Carl Alexander Bird

102109116@student.swin.edu.au

Assignment 1

Tree-based searching algorithms programming assignment

Introduction to AI - Assignment 1, Option B

COS30019 – 2019 Semester 1

# Introduction

## Basics

For the first assignment of COS30019 – Introduction to AI, students were tasked with utilizing tree-searching algorithms to create a solution to 1 of 2 options of problems. For this assessment submission, I chose Option B – using these algorithms to find a path within a 2D plane maze generated in a particular format with text files.

Using a combination of informed and uninformed traversal methods, my submission allows users to provide a filename and search method to a command prompt window and the program will output a solution (if there is one!) to the console in return. The output contains a 2 paths (one from the start to the first goal, and one from the first goal to the second) in position format (printing co-ordinates) and direction format (printing the directions in which a “player” would move to reach the goal), along with other diagnostics information such as time taken. When parsing files containing maze data

## Assumptions

## Definitions

* Tree Graph
* Pathfinding Method
* Informed Traversal Method
  + A search algorithm that uses a heuristic to find a solution faster/more efficiently or to create a shorter heuristic.
* Uninformed Traversal Method
  + A search algorithm that finds a route without using a heuristic value within its calculations.
* DFS
  + *Depth First Search.*
* BFS
  + *Breadth First Search.*

# Search algorithms

## Depth first search

Being one of if not the most simplest of search algorithms, Depth first search is an uninformed traversal method that starts from an origin point and traverses down one possible path until it can no longer make any valid traversals deeper down the path tree. If a “dead-end” is reached, it will then backtrack and repeat the process until a goal has been found or all potential nodes have been explored.

Whilst simple and easy to implement, DFS does not offer many advantages compared to other search methods when finding a route towards a goal (in the case of pathfinding) or traversing a tree. Compared to informed search methods that use a heuristic (such as A\*), quick tests on a map with a single wall between an evenly spaces start and goal suggests Depth-first search requires *2.3 times* the iterations to complete, and *51%* more nodes to be explored to find a valid (not even shortest!) path to the goal.

## breadth first search

Similar to depth-first search, Breadth first search is another simple uninformed search algorithm that utilizes a queue to find a path to a goal, expanding each neighbouring node on the tree before moving deeper “down” into other sub-nodes present within the tree graph.

As Breadth-first search is so similar to Depth first search, it again shares many of the performance weaknesses and issues that DFS has. As an uninformed search method without a Heuristic value to use to guide more effectively to the goal state/node, Breadth first search explores a lot of unnecessary nodes to find a valid path to the goal, which both increases search time and amount of nodes explored – not to the degree of Depth first search, but still less efficient (around 50%) compared to informed search methods. Because Breadth-first search searches all neighbours on the same “hierarchy” level of the tree graph’s currently explored node, it often can return a shorter result before Depth-first as it will find the first valid goal state/goal node that it runs into on the tree graph, rather than traversing one way down the tree graph and then re-iterating in the same fashion as Depth-First search.

## Greedy best-first search

Being the first of the informed search methods within my solution, greedy-best first search uses Manhattan distance as a heuristic to quickly find a path towards the goal. When traversing the 2 dimensional mazes within the Robot Navigation Problem, a greedy best-first search will abandon nodes that are more expensive according to the Manhattan cost heuristic, in favour of cheaper nodes. This will cause the algorithm to abandon nodes that look unfavourable by favouring nodes with a lower heuristic cost (as they are perceived to be closer to the goal), resulting in a usually faster pathfinding time for larger/more open mazes. This does come at a cost however, as it is not always guaranteed that the path found by a greedy best-first search is the shortest as it will abandon nodes that could potentially lead to a shorter path overall whilst searching for the fastest route it can find.

Compared to another informed search method such as A\*, greedy best search is able to find results *35%* faster (using the same simple map used within the BFS/DFS tests), but when testing with multiple maps does not always return the same (nor the shortest possible) path as A\* or Uniform Cost (Dijkstra’s Algorithm). Greedy-best is effective at quickly finding *a* possible path to a goal, but may end up being longer overall than other informed methods.

## A\* search

As a combination of Dijkstra’s algorithm and Greedy best-first search, A\* is a tree traversal method that combines two heuristics to allow for faster searching of more efficient paths in most scenarios. A\* search uses as it’s heuristic, with *g* referring to the cost so far to reach the current node (from the start to the goal) and *h* being the heuristic value representing the distance from the currently explored node to the goal. When searching for valid paths to a goal, the A\* algorithm will favour finding the route with the lowest overall *f* cost – meaning that the route found is as short as possible (because the *g* cost increases with each step added) and unnecessary nodes that need not be expanded can be abandoned thanks to the *h* value heuristic.

Whilst not perfectly optimal, A\* is one of the most popular tree-traversal methods due to its flexibility and its speed at finding the shortest path possible. As said above when comparing A\* search with Greedy best-first search, the extra heuristic value included within A\* causes it to return a path to the goal more slowly than other informed search methods, but because it avoids expanding paths that are already expensive, it overall searches less nodes and finds the shortest path in a faster time than uninformed search methods, and always finds the shortest path possible (unlike Greedy best-first search).

## uniform search

Uniform search (also known as Dijkstra’s algorithm) is an informed search method that uses *g* as its heuristic, with the value of *g* increasing with every node added to a potential path. When evaluating new nodes to be explored, Uniform Cost search will expand the node with the current lowest *g* value, aiming to find the shortest path to the goal possible.

Compared to Breadth-first search, Uniform Cost search’s use of heuristics comes at a minor performance increase (only around *8-12%* in node count and time efficiency), but is able to find the shortest path to a single goal faster than other search routes. Because of the single heuristic value, compared to A\* a Uniform Cost Search/Dijkstra’s algorithm will explore and expand more nodes overall (around *30-40%* more on simple map tests), but this is expected as it is a much more simple algorithm. Whilst irrelevant to the basic requirements of the assignment, Uniform cost search (as the name implies) only correctly and efficiently works if all of the path costs are equal regardless of direction travelled. If the assignment required certain directions to be more expensive (i.e. UP had a cost of 10, but DOWN has a cost of 15), or certain tiles to reduce the cost (negative weight edges), then a Bellman-Ford algorithm would have to be used instead.

## bidirectional breadth first search

Whilst functionally almost identical to standard Breath First Search, I have included a bidirectional Breadth-first search both as an interesting “code challenge” and to see how it’s efficiency would compare with a simple breadth first search method. Functionally, a Bidirectional Breadth-First Search only operates when the entire grid is known (as is the case with the mazes in this assignment), and when the goal state has been reached. As an uninformed search method, bidirectional search starts simultaneously from the goal node and the starting node – with the goal node searching for the start node, and the start node searching for the goal node. When the two paths being searched for intersect (they meet on the same node), a solution is deemed found and the two paths can be joined together into one final path solution.

Performance-wise, the Bidirectional Variant of Breadth-first search is much less time efficient, being *2.25x* slower than the single-directional search. This is most likely because the bidirectional search has to do over double the operations of the single-direction search, making two paths and then assessing if each of the paths have intersected at the end of each also iteration. This time deficit is made up by requiring an average of *1/6th* the number of iterations compared to a single-directional search, and also requiring roughly half the nodes to be assessed overall.

# Implementation

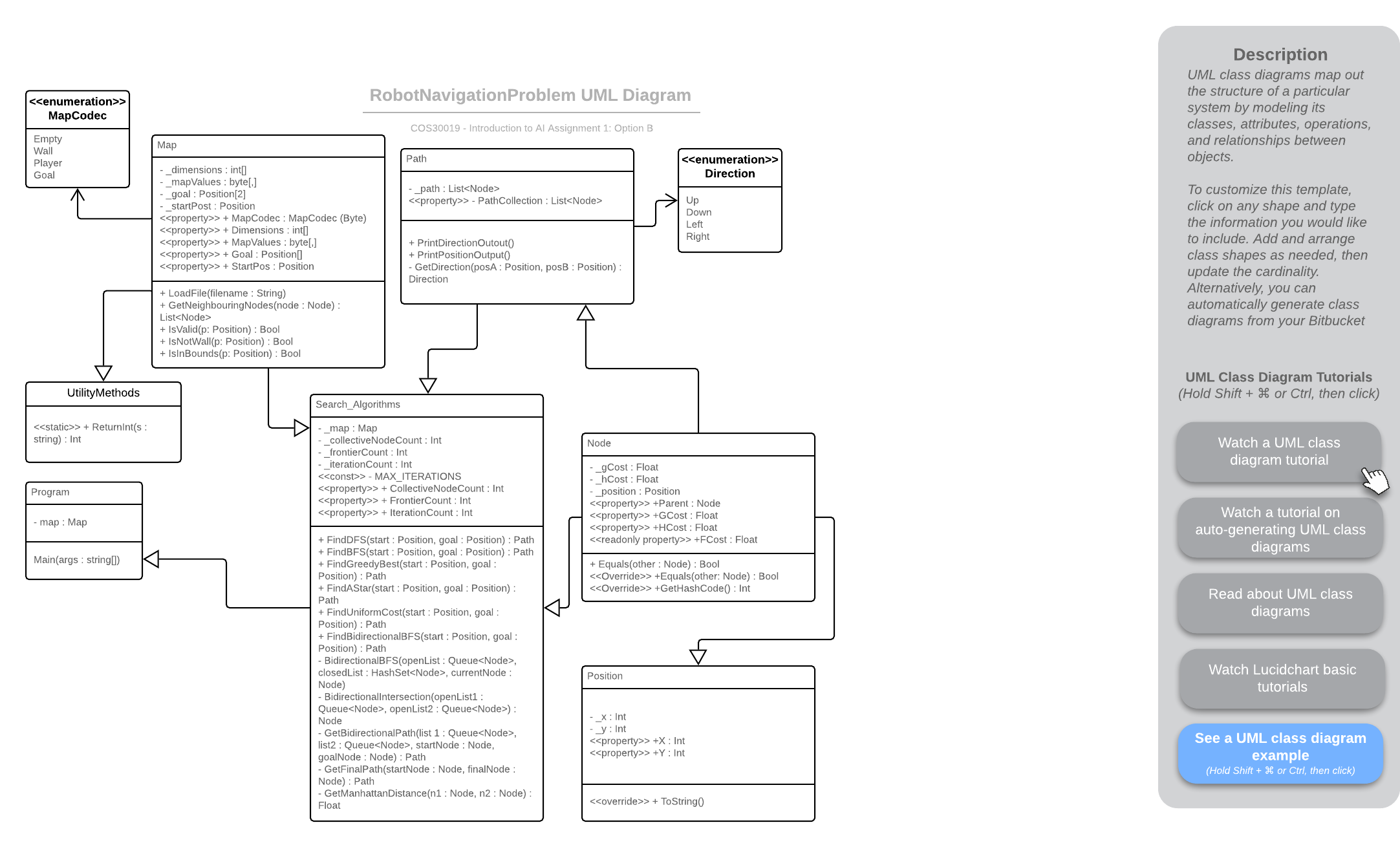


Figure : UML Class Diagram of entire solution

## General implementation comments

One of the common features throughout all of the search algorithms used within my project are HashSets – a data type I found whilst researching about possible methods of implementation for my project. From my code comments:

HashSets have generally much more consistent performance compared to lists when dealing with large amount of objects contained within the set. OpenList will be represented as a List as there will not be many items contained within it over time. Since ClosedList will contain many nodes over time as the path is being found, to keep performance fast a HashSet is utilized.

The use of HashSets assisted with the performance of my node comparisons (keeping them short and efficient), especially when the amount of nodes increases (which is more of a prevalent issue on larger mazes to find a solution to). The frontier within each of the search algorithms cannot be represented by a HashSet however as they do not allow for duplicate elements to be added nor are they able to be sorted (which is important for informed search methods). These limitations are perfect for the closedList within each of my algorithms however and will not impede on any features of the solution.

## Search algorithm implementation

### Depth first search

My Depth-first search implementation uses a simple list object to store the “frontier” (known henceforth as the openList in this and other algorithms), which is the list of nodes that potentially lead towards a goal. The frontier is first set as the starting node, before a loop commences that continues to operate until the maze has exhausted all options within the frontier. When in this loop, the frontier will pop off the first value within the list, add it to a HashSet of nodes to now ignore (known as the “closedList” in all algorithms, discussed above), and then check if the current node is at the goal state. If not, the current node will be expanded for its orthogonal neighbours and then before being added to the openList, these neighbouring nodes will be checked if they are invalid (within a wall or out of bounds), or on the open/closed list already. This process will loop until either the max iteration count is reached, or until a goal node is found. This personal implementation of Depth-first search uses a list instead of the often normal stack, as I had issues implementing a stack effectively.

### Breadth first search

Similar to Depth-first search, Breadth-first search uses a queue to store the potential nodes to be explored, and a HashSet to store nodes that have already been explored and dismissed (so they aren’t re-opened). The openList is looped through and explored for neighbours repeatedly, with valid (within maze bounds, not in a wall, not on the closedList or the open list) neighbours being added to potential nodes to be explored and then expanded on later. If the max iteration count is reached or if the currently explored node is in the same “state” (position) as the goal node, the loop is broken and either an error or the final path is returned respectively.

### greedy best-first search

As the first and most simple of the informed search methods included within the program, Greedy-best first search

### A\* Search

### Uniform cost search

### bidirectional breadth first search

# Additions

## Additional features

## Bugs

## Missing features

# Research

# Acknowledgements/references

