displayed when the program is executed).

(***example output of A* Algorithm with Variable Cost Mode ON***)



(Please **READ** the README.pdf for instructions on building the programs for execution)

Search Algorithms

The following information relates to different search algorithms use for pathfinding in my robot maze navigation program.

Depth-First Search

The Depth-First search algorithm (DFS) traverses a tree of nodes by always selected the first child from the parent node. The general idea is that the DFS algorithm will keep going deep within the search tree until it has reached the end of the branch.

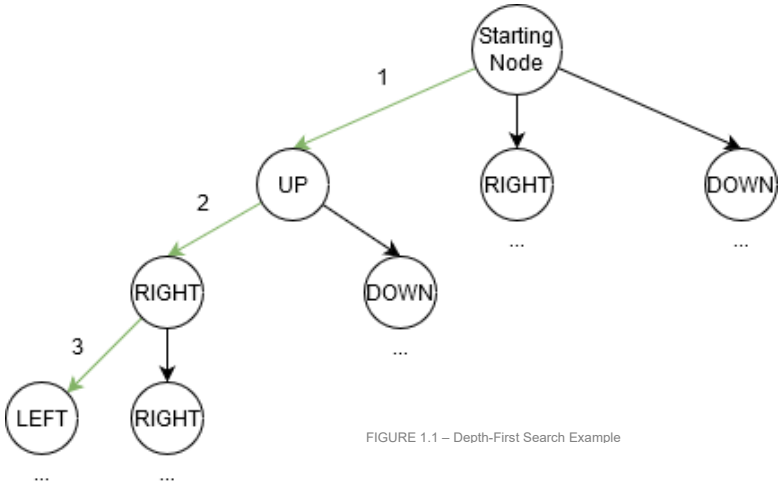


FIGURE 1.1 – Depth-First Search Example

The following search tree above demonstrates the path taken in the DFS algorithm. From the starting node identified at the top of the search tree, we first take the direction “UP” as this is the first child of the parent. From the second node, the “RIGHT” direction is taken as it is the first child from the “UP” parent. From there, the algorithm takes the “LEFT” direction as it is again, the first child node of the “RIGHT” direction.

Breadth-First Search

The Breadth-First search algorithm (BFS) will traverse the tree of nodes by exploring all child nodes in order. This approach opposed to DFS will search wide rather than deep. Once, a level of the search tree is explored, it continues to expand all nodes in this fashion.

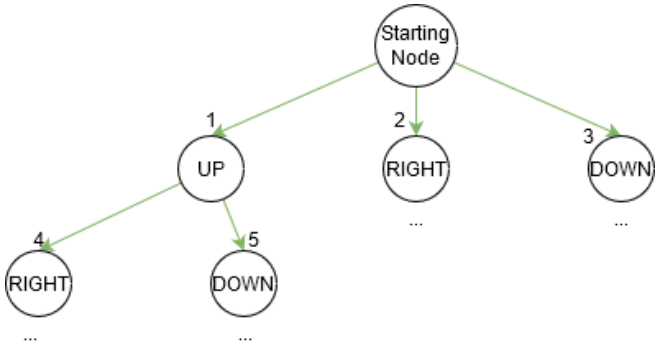


FIGURE 1.2 – Breadth-First Search Example

The following search tree above demonstrates this BFS pattern. The search tree first explores the “UP” direction followed by “RIGHT” then, “UP”. This pattern repeats for each layer as the 4th node explored is the “RIGHT” direction from the parent “UP”, followed by “DOWN”.

Greedy Best-First Search

The Greedy Best-First Search algorithm (GBFS) will traverse nodes in a search tree based on a heuristic value. The evaluation function used to determine the best direction to move in the program is *f(n) = h(n)* where *h(n)* is the distance from the current state to the goal node. Unlike the previous demonstrates search algorithms, GBFS is an informed method (has information that assists it in finding the goal node).

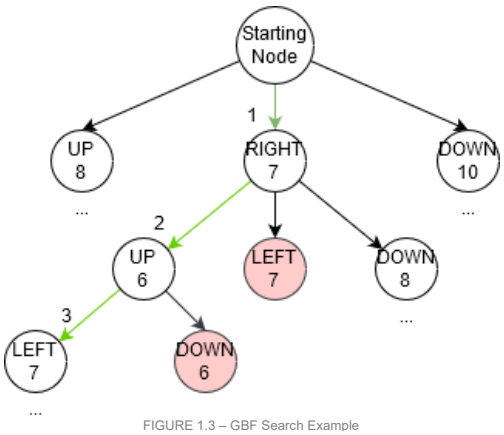


FIGURE 1.3 – GBF Search Example

The following search tree above demonstrates how the GBFS algorithm traverses the path. Each node contains the distance from the current state to the goal state (using the Manhattan distance calculation). In this example, the “RIGHT” direction to chosen first to explore as it has the lowest cost of all 3 available moves. Next, the direction “UP” with a cost of 6 is chosen as it has the lowest cost of the 5 available moves (The red nodes represent repeated states that are unwanted).

A-Star (A*) Search

The A-Star Search algorithm (A*) will traverse nodes in a search tree based on a more advanced heuristic compared to the previously demonstrated GBFS. Instead of just using *h(n)* as a heuristic, the A* algorithm uses the evaluation function *f(n) = g(n) + h(n)* where *g(n)* is the costs traversed so far and *h(n)* is the distance from the current state to the goal node.

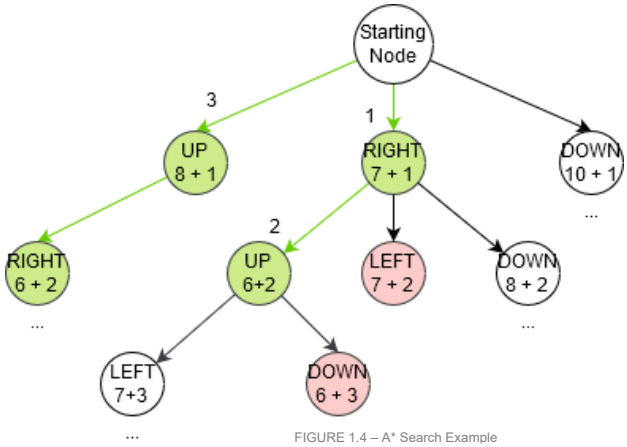
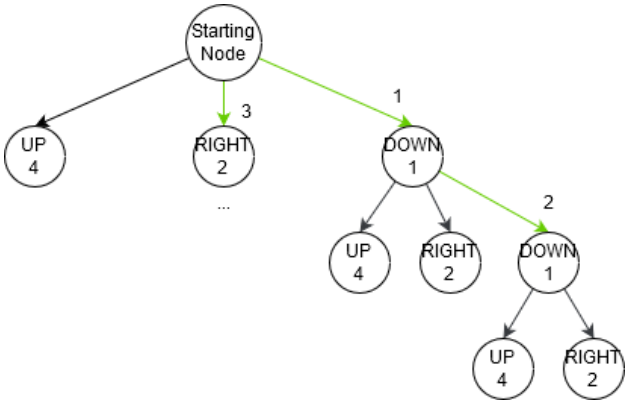


FIGURE 1.4 – A* Search Example

The following search tree above demonstrates the A* algorithm traverses the path. Compared to the GBFS algorithm, each node contains a cost which is the sum of the current cost traversed so far (by default it costs 1 per move) and the distance from the current state to the goal state.

Uniform-cost Search

The uniform-cost search algorithm (UCS) will traverse nodes in a search tree by expanding the nodes which have the smallest length first. This pattern will continue until it reaches the goal state regardless of the total cost at the end of the search.



The following search tree above demonstrates the UCS algorithm traverses the path. The first direction taken is “DOWN” as it has the shortest length of 1. It will look at all available nodes to traverse and look for the cheapest path to traverse. The “DOWN” direction is chosen one more as it has the lowest cost of 1. This repeats over and over until the goal state is discovered.

Which search Algorithm is the best?

To determine which algorithm performs the best, many factors should be considered. Below, there are two figures which contain graphs on each of the performance of the algorithms based on the cost to reach the goal state, and another on the number of iterations made to reach the goal state.

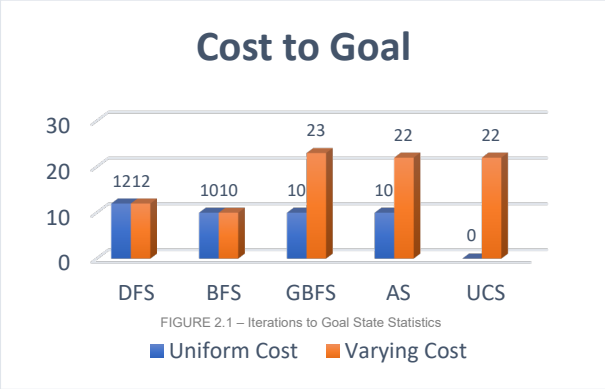


FIGURE 2.1 – Iterations to Goal State Statistics

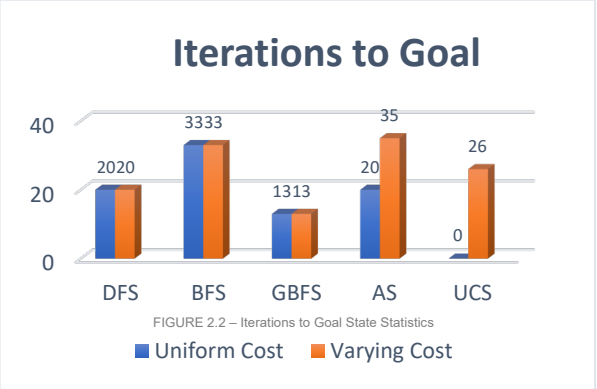
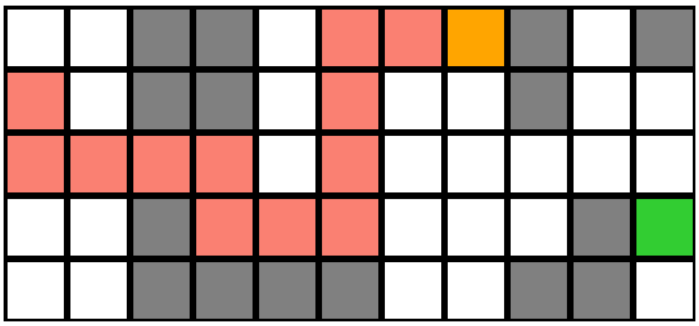


FIGURE 2.2 – Iterations to Goal State Statistics

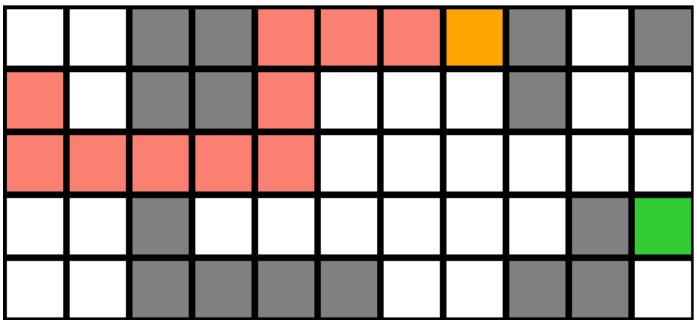
Depth-First Search Analysis

The DFS algorithm has a path cost of **12** (when all movements cost the same and with varying cost). This algorithm had to make **20** iterations to generate the path to the goal state.



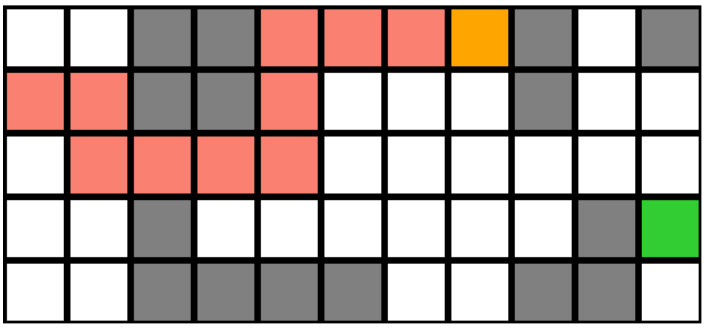
Breadth-First Search Analysis

The BFS algorithm has a path cost of **10** (When all movements cost the same and with varying cost). This algorithm had to make **33** iterations to generate the path to the goal state



Greedy Best First Search Analysis

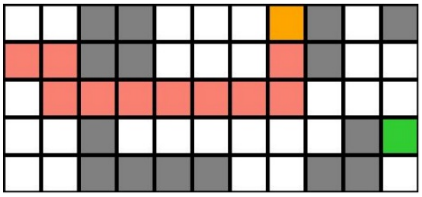
The GBFS algorithm has a path cost of **10** with no variation in costs and a cost of **23** with varying costs. This algorithm had to make **13** iterations for uniform and varying cost to reach the goal state.



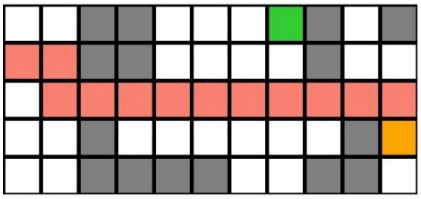
A* Search Analysis

The A* search algorithm has a path cost of **10** with no variation in costs and a cost of **22** with varying costs. This algorithm has to make **20** iterations for no variation in costs and make **35** iterations with varying costs to reach the goal state.

No Varying cost

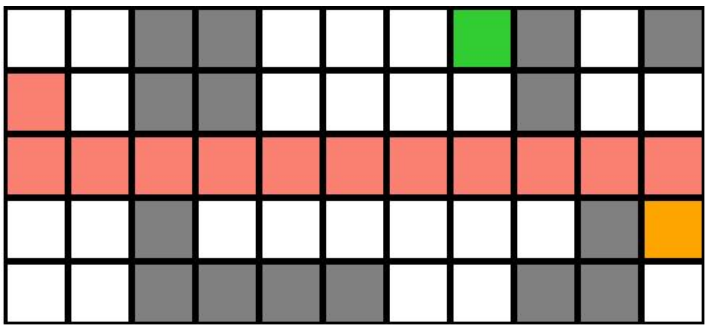


With Varying cost



Uniform-cost Search

The Uniform-cost search algorithm has a path cost of **22** with varying costs. This algorithm has to make **26** iterations with varying costs to reach the second goal state.



Overview

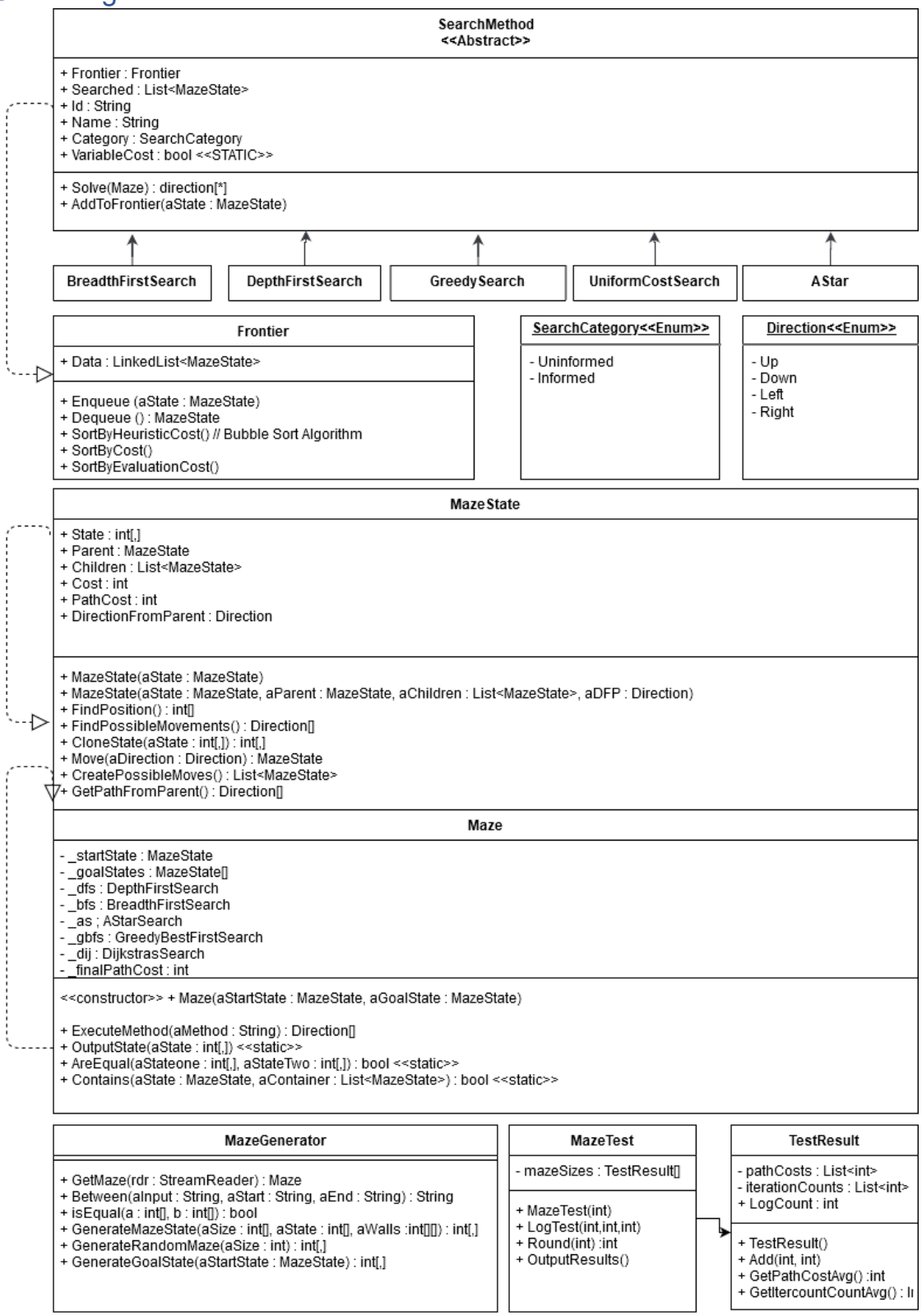
From the data gathered from the maze provided in the assignment, the **Breadth-First Search** algorithm would appear to be the best out of all four search algorithms. The BFS algorithm had one of the lowest costs when traversing the path, matching with A* and GBFS, although it has one of the highest iteration counts of **33**. Realistically, both BFS and DFS are only uninformed search methods and find the path to the goal state based on luck essentially. It can be stated that BFS would be the best-uninformed method for this sample.

For informed algorithms, it can be shown that the **A*** search algorithm proves to be the most effective between the other informed search algorithm, GBFS. Although the A* search algorithm requires more iterations to solve the maze, it provided a path with a cost lesser to the GBFS algorithm when costs vary. We can see that from the two A* algorithm executions in the images above, that without variation in movement cost, it elects goal state (1) as the optimal node. Although when we use varying costs, the A* algorithm elects goal state (2) as the optimal node since the cost of going in the UP direction is too costly. We can see that the GBFS algorithm ignores the heuristic cost of traversal so far and that is why A* is superior.

END OF SECTION

Implementation

UML Diagram



How each search algorithm was implemented

Depth-First Search Implementation

In my program, the Depth-First search algorithm is solved within the class called `DepthFirstSearch` which is a sub-class of the abstract super-class `SearchMethod`. The code below demonstrates the Solve function which takes a Maze object as a parameter. The function starts by adding the Initial state to the frontier (in this case, the frontier is a stack data structure). The function will enter a while-loop with the exiting condition of the Frontier being empty. While it contains nodes, we obtain the current state from popping (from the top) a node of the frontier when checking if this state is one of our desired goal states. If the current state in the loop is one of the desired goal states, we return the path from the initial state to the goal state. If not, we find all valid child nodes of the current state into a list and add them to the frontier. If the frontier is empty before finding the matching state, the function will return null;

```
public override Direction[] Solve(Maze aMaze)
{
    //push the StartState to the frontier
    AddToFrontier(aMaze.StartState);

    while (!Frontier.Empty())
    {
        //Pop frontier state into curState
        MazeState curState = Frontier.Pop();
        Searched.Add(curState);

        //check if curState is a goalState
        //using a loop for each of the varient goal states listed
        for (int i = 0; i < aMaze.GoalStates.Length; i++)
        {
            if (Maze.AreEqual(curState.State, aMaze.GoalStates[i].State))
            {
                Maze.OutputState(curState.State);
                aMaze.FinalPathCost = curState.Cost;
                Console.WriteLine($"Path cost: {aMaze.FinalPathCost}");
                iterationCount++;
                return curState.GetPathFromParent();
            }
        }

        //get all possible new states from curState
        List<MazeState> newStates = curState.CreatePossibleMoves();

        foreach (MazeState s in newStates)
        {
            AddToFrontier(s);
        }
        iterationCount++;

    }

    return null;
}
```

Breadth-First Search Implementation

In my program, the Depth-First search algorithm is solved within the class called `BreadthFirstSearch` which is a sub-class of the abstract super-class `SearchMethod.` The code below demonstrates the Solve function which takes a Maze object as a parameter. The function starts by adding the Initial state to the frontier (in this case, the frontier is a queue data structure). The function will enter a while-loop with the exiting condition of the Frontier being empty. While it contains nodes, we obtain the current state from dequeuing (from the end) a node of the frontier when checking if this state is one of our desired goal states. If the current state in the loop is one of the desired goal states, we return the path from the initial state to the goal state. If not, we find all valid child nodes of the current state into a list and add them to the frontier. If the frontier is empty before finding the matching state, the function will return null;

```
public override Direction[] Solve(Maze aMaze)
        {
            //push the StartState to the frontier
            AddToFrontier(aMaze.StartState);

            while(!Frontier.Empty())
            {
                //Pop frontier state into curState
                MazeState curState = Frontier.Dequeue();
                Searched.Add(curState);

                //check if curState is a goalState
                //using a loop for each of the variant goal states listed
                for (int i = 0; i < aMaze.GoalStates.Length; i++)
                {
                    if(Maze.AreEqual(curState.State, aMaze.GoalStates[i].State))
                    {
                        Maze.OutputState(curState.State);
                        aMaze.FinalPathCost = curState.Cost;
                        Console.WriteLine($"Path cost: {aMaze.FinalPathCost}");
                        iterationCount++;
                        return curState.GetPathFromParent();
                    }
                }

                //get all possible new states from curState
                List<MazeState> newStates = curState.CreatePossibleMoves();
                newStates.Reverse(); //Reverse elements to ensure priority is U > L >
D > R

                foreach (MazeState s in newStates)
                {
                    AddToFrontier(s);
                }
                iterationCount++;

            }

            return null;
        }
```

Greedy-Best-First Search Implementation

In my program, the Depth-First search algorithm is solved within the class called `GreedyBestFirstSearch` which is a sub-class of the abstract super-class `SearchMethod`. The code below demonstrates the abstract function Solve which takes a Maze object as a parameter and returns the path for the algorithm. This function begins by first adding the initial state of the maze to the frontier. The function enters a while-loop set the current state by popping a value off the frontier. The current state is checked to see if it is one of the goal states and if so, it will return the path from the initial state to the goal state. If not, the function then creates a new MazeState object which is created by a function called PickDesiredGoalState. This function is picking the closest goal state to the current state being explored. The function will then find all viable child nodes from the current state and add them to the frontier. At the end of the function, the frontier is sorted in ascending order by the Heuristic value (*f(n) = h(n)*)

```
        public override Direction[] Solve(Maze aMaze)
        {
            //push the StartState to the frontier
            AddToFrontier(aMaze.StartState);

            while (!Frontier.Empty())
            {
                //Pop frontier state into curState
                MazeState curState = PopFrontier();

                //check if curState is a goalState
                //using a loop for each of the varient goal states listed
                for (int i = 0; i < aMaze.GoalStates.Length; i++)
                {
                    if (Maze.AreEqual(curState.State, aMaze.GoalStates[i].State))
                    {
                        Maze.OutputState(curState.State);
                        aMaze.FinalPathCost = curState.Cost;
                        Console.WriteLine($"Path cost: {aMaze.FinalPathCost}");
                        iterationCount++;
                        return curState.GetPathFromParent();
                    }
                }

                //Find most desirable goal state - i.e. which is closes to the current
state
                MazeState ClosestGoal = PickDesiredGoalState(curState, aMaze.GoalStates);

                //get all possible new states from curState
                List<MazeState> newStates = curState.CreatePossibleMoves();
                newStates.Reverse(); //Reverse elements to ensure priority is U > L > D > R
                //add all children frontier
                foreach (MazeState s in newStates)
                {
                    s.HeuristicCost = GetManhattanDistance(s, ClosestGoal);
                    AddToFrontier(s);
                }

                //Sort the frontier by f(n) = h(n)
                Frontier.SortByHeuristicCost();
                iterationCount++;

            }
            return null;
        }
```

Uniform-cost Search

In my program, the Depth-First search algorithm is solved within the class called `UniformCostSearch` which is a sub-class of the abstract super-class `SearchMethod.` The code below demonstrates the Solve function which takes a Maze object as a parameter. The function starts setting the Variable Cost mode to true so it can execute with different cost directions. Then, the function continues by adding the Initial state to the frontier (in this case, the frontier is a stack data structure). The function will enter a while-loop with the exiting condition of the Frontier being empty. While it contains nodes, we obtain the current state from popping (from the top) a node from the frontier when checking if this state is one of our desired goal states. If the current state in the loop is one of the desired goal states, we return the path from the initial state to the goal state. If not, we find all valid child nodes of the current state into a list and add them to the frontier. After the nodes are added to the frontier, the frontier is sorted by the cost of each node in ascending order. Sorting in ascending order ensures that the lowest value child is popped from the frontier. If the frontier is empty before finding the matching state, the function will return null;

```
public override Direction[] Solve(Maze aMaze)
{
    //ensure variable cost mode is on
    SearchMethod.VariableCost = true;

    AddToFrontier(aMaze.StartState);

    while(!Frontier.Empty())
    {
        //Pop frontier state into curState
        MazeState curState = Frontier.Pop();
        Searched.Add(curState);
        //check if curState is a goalState
        //using a loop for each of the varient goal states listed
        for (int i = 0; i < aMaze.GoalStates.Length; i++)
        {
            if (Maze.AreEqual(curState.State, aMaze.GoalStates[i].State))
            {
                Maze.OutputState(curState.State);
                aMaze.FinalPathCost = curState.Cost;
                Console.WriteLine($"Path cost: {aMaze.FinalPathCost}");
                iterationCount++;
                return curState.GetPathFromParent();
            }
        }

        //get all possible new states from curState
        List<MazeState> newStates = curState.CreatePossibleMoves();

        foreach (MazeState s in newStates)
        {
            AddToFrontier(s);
        }
        iterationCount++;

        Frontier.SortByCost();
    }

    return null;
}
```

A* Search Implementation

In my program, the Depth-First search algorithm is solved within the class called `AStarSearch` which is a sub-class of the abstract super-class `SearchMethod.` The code below demonstrates the abstract function Solve which takes a Maze object as a parameter and returns the path for the algorithm. This function begins by first adding the initial state of the maze to the frontier. The function enters a while-loop set the current state by popping a value off the frontier. The current state is checked to see if it is one of the goal states and if so, it will return the path from the initial state to the goal state. If not, the function then creates a new MazeState object which is created by a function called PickDesiredGoalState. This function is picking the closest goal state to the current state being explored. The function will then find all viable child nodes from the current state and add them to the frontier. At the end of the function, the frontier is sorted in ascending order by the EvaluationCost (*f(n) =g(n) + h(n)*)

```
public override Direction[] Solve(Maze aMaze)
        {
            //push the StartState to the frontier
            AddToFrontier(aMaze.StartState);

            while (!Frontier.Empty())
            {
                //Pop frontier state into curState
                MazeState curState = PopFrontier();

                //check if curState is a goalState
                //using a loop for each of the varient goal states listed
                for (int i = 0; i < aMaze.GoalStates.Length; i++)
                {
                    if (Maze.AreEqual(curState.State, aMaze.GoalStates[i].State))
                    {
                        Maze.OutputState(curState.State);
                        aMaze.FinalPathCost = curState.Cost;
                        Console.WriteLine($"Path cost: {aMaze.FinalPathCost}");
                        iterationCount++;
                        return curState.GetPathFromParent();
                    }
                }

                //Find most desirable goal state - i.e. which is closes to the current
state
                MazeState ClosestGoal = PickDesiredGoalState(curState, aMaze.GoalStates);

                //get all possible new states from curState
                List<MazeState> newStates = curState.CreatePossibleMoves();

                //add all children frontier
                foreach (MazeState s in newStates)
                {
                    //Need to implement a way to have a heuristic cost for each of the goal
states
                    s.HeuristicCost = GetManhattanDistance(s, ClosestGoal);
                    AddToFrontier(s);
                }

                //Sort the frontier by f(n) = g(n) + h(n)
                Frontier.SortByEvaluationCost();
                iterationCount++;
            }
            return null;
        }
```

Features, Bugs and Missing Content

Features in the Program

Here is a list of features my assignment program contains. These features assist in demonstrating the abilities of the different range of search algorithms.

- 5 functioning search algorithms which can be used to solve a maze
- Graphical user interface (GUI) which visually demonstrates the process of each of the 5 search algorithms. (Figure 5.1)
- Varying cost mode for search methods
    - It allows the search algorithm to use different costs when traversing in different directions instead of having a cost of 1.
- Random Maze generation **testing** for search algorithms (Console ONLY)
    - The user can use the command-line argument 0 instead of a filename for 45 randomly generator mazes with varying sizes which outputs a collection of results, sorting the sizes of the mazes with path cost and iteration count average.

Bugs found in the Program

There are currently no known bugs or issues during the execution of the current version of my program.

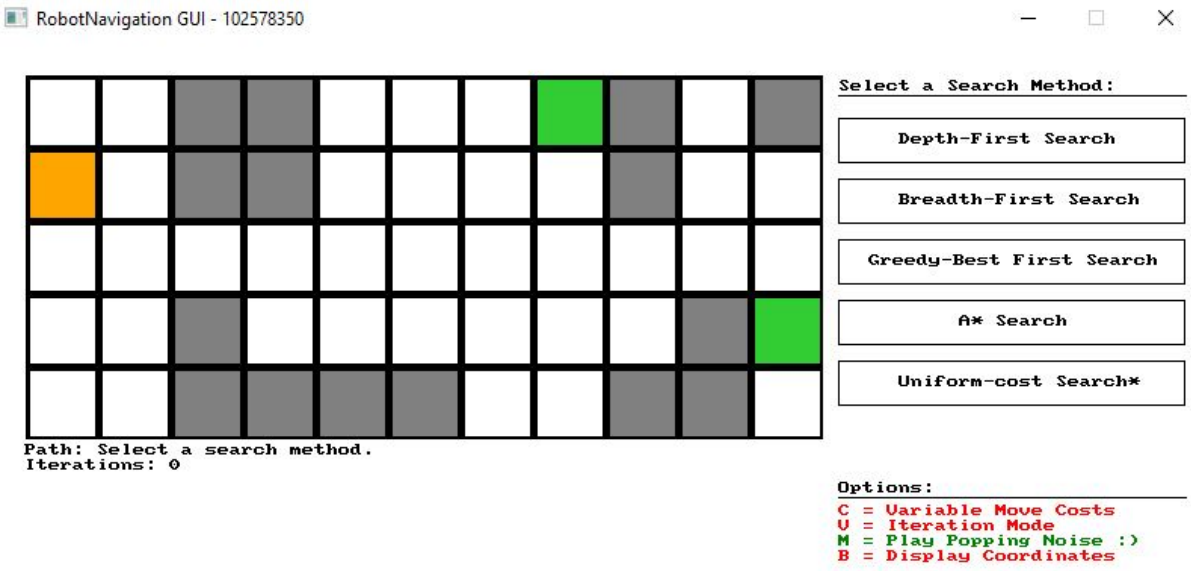
Missing content in Program

- I have failed to implement a 6th search method. The last custom method I should have implemented was supposed to be **Bidirectional A* Search.**

END OF SECTION

Research

Graphical User Interface



Using the SwinGame and RobotNavigation (The assignment project) class libraries, I was able to develop a visual representation of the different search algorithms used in the console application. The use of this GUI application is simple.

Each search algorithm can be played through by **clicking** any of the boxes with the search algorithm names on the left panel.

The program also includes different options:

1) Variable Move Costs Toggle (UP = 4, DOWN = 1, LEFT = 2, RIGHT = 2)
2) Iteration mode – Rather than displaying the path, it iterates through each of the search algorithm's search tree iterations to demonstrate its solving process.

Some side options that **don't** affect how the algorithm is executed are:

1) Play popping noise
2) Display Coordinates

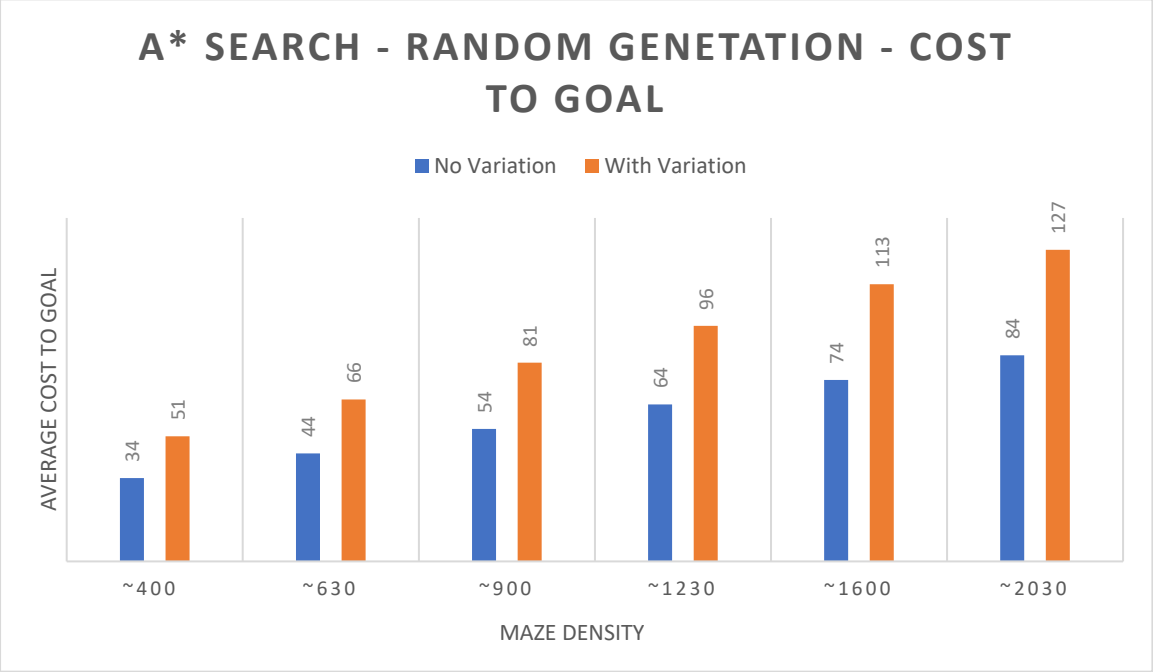
Testing Randomly Generated Mazes w/ and without Variation of Costs

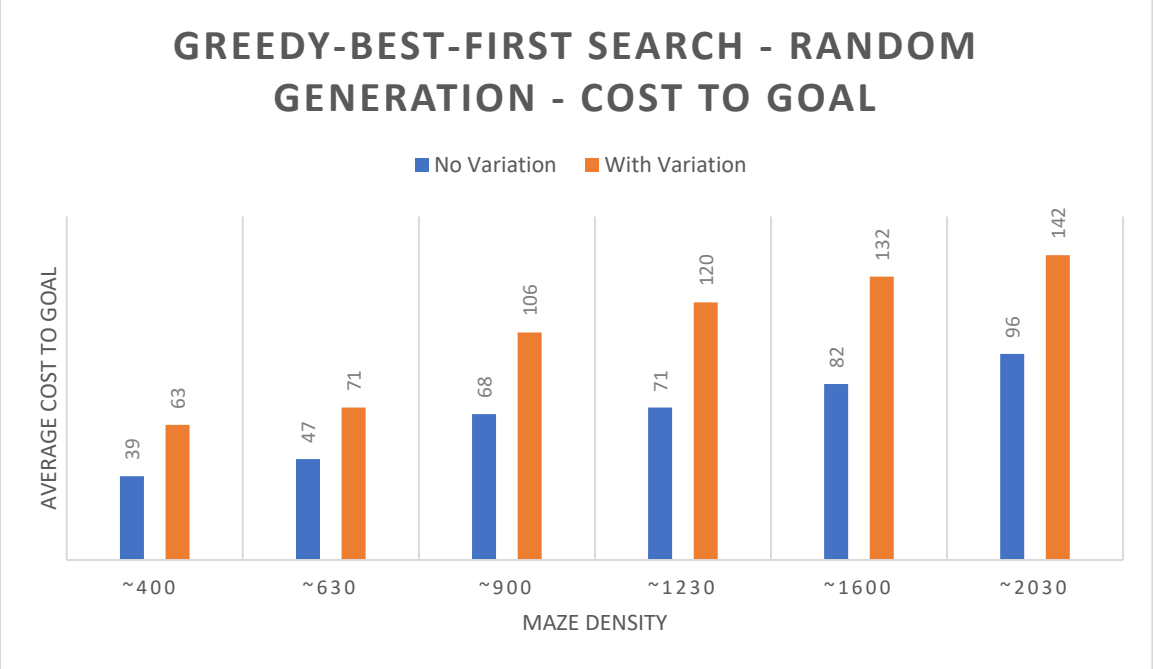
I've also decided to research the effectiveness and efficiency of each of the search algorithms when dealing with larger mazes with randomly generated conditions. Using the MazeGenerator, MazeTest, and TestResult class, I have implemented a method that generates 45 different mazes with sizes that increment throughout the random generation.

Results from Testing Random Generated Mazes

The following graphs below demonstrate 4 of the different tests done with the random generation of the mazes. Each of the different mazes is categorized into a **density** which is simply (**mazeWidth * mazeHeight) rounded to the nearest 10.** The different tests were for both no variation in traversal cost and with variation in traversal cost.

The main goal from these larger-scale tests was to see which of the two algorithms would solve the maze and achieve a lower average cost of the path. As the A* algorithm considers the cost its traversed so far, it was expected to be the optimal solution.





From the graphs above, it can be shown there is a lesser cost of the path with the A* algorithm when a variation of cost is used for different movements. As the A* algorithm can produce these lower costing paths it would be considered **superior** to the GBFS algorithm.

Conclusion

It can be concluded that the A* algorithm performs the best as an informed method which is more evident when presented with a larger problem. As demonstrated with the random generation of different mazes with varying sizes, the A* algorithm achieved a lower average in cost throughout different maze densities. In terms of uninformed search methods, it has been shown that the Breadth-First Search performed the best as an uninformed method over DFS and UCS.

Overall, If I were to improve anything in this assignment it would be to plan out everything better and ensuring I was certain with my decision before starting to code the project. I would have liked to implement a second custom informed method and I think that the performance of my code could be improved through the use of different data structures.

Acknowledgments

SwinGame – GUI Class Library

Resource link: http://www.swingame.com/index.php/documentation.html
The SwinGame class library assisted me in developing the graphical user interface for the research task.

Swinburne University Faculty

Both **A/Prof. Bao Vo and (**COS30019 Convenor and Lecturer**)** and **Mahbuba Afrin (**COS30019 Tutor**)** have worked exceptionally to teach and demonstrate the concepts of AI agents and explaining different search algorithms which have been extremely valuable to me completing this task.

Draw.io

Resource link: http://www.draw.io/
This resource assisted me in making some of the diagrams and UML featured in this assignment which assisted me in understanding the task ahead and to present my thinking.

References

www.javatpoint.com. (2011). *Uninformed Search Algorithms - Javatpoint*. [online] Available at: https://www.javatpoint.com/ai-uninformed-search-algorithms.

www.javatpoint.com. (2011). *Informed Search Algorithms in AI - Javatpoint*. [online] Available at: https://www.javatpoint.com/ai-informed-search-algorithms.

GeeksforGeeks. (2014). *Bubble Sort - GeeksforGeeks*. [online] Available at: https://www.geeksforgeeks.org/bubble-sort/.

END OF REPORT