COS30019 - Introduction to AI

Assignment 01 – Tree Based Search

Minh Thu Nguyen - 103499232
Wednesday 12:30

Table of Contents

1. Instructions

1.1 Basic instructions:

Search Tree Algorithms:

Please note that the research initiatives carried out in this work is **getting the robot visit all green cells with the shortest path**. Therefore, the program can get operated at ease by following those steps:
- Open CMD.
- Set the directory in CMD to the workspace folder.
- Input format: *search.exe <filename> <method>*

→ Output will be shown as the following format: *filename method number_of_nodes*
path

Note: This is how I name my methods: BFS, DFS, CUS1, GBFS, AS, CUS2. Please note that the input file must strictly follow the given format in the instruction file. Otherwise, my program will not work. It has not been designed to handle random input value or wrong format input file.

```
D:\Swinburne\S1 2023\Intro to AI\A1\Code>search.exe RobotNav-test.txt BFS
RobotNav-test.txt BFS 34
DOWN RIGHT RIGHT RIGHT RIGHT UP UP RIGHT RIGHT RIGHT
```

Figure 1. Screenshot showing an example of how to trigger my program.

Research Initiative – multi-goal shortest paths:

For my research initiative, the program can still be executed as the preceding way with the name of method as "**MGPF**". Output will be displayed as:
- *The list of grids visited chronologically and the path to get to them*. This output is calculated based on my own design of searching algorithm.
- *The actual shortest path computed by permutation*. The purpose of this calculation is to examine whether my own algorithm got the correct answer.

This is one example of the input to trigger the execution: **search.exe RobotNav-test.txt MGPF.**

```
D:\Swinburne\S1 2023\Intro to AI\A1\Code>search.exe RobotNav-test.txt MGPF
RobotNav-test.txt MGPF just been to (7, 0) by DOWN RIGHT RIGHT RIGHT RIGHT UP UP RIGHT RIGHT RIGHT
Steps taken: 10
just been to (10, 3) by DOWN DOWN RIGHT RIGHT RIGHT DOWN
Steps taken: 6
Number of steps: 16
Number of nodes: 311

PLEASE NOTE THAT PERMUTATION IS RUNNING IN PROGRESS.
The higher number of goals, the longer waiting time.
Actual cost by permutation: Shortest path: 16
```

Figure 2. Screenshot showing an example of how to trigger my research intuitive

1.2 Note to marker about my research intuitive:

For my own design algorithms to find the shortest path to all green cells, it is not able to find the shortest path in all cases. In most cases, my program can still yield the optimal path. When the input size is large or density of goals is too high, however, it will return the sub-optimal path. In such cases, permutation result can give you an idea of how good the returned path is.

Currently, there is no algorithm that can fully solve the problem. To be clear, on the one hand, we will always be able to find the optimal solution using brute-force. That is, however, a trade-off between execution time and the accuracy. We cannot just wait centuries to get the optimal solution for a case. The proof that this problem is a variant of Travelling Salesman Problem (TSP) will be shown in section 6. TSP is known to be NP-hard, which means that currently there is no polynomial-time algorithm that can solve TSP optimally for all possible instances of the problem. In other words, as the number of nodes increases, the time required to find the optimal solution to the TSP grows exponentially.

My research intuitive may not require hard code as other extension options as it focuses much more on reading and researching. Accordingly, the viewable amount of work presented via coding may be much less than the effort that I took to complete this research intuitive. I apologize for exceeding the page limit,

but 12 pages can really not reflect what I dedicated to this assignment. I understand if this causes any inconvenience and I appreciate your understanding.

2. Introduction

Path finding is a common problem in computer science and is applied into many fields, including AI. The goal of pathfinding in AI is to enable the agent to navigate its environment and reach its destination from its start position while avoiding obstacles and minimizing the cost required to do so. Robot navigation Problem is one of the possible instances of pathfinding problems.

In Robot navigation Problem, the environment is a grid map with several walls placed randomly around. Robot is initially placed at a valid empty cell and its goal is to reach one of specific cells, which are colored in green. In this problem, the robot needs to perceive its environment and plan its movement to avoid walls, reach its destination, and potentially optimize cost.

Different approaches will be used to solve this problem. Three uninformed searches, which are BFS, DFS and Bi-directional BFS, are applied to help robot find the path to one of any goal grids. Meanwhile, other three informed searches, which are GBFS, A* and IDA*, are implemented to help robot find the path to the nearest possible goal. Among six of those algorithms, only BFS, A* and IDA* can yield the optimal path.

A new algorithm using the idea of heuristic function is also proposed as a solution to research intuitive that finds the shortest path to visit all the goal cells. Its space and time complexity will be presented with a proof that no current algorithm can fully solve the problem.

Glossary

Term	Definition
Robot	An object that can move in its environment to perform a task.
Environment	The physical space in which the robot operates and navigates.
Wall	An object in the environment that the robot must avoid or navigate around.
Goal	The destination or objective that the robot is trying to reach.
Path	The route that the robot takes to reach its goal.
Pathfinding	The process of finding the path from the robot's current location to its goal while avoiding walls.
Map	The representation of the environment that the robot navigates in the form of a grid.
Graph	A collection of nodes (vertices) and edges that connect them.
Node	A point or vertex in a graph.
Edge	A line or connection between two nodes in a graph.
Tree	A connected acyclic graph with no cycles.
Root	The topmost node in a tree from which all other nodes descend.
Leaf	A node in a tree with no children.
Parent	The node that is connected to a child node.
Child	The nodes that are connected to a parent node.
Neighbor	Nodes that share the same parent.
Breadth-First Search (BFS)	A search algorithm that expands all children in one level at a time.
Depth-First Search (DFS)	A search algorithm that expands one child at a time, go deeply through that child and go back when reaching a leaf.
A Star (A*)	A search algorithm that does the expansion upon both the cost from root to the current node and the cost from that current node to the goal.

Greedy Best-First Search (GBFS)	A search algorithm that performs expansion upon the cost from the current node to the goal.
Bi-directional BFS	A variant of BFS algorithm that performs two separate BFS searches simultaneously, one from the source node and the other from the target node, until they meet in the middle.
Iterative Deepening A* (IDA*)	A memory-efficient variant of the A* search algorithm.
Multi-goal Path Finding (MGPF)	A pathfinding problem that visits all goal nodes from the start node.
Travelling Salesman Problem (TSP)	A problem that finds the shortest possible route that a salesman can take to visit a set of cities exactly once and return to the starting city
NP-hard	A class of computational problems that are at least as hard as the hardest problems in the nondeterministic polynomial time class
Informed search	A search algorithm type that uses heuristic functions to guide the search process towards the goal state.
Uninformed search	A search algorithm type that replies on simple strategies to explore the search space until a solution is found.
Heuristic function	A function that is used to estimate the "goodness" of a possible solution and to prioritize the exploration of solutions.
Manhanttan distance	A distance between two points that is calculated by the sum of the absolute differences of their x-coordinates and y-coordinates.
Optimal	An optimal solution is the shortest possible path from the starting point to the goal point
Completeness	The ability of an algorithm to find a solution if one exists

Figure 3 Glossary table

3. Search Algorithms

This section presents the qualities of six search algorithms used in my program. It also includes the discussion on which cases that a search algorithm would outperform others by using output collected from my program.

3.1 Search Algorithms Comparison based on the implementation:

	BFS	**DFS**	**Bi-directional BFS**	**GBFS**	**A***	**IDA***
Time complexity (in searching phrase only)	$O(kb^d)$	$O(kb^m)$	$O(kb^{m/2})$	$O(kb^m)$	$O(kb^d)$	$O(kb^d)$
Space complexity (in searching phrase only)	O(rc)	O(rc)	O(rc)	O(rc)	O(rc)	O(d)
Optimality	**Yes**	No	**Yes**	No	**Yes**	**Yes**
Evaluation Function	None	None	None	**f(n) = h(n)**	**f(n) = g(n) + h(n)**	**f(n) = g(n) + h(n)**
Past Knowledge	None	None	None	None	**g(n) is the cost so far**	**g(n) is the cost so far**
Completeness	**Yes**	No	**Yes**	No	**Yes**	**Yes**
Frontier Type	Queue	Stack/ Recursion	Two Queues	Priority Queue	Priority Queue	Recursion

Figure 4. Comparison table for search algorithms based on their implementations.

b is the branching; d is the depth of the shallowest solution; m is the maximum depth of the search tree; k is the number of goals; r is the number of rows in the map; c is the number of columns in the map

From the table, we can conclude the followings:

- All search algorithms consume the same amount of memory of O(rc), except for IDA*.

Most of algorithms are implemented with a 2D vector *path* and another 2D vector *direction* to store the result of final path. Meanwhile, IDA* does not use neither of them. Instead, two stacks of *path* and *direction* are used to record the solution.

IDA* solution-stored data structure:	Others' solution-stored data structure:
vector<Direction> lDirection; vector<Coordinate> lPath;	vector<vector<Direction>> direction; vector<vector<Coordinate>> path;

Note: only push_back() and pop_back() are used to manipulate data in IDA*.

- Time complexity for each search algorithm is shown in the table above. However, with the big O notation, the given stats are talking about the worst case for each algorithm. It also means that they may not reflect the actual computation time. In the subsection 3.2, we will discuss in which cases, one algorithm will stand out.

In a basic tree-search version, a problem only has one goal node, which causes a shorter runtime, compared to my implementation. In our problem, multiple k goal nodes will result in k times running time longer than the normal running time because we have to do goal check that walks through all goal nodes at every expansion. Therefore, most of algorithms experienced a multiplication with k.

In terms of informed search such as IDA*, A* and GBFS, with a good heuristic function that I used, the program only performs the search one time, even if there is an unreachable goal. Together with BFS and DFS, when they expand a goal node, the searching can stop immediately and return the solution. These search algorithms do not need a specific goal to reach at the compile time while they just keep searching until a solution is found. In short, their k factor in time complexity is the number of goals they have to check at every time they expand a node.

However, k factor in Bidirectional BFS carries different meaning. Unlike BFS and DFS, it does not need to check k goal nodes every time it expands a node. With a good implementation, when a goal is chosen for the robot to reach, Bidirectional BFS only checks whether the robot reaches that goal, which does not require a walk through all goal nodes at goal check stage. Meanwhile, when the chosen goal is unreachable, the program has to perform the BFS search all over again on the next un-processed goal. In such cases that the last chosen goal is the only one that is reachable, the program has to perform BFS k times on every goal node. That is also the worst case of Bidirectional BFS. In short, their k factor is the number of goals that they have to perform BFS on.

3.2 Search Algorithms Comparison by Output collected:

Based on preceding conclusions in 3.1, we are going to do some tests to challenge some of them, which are related to time complexity. We will run 16 same test cases for each algorithm and calculate their average running time. To ensure a thorough test, those 16 test cases are generated with a variety of wall locations, goal locations.

To perform this justification, some lines of code are added to compute the execution time:

```
auto start = std::chrono::high_resolution_clock::now();

auto end = std::chrono::high_resolution_clock::now();

auto duration
      = std::chrono::duration_cast<std::chrono::milliseconds>(end-start).count();

std::cout << "Execution time: " << duration << " ms" << std::endl;
```

Figure 5. Snippet code to set up clock

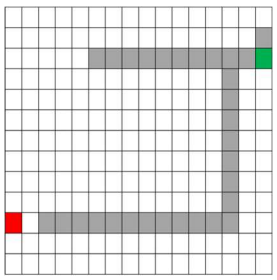
Figure 6. Test case 3.2.1

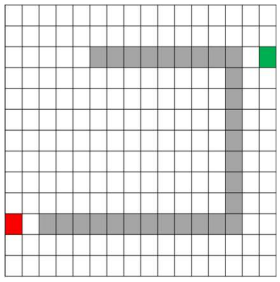
Figure 7. Test case 3.2.2

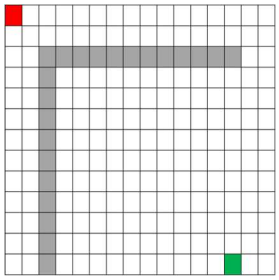
Figure 8. Test case 3.2.3

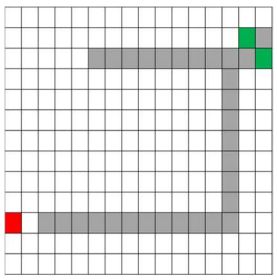
Figure 9. Test case 3.2.4

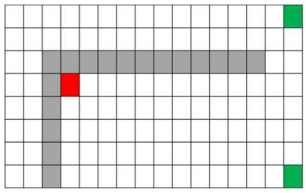
Figure 10. Test case 3.2.5

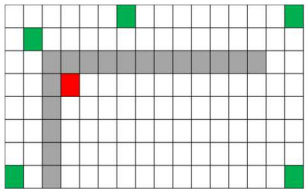
Figure 11. Test case 3.2.6

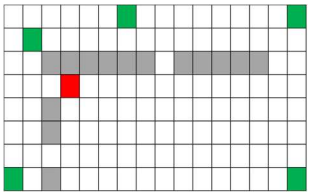
Figure 12. Test case 3.2.7

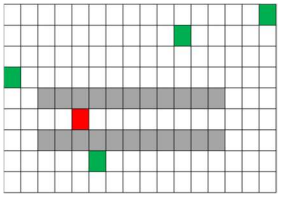
Figure 13. Test case 3.2.8

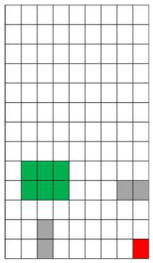
Figure 14. Test case 3.2.9

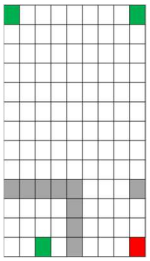
Figure 15. Test case 3.2.10

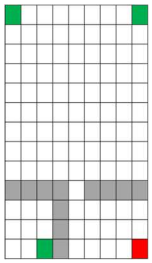
Figure 16. Test case 3.2.11

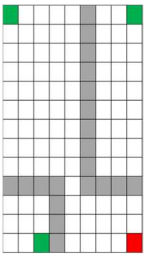
Figure 17. Test case 3.2.12

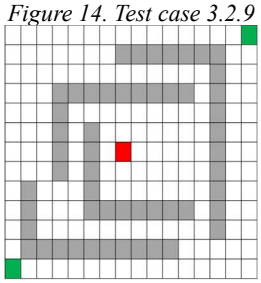
Figure 18. Test case 3.2.13

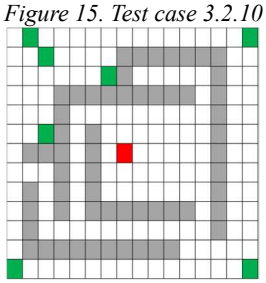
Figure 19. Test case 3.2.14

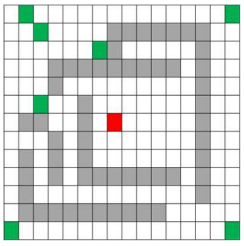
Figure 20. Test case 3.2.15

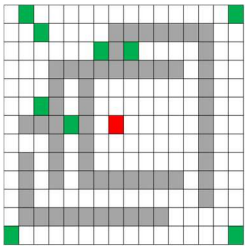
Figure 21. Test case 3.2.16

The following is the result we got from running those test cases:

Test 3.2	.1	.2	.3	.4	.5	.6	.7	.8	.9	.10	.11	.12	.13	.14	.15	.16	Avg exe. time
BFS	5	5	5	6	5	5	5	5	1	5	5	5	7	1	5	6	5.06
DFS	1	2	1	1	1	1	1	1	1	1	1	<1	1	1	1	1	1
Bi-BFS	3	3	4	3	2	2	2	3	2	2	6	6	4	4	4	4	3.25
GBFS	3	1	2	1	1	1	1	1	1	1	1	1	1	2	1	1	1.25
AS	3	3	2	3	1	1	1	1	1	1	2	1	2	2	2	4	1.43
IDA	7602	632	15	1	1	45	<1	<1	1	2	165	27	59	42	6	36	1079.23

Figure 22. Runtime result table

Run time is measured in millisecond. Each value is measured 10 times and its mode is taken as the final record.

Test 3.2	.1	.2	.3	.4	.5	.6	.7	.8	.9	.10	.11	.12	.13	.14	.15	.16	Avg path lgth
Shortest possible path	**25**	**25**	**27**	**23**	**15**	**15**	**4**	**6**	**8**	**14**	**20**	**20**	**19**	**13**	**8**	**14**	**17.25**
BFS	25	25	27	23	15	15	4	6	8	14	20	20	19	13	8	14	**17.25**
DFS	75	79	175	75	63	70	4	12	19	20	20	20	33	29	8	40	**92.75**
Bi-BFS	25	25	27	23	15	26	4	9	8	14	20	20	19	13	15	21	**17.75**
GBFS	25	25	27	23	15	16	4	6	8	20	20	20	26	13	8	14	**18.12**
AS	25	25	27	23	15	15	4	6	8	14	20	20	19	13	8	14	**17.25**
IDA	25	25	27	23	15	15	4	6	8	14	20	20	19	13	8	14	**17.25**

Figure 23. The solution returned from 16 test cases.

An example of running one test case: (Please note that execution time may vary upon different times of measurements because execution time is affected by other applications running in the background.)

```
D:\Swinburne\S1 2023\Intro to AI\A1\Code>103499232_clock.exe 1.2.txt BFS
1.2.txt BFS Execution time: 1 ms
196
UP UP UP UP UP UP UP UP RIGHT RIGHT RIGHT RIGHT RIGHT RIGHT RIGHT RIGHT RIGHT RIGHT RIGHT RIGHT RIGHT DOWN RIGHT

D:\Swinburne\S1 2023\Intro to AI\A1\Code>103499232_clock.exe 1.2.txt DFS
1.2.txt DFS Execution time: 2 ms
185
UP UP UP UP UP UP UP UP UP RIGHT DOWN DOWN DOWN DOWN DOWN DOWN DOWN DOWN DOWN DOWN LEFT DOWN DOWN RIGHT UP RIGHT UP RIGHT DOWN DOWN RIGHT UP UP RIGHT DOWN DOWN RIGH
T UP UP RIGHT DOWN DOWN RIGHT UP UP RIGHT DOWN DOWN RIGHT UP UP RIGHT DOWN DOWN RIGHT UP UP RIGHT DOWN DOWN RIGHT UP UP UP UP UP UP UP UP UP UP UP RIGHT DOWN DOWN

D:\Swinburne\S1 2023\Intro to AI\A1\Code>103499232_clock.exe 1.2.txt CUS1
1.2.txt CUS1 184
UP UP UP UP UP UP UP UP RIGHT RIGHT RIGHT RIGHT RIGHT RIGHT RIGHT RIGHT RIGHT RIGHT RIGHT RIGHT RIGHT RIGHT DOWN Execution time: 6 ms


D:\Swinburne\S1 2023\Intro to AI\A1\Code>103499232_clock.exe 1.2.txt GBFS
1.2.txt GBFS Execution time: 1 ms
55
UP UP UP UP UP UP UP RIGHT RIGHT RIGHT RIGHT UP RIGHT RIGHT RIGHT RIGHT RIGHT RIGHT RIGHT RIGHT RIGHT RIGHT DOWN RIGHT

D:\Swinburne\S1 2023\Intro to AI\A1\Code>103499232_clock.exe 1.2.txt AS
1.2.txt AS Execution time: 3 ms
174
UP UP UP RIGHT UP UP RIGHT UP UP RIGHT UP RIGHT UP RIGHT RIGHT RIGHT RIGHT RIGHT RIGHT RIGHT RIGHT RIGHT RIGHT DOWN RIGHT

D:\Swinburne\S1 2023\Intro to AI\A1\Code>103499232_clock.exe 1.2.txt CUS2
1.2.txt CUS2 257835
UP UP UP UP UP UP UP UP RIGHT RIGHT RIGHT RIGHT RIGHT RIGHT RIGHT RIGHT RIGHT RIGHT RIGHT RIGHT RIGHT RIGHT DOWN RIGHT Execution time: 632 ms
```

Figure 24. A screenshot shows an example of running a test case

In the scope of Robot Navigation Problem, examined by the preceding test cases:

Algorithm	Pros	Cons
BFS	It always yields the optimal path from start node to goal node. The returned solution is the shortest among all the possible paths. Run time is still acceptable with an average of 5ms.	It requires a lot of memory and runs with a longer execution time, which is as five as that of DFS.
DFS	It runs extremely fast, outperforms other algorithms in terms of runtime.	The solution is not optimal in most cases, which is even 5 times longer than the optimal solution.
Bi-directional BFS	It always yields the optimal solution in an acceptable execution time.	The optimal solution may not be the shortest possible path. Implementation is complex and space consumption is as twice as other normal algorithms. One of several goal grids must be specific to perform double BFS at the beginning of searching. If the chosen one is unreachable, it takes time for Bi-directional search to perform the search all over again from the side of goal node.

GBFS	It's efficient. Sometimes, the return path is the optimal one.	There is no guarantee that GBFS can return the optimal solution. Its goodness depends heavily on heuristic function.
A*	It always returns the shortest possible path in all cases in a relatively small amount of time with an average of under 2s.	Its goodness depends heavily on heuristic function. Implementation is tricky.
IDA*	It consumes little memory that only needs to store the path. Implementation is simple.	Execution time is extremely large. Its goodness depends heavily on heuristic function.

Figure 25. Summary table based on the results got from 20 test cases.

Discussion on which cases that a search algorithm would outperform others:

Test 3.2	Avg exe. time	Avg path length	Rank
BFS	5.06	17.25	2
DFS	1	92.75	6
Bi-BFS	3.25	17.75	4
GBFS	1.25	18.12	5
AS	**1.43**	**17.25**	**1**
IDA	1079.23	17.25	3

Figure 26. Summary result table for 6 search algorithms

Rank column is in the order of average path length, then average execution time.

In terms of runtime, DFS completely stands out when it comes to uninformed search while GBFS is the fastest informed search.

From the table 22, DFS outperforms BFS in all cases. When performing tree search, DFS is generally faster than BFS because DFS explores a path all the way to its deepest level before backtracking to explore other paths, while BFS explores all the nodes at a given depth before moving on to the next depth level. Furthermore, in DFS, if the goal state is deep down one path, DFS is likely to find it faster because it focuses on exploring one branch of the tree deeply, while BFS would explore all the branches at each level before reaching the goal node. In other words, DFS will run extremely fast when the goal node is far from the start node while BFS can perform better when the goal node is close to the start node. There is no specific case that BFS runs much faster DFS. In such cases that goal node is close to the start node, BFS may have a faster performance, but the gap is insignificant.

Bidirectional BFS runs faster than BFS because it reduces the search space by exploring both ends of the graph simultaneously, rather than searching the entire graph in a single BFS. This can significantly reduce the time requirements of the search, especially in our Robot Navigation Problem when the search space is large. However, as highlighted in table 25, the main disadvantage of Bi-directional BFS in this problem is that if the chosen goal is unreachable, it takes time for Bi-directional search to perform the search all over again from the side of goal node. Therefore, in some cases, it can produce the longer run time than a single BFS. In test case 11 and 12, for example, there are some goals that has been completely blocked by walls, which causes Bi-directional BFS took longer run time.

GBFS runs faster than A* in all test cases. GBFS only considers the heuristic value when selecting the next node to expand, while A* search considers both the heuristic value and the actual cost of reaching each node. Therefore, GBFS is faster than A* search in the scope of Robot Navigation Problem when we have a very large state space. At the same time, IDA* is slower than A* search in terms of time complexity, because IDA* explores the same nodes multiple times at different depth levels, while A* search explores each node only once. This means that IDA* has a higher time complexity than A* search for large and complex search spaces.

In terms of optimality, BFS completely stands out when it comes to uninformed search while AS is the best informed search.

Obviously, right from the implementation of BFS, we can see that BFS guarantees the solution found would be the optimal. Meanwhile, DFS focuses more on time complexity and exploring the entire tree, which disables DFS to return the optimal solution. At the same time, Bidirectional BFS, which is theoretically supposed to yield the same path with a traditional BFS, is not a good choice for our Robot Navigation problem. As point out above, at the beginning of searching, it must pick one of given goals and head to that goal. Although priority queue is used and ordered by the increasing values of Manhanttan distance, there is no guarantee that the path to the goal having the lowest Manhanttan distance is the shortest possible paths because that distance does not count the walls. However, when a chosen goal is reachable, Bi-directional BFS will definitely yield the shortest path from the start node to it.

Without considering the actual cost of reaching the current node, GBFS, a greedy algorithm, only considers the heuristic function that estimates the distance between the current node and the goal state. Therefore, GBFS may not find the optimal solution, as it can get stuck in local minima where the estimated cost to the goal state is low, but the actual cost to the goal state is high. AS, on the other hand, uses an evaluation function that combines the heuristic value with the actual cost of reaching the current node. This means that AS can find the optimal solution, as it considers both the heuristic value and the actual cost when selecting the next node to expand. All three informed searches are using the same heuristic function, which is the minimum distance from the start node to all goal nodes. In the scope of Robot Navigation Problem, this is a good heuristic function that can guarantee the returned path is the best possible path. In other words, unlike Bi-directional BFS, these informed searches only need to be triggered once during a test case to yield the desired solution.

In short, when all factors are considered, AS is the best algorithm for Robot Navigation. Not only does it ensure the shortest possible path, it also has a relatively small amount of running time. Though BFS can give the optimal solution, AS still dominates it with a faster running time, especially when the map size is large. Meanwhile, GBFS and DFS cannot guarantee the optimal solution, which eliminates them out of the race. Bi-directional BFS depends heavily on the goodness of the chosen node at the beginning of searching, which makes it unable to find the shortest possible path sometimes. All of them consume a lot of memory, while IDA* can still give the optimal solution without using that much amount of memory. However, its downside is an extremely long execution time taken to get the desired result.

4. Implementation

This section presents how each search program was implemented. Differences in implementing each approach are also discussed.

4.1 General Implementation:

The class diagram models the core classes and relationships in my program. The diagram includes abstract classes for SearchBase, UninformedSearch and Informed-Search, a separate class for every single search algorithm and a factory for creating "search" objects. A closer look at the class diagram reveals that all four pillars of OOP have been applied together with **a design pattern of Factory**.

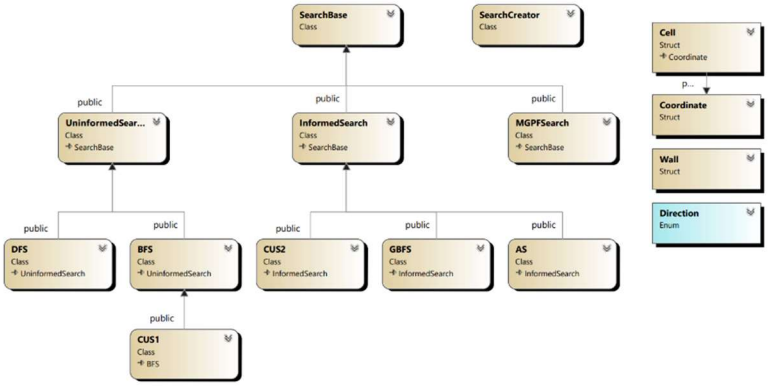

Figure 27. Class diagram for my program

In the class diagram, we have a SearchBase abstract class that serves as the base class for all search algorithms. All search algorithms inherit from this class and share the same method of file loading in the constructor of SearchBase. It contains private fields startGrid, goalArr, and path, direction to keep track of the starting node, list of goal nodes, resulting path and direction of the search, respectively. The search() method is abstract and must be implemented by each concrete subclass to perform the actual search algorithm. There are a lot of other methods implemented under this class that will be used later in our concrete subclass.

SearchCreator factory takes a string parameter searchType and another string parameter filepath and returns an instance of the appropriate search algorithm subclass based on the type.

```
class SearchCreator
{
public:

        virtual ~SearchCreator() {};
        SearchBase* GetSearchType(string searchType, string fp);
};

SearchCreator myFactory;
SearchBase* mySearch = myFactory.GetSearchType(searchType, filepath);
```

Figure 28. Snippet code for SearchCreator class

We then have other two abstract subclasses, which are UninformedSearch and InformedSearch. They would serve as base classes for different types of search algorithms that are categorized as uninformed (BFS, DFS, Bi-directional BFS) and informed (A*, GBFS, IDA*), respectively.

Finally, we have 5 concrete search algorithm subclasses: BFS, DFS, inheriting from UninformedSearch and others inheriting from InformedSearch. Meanwhile, CUS1, which is Bi-directional BFS, is also a child class of BFS. Each of these classes implements its own version of the search() method.

Besides, enum of direction is also used to represent the four directions. Then, we can create variables of the Direction type to store and manipulate the current direction of the robot.

4.2 Search Algorithm Implementation:

Despite the differences in the fringe data structure and how the fringe is updated during the search process, BFS, DFS, GBFS and AS all follow the same basic framework of visiting nodes and expanding the fringe until the goal is found.

BFS: fringe as **a queue**

```
bfs(start_node):
   queue = Queue()
   queue.put(start_node)
   while not queue.empty():
      current_node = queue.get()
      if isGoal(current_node): return true

      mark current_node as visited

      for neighbor in current_node.neighbors:
         if neighbor not in visited:
            queue.put(neighbor)
   return false
```

DFS: fringe as **a stack**/recursion

```
dfs(start_node):
   mark start_node as visited

   if isGoal(start_node):
      return true

   for neighbor in start_node.neighbors:
      if neighbor not visited:
         if dfs(neighbor):
            return true

   return false
```

GBFS:

AS: fringe as **a priority queue** using f(n) = h(n)+g(n)

```
fringe as a priority queue using f(n) = h(n)
gbfs(start_node, heuristic):
   open_list = PriorityQueue()
   open_list.put((heuristic(start_node),
start_node))
   while not open_list.empty():
      current_cost, current_node = open_list.get()
      if isGoal():
         return true
      mark current_node as visited

      for neighbor in current_node.neighbors:
         if neighbor not visited:
            neighbor_cost = heuristic(neighbor,
goal_node)
            open_list.put((neighbor_cost,
neighbor))
   return false
```

```
a_star(start_node, heuristic):
   open_list = PriorityQueue()
   open_list.put((0, start_node))

   while not open_list.empty():
      current_cost, current_node = open_list.get()
      if isGoal():
         return true
      mark current_node as visited

      for neighbor in current_node.neighbors:
         if neighbor not visited:
            neighbor_cost = current_cost +
current_node.get_cost(neighbor) + heuristic(neighbor,
goal_node)
            open_list.put((neighbor_cost, neighbor))

   return false
```

Figure 29. Pseudocode for BFS, DFS, GBFS and AS

In my implementation, I mostly followed the pseudocode presented above. However, for AS, instead of using visited 2D array for the close list, I implemented an 2D array to store the f value of every visited nodes in A* with the initial value of -1. It serves two main purposes: as a close list and as a way for me to update the f value in open list. To be specific, when f value of a node is -1, it means that that node has not been visited yet. Otherwise, it may be in close list or open list. In that case, if its f value is not better than the fresh f value, we can update the f value and add it to the open list again.

```
Bi-directional BFS
bidirectional_bfs(start_node, goal_node):
   queue_start = Queue()
   queue_goal = Queue()
   queue_start.put(start_node)
   queue_goal.put(goal_node)

   while not queue_start.empty() and not queue_goal.empty():
      current_node_start = queue_start.get()
      current_node_goal = queue_goal.get()
      if current_node_start visited in visited_goal: return true
      if current_node_goal visited in visited_start: return true

      mark current_node_start as visited
      mark current_node_goal as visited

      for neighbor_start in current_node_start.neighbors:
         if neighbor_start not in visited_start:
            queue_start.put(neighbor_start)
      for neighbor_goal in current_node_goal.neighbors:
         if neighbor_goal not in visited_goal:
            queue_goal.put(neighbor_goal)

   return false
```

```
IDA*
IDA(start_node):
   threshold=heuristic(start_node)
   while(True):
      tmp =search(start_node,0,threshold)
      if(temp==FOUND):  return true
      if(temp== -1): return false
      threshold=temp

Search(start_node, g, threshold):
   f=g+heuristic(node)
   if(f>threshold): return f
   if(node==Goal): return -1
   min=MAX_INT
   for neighbor in start_node.neighbors:
      temp = search(neighbor, g +
cost(start_node, neighbor), threshold);
       if(temp==FOUND): return 1
       min = min(min, tmp)
    return min
```

*Figure 30. Pseudocode for Bidirectional-BFS and IDA**

In my implementation, instead of implementing BFS all the way again in Bi-directional BFS, I put BFS as a base class from which Bi-directional BFS inherits and an additional BFS is initialized inside Bi-BFS to perform searching on the other end.

In terms of heuristic function, minimum Manhanttan distance from start node to every goal node is the one that I applied. I am confident with my choice of this heuristic function. It has been successfully dealt with every single test case and all produced the optimal result. The followings are several main reasons behind its extraordinary performance:

First, it is admissible, which means that it never overestimates the actual cost of reaching the goal state from the current state. The function calculates the distance to all the goal states and selects the minimum, guiding the search towards the nearest goal state and avoiding unnecessary exploration of the search space. This is a crucial property for any heuristic function in A* as it guarantees that the algorithm will always find the optimal path to the goal state.

Second, it is computationally efficient and easy to implement. The Manhattan distance can be easily calculated by adding the absolute differences in x and y coordinates, making it an ideal heuristic function for this problem.

Third, it is well-suited for our Robot Navigation Problem because it takes into account the constraints and characteristics of the environment. To be specific, in a grid-based navigation problem, the distance between two points is not just the Euclidean distance, but it also depends on the walls and the layout of the environment. On top of that, in this environment, the robot is only able to move in four directions, which also means that the diagonal moves are not allowed. In such cases, Euclidean distance would overestimate our actual distance between two cells.

4.3 Research Intuitive:

In my implementation for finding the shortest path to multiple goals (MGPF), there are two main parts:

1. Finding the shortest path to multiple goals MGPF by my own designed algorithm.

In the set-up phase:

i) A graph fGraph is generated using BFS to store the shortest length of path between each pair of goal nodes and from the start node to every goal node. This fGraph is stored as a 2D vector with the value of each element is the length of path calculated by BFS.

The first subvectors are used to store the shortest lengths between one goal to every other goals.

The last subvector in fGraph is used to store the shortest length between start node to every goal node.

From now on, we will be mainly working on this fGraph, instead of walls and goals because this fGraph has already taken the surrounding environment into account.

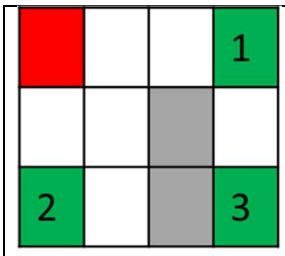

Figure 31. An example of a grid map to demonstrate the representation of fGraph

fGraph =
{{0,5,2}, //g1 to g1, g2, g3
{5,0,7}, //g2 to g1, g2, g3
{2,7,0}, //g3 to g1, g2, g3
{3,2,5}} //red to g1, g2, g3

```
setUpfGraph(start_node, goalArr):
   bfs_start = initialize BFS(start_node) with start_node as start node
   for i in [0,goalArr.size()):
      bfs_goal = initialize BFS(goalArr[i]) with goalArr[i] as start node

      for j in [i+1,goalArr.size()):
         if bfs_goal not visit goalArr[j]:
            trigger bfs_goal to search() from goalArr[i] to goalArr[j]

         len = bfs_goal.get_len(goalArr[j])   //from gArr[i] → gArr[j]
         add len to fGraph[i]
         add len to fGraph[j]
      // end for loop for j

      if bfs_start not visit goalArr[i]:
            trigger bfs_start to search() from start_node to goalArr[i]

      len = bfs_start.get_len (goalArr[i]) //from start node → goalArr[i]
      add len to fGraph[goalArr.size()]
```

Figure 32. Pseudocode to set up fGraph

ii) A computeHeuristicValue() is implemented to estimate the best node to be visited next.

To make the h value still admissible and more asymptotic to the actual value, every node experiences an adjustment value. For all the nodes that are not in the same row or columns with my current node, their h value will be increased by the value of 1. For all neighbors of the current node, if their neighbors have all eaten, that neighbor of the current node will have its h value decrement.

Please note that those actual shortest distances I mention below have been calculated and stored in fGraph.

h(n) = the actual shortest distance between two currently furthest green cells (let them be goal A, goal B) + the actual shortest distance from current node to the closer of those two green cells (goal A, goal B) + adjustment value

```
computeHeuristicValue (current_index, adjustment_value):
   lcost = -1
   goalA, goal B = 0
   for i in [0,goalArr.size()):   //walk through fGraph
      if goalArr[i] visited: continue

      for j in [i+1,goalArr.size()): // walk through fGraph to find the longest path
         if goalArr[j] visited: continue
         if (lcost < fGraph[i][j]): //save the local max in lcost
           lcost = fGraph[i][j]
           goalA, goal B = i, j //save the two current furthest nodes into goalA, goalB

   return fGraph[goalA][goalB] + min(fGraph[current_index][i],   fGraph[current_index][j]) + adjustment_value
```

Figure 33. Pseudocode to compute heuristic value

In main phase:

iii) The most significant part lies in search() method. This method runs infinitely until all goal nodes have been reached. To be specific, this loop visits the best goal node estimated by my f(n) in turn, marks that goal node as visited until isGoal() is true. When a goal node is on the way of the actual shortest path, BFS is triggered to print the path from the current start_node to that goal node.

```
search (start_node):
   i_at_min_f =  goalArr.size()
  current_start_node = start_node

   while (true):
      current_index = i_at_min_f
      if isGoal(): return true
      if isPrint():
         BFS.printPath(current_start_node,  goalArr[current_index])
         current_start_node = goalArr[current_index]

      if goal at current_index visited: continue
      min_f = 1e9
      mark goal node at current_index visited

      for i in [0,goalArr.size()): // walk through all goal nodes
         if goal node at i visited: continue
         f_current = fGraph[currentIndex][i] + computeHeuristicValue(i)
         if (min_f > f_current): //save the local min in min_f
            min_f = f_current
             i_at_min_f = i

   return false
```

2. Finding the length of the actual shortest path using Permutation with dynamic programming.

In the set-up phase:

i) We are going to reuse the fGraph generated in 1 but with a different format. An additional subvector is inserted at the top of fGraph to serve as the checker for visited nodes. The subvector that stores the path from start node to every goal nodes will be moved to be in the second subvector, right after the additional subvector.

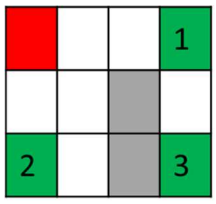

Figure 34. An example of a grid map

In 4.3.1:
fGraph = {{0,5,2},
{5,0,7},
{2,7,0},
{3,2,5}}

In 4.3.2:
fGraph = {{0,0,0,0},
{0,3,2,5},
{0,0,5,2},
{0,5,0,7},
{0,2,7,0}}

In main phase:

ii) We will try every possible path from start node to all goal nodes.

The for loop in executePermutation() serves the purpose of going from start node, then visiting all goal nodes in goalArr.

In func, first, we will check whether if only ith bit and 1st bit is set in our mask, it means that we have visited all other nodes already. If the current node has been visited, its value in memo will be returned.

The loop in func carries the similar purpose with that in executePermutation().

```
function func(i, mask):
  n = goalArr.size() + 1;
  if mask == ((1<<i) | 3)): return fGraph[1][i]
  if memo[i][mask] != 0: return memo[i][mask]
  res = 1e9
```

```
function executePermutation():
  n = goalArr.size() + 1;
  res = 1e9
  for i in [1,n]:
    res = min(res,
```

```
for j in [1,n]:
  if ((mask & (1<<j)) && j!=i && j!=1):
    res = min(res,
          func(j,mask&(~(1<<i)))+fGraph[j][i])
return memo[i][mask]=res
```

```
        func(i,(1<<(n+1))-1)))

print out res;
```

Figure 35. Pseudocode for func() and executePermutation

5. Features/Bugs/Missing

This section includes a list of the features that have been implemented. A list of known bugs is also outlined in this section.

5.1 Features of the program:

Overall, all search algorithms can handle variety types of input, even when the number of goal nodes is 0 or hundreds. The followings are the features implemented in the program:

- File Loading: a feature allows users to environments such as position of walls, the location of the robot's starting point, and the coordinates of the green cells that the robot needs to reach, number of rows, number of columns from external files.
- BFS: an algorithm can be used to determine the shortest path between the robot's starting location and its destination in a breadth-first manner.
- DFS: an algorithm can be used to determine the path between the robot's starting location and its destination in a depth-first manner.
- Bi-directional BFS (CUS1): an algorithm uses two simultaneous BFS searches to find the shortest path.
- GBFS: an algorithm evaluates nodes based on a heuristic function that estimates how close a node is to the goal, and then expands the node that is closest to the goal based on that estimate to find the path to a goal node.
- AS: an algorithm evaluates nodes based on two metrics: the actual distance from the starting node, and an estimate of the remaining distance to the goal node, as estimated by a heuristic function to find the shortest to a goal.
- IDA* (CUS2): an algorithm based on A* that performs multiple depth-limited searches with increasing depth limits until the goal node is found.
- MGPF: an algorithm to find the shortest path that walks through all green cells. To justify how good this path is, Permutation is applied to generate all possible solution to get the actual shortest path. However, this permutation is only triggered when the number of goals is less than 25. The reason for it is that if the number of goals is too high, we have to wait hundreds of years to get the result.

On the top of that, my algorithm that I designed to find the shortest path to all green cells may not always be able to identify the shortest path in every case. The proof that this problem is a variant of Travelling Salesman Problem (TSP) will be shown in section 6.

5.2 Bugs:

During this assignment, I have encountered hundreds of bugs and I have all of them fixed. The followings are some bugs that I have encountered during implementing the program:

No	Bugs	Reason	Solution
1	DFS did not return the path as same as the given solution.	The order of node expanding was different from the specific order in the instruction file when there are equal nodes.	Change the order to UP, LEFT, DOWN, RIGHT. Use Recursion instead of Stack.
2	Access to member that is out of range of a vector.	The way instruction file represents a grid was different from mine. I thought a grid at (x,y) would be accessed as grid[x][y].	Change the way represent a grid as grid[y][x].

3	Infinite loop when the input has some redundant blank lines at the end.	The program could not handle the case initially.	Use if-else, try catch.
4	LINK2005 Error	Collision in header files	Put the public functions in one highest header file. Put variable in class.
5	Wrong implementation of A star	Wrong understanding of how open list and close list work.	Update data structure. Strictly follow pseudocode.
6	Incorrect heuristic function for my research intuitive	Unknown	Replace it by my own designed algorithm.
7	Bi-directional BFS was implemented improperly	To solve Robot Navigation Problem, there must be modifications in the basic algorithm of Bi-directional BFS to handle several goal nodes. A queue for BFS at start node would be reused for all goal nodes. However, it was reused improperly, which led me to some bugs.	Debug. Dequeue a node from queue after performing goal check.
8	Incorrect Cycle detection of Minimum Spanning Tree	Wrong understanding of pseudocode	Debug and strictly follow pseudocode.
9	AS does not do prioritization as expected when there are 2 equal nodes	Priority Queue is built as a self balance tree, which will do swappings between elements when pop() is called.	When fcost is equal, do comparison on h value
10	Currently, this program cannot handle all invalid test cases.	To be worked on.	To be worked on.

Figure 36. Table of known bugs

To handle these bugs, I also used GitHub as my version control so that I could roll back to when I wanted.

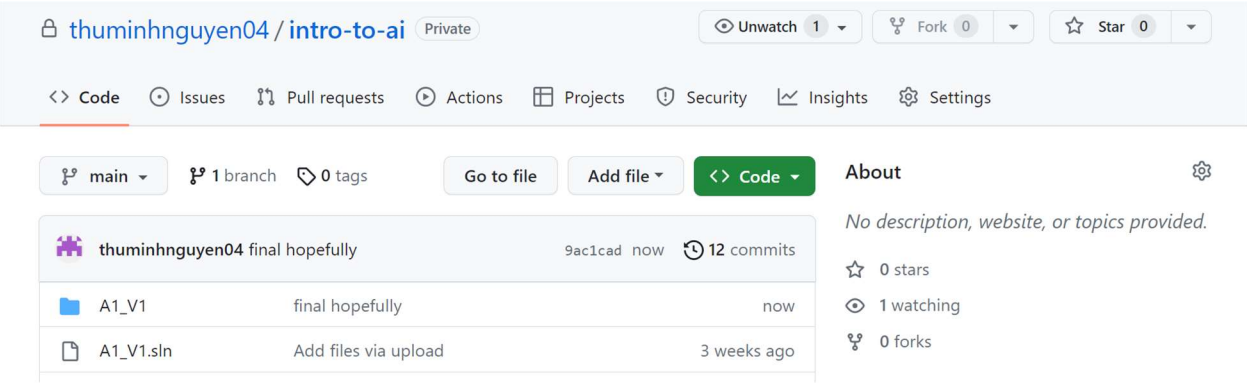

Figure 37. Screenshot of my GitHub for this assignment

6. Research

In this section, MGPF stands for the problem that robot has to visit all goal nodes starting from its initial position.

The issues I encountered while doing this requirement:

- The current heuristic function used in basic informed search algorithms cannot be used for MGPF. Meanwhile, Minimum Spanning Tree (MST) seems to be a good heuristic for this problem. The idea behind using MST as a heuristic in A* is to estimate the remaining cost of reaching the goal node by computing the minimum spanning tree of the remaining part of the graph that has not been explored yet. Therefore, the length of the shortest path connecting all the goal nodes must be between T and 2T, which ensure the admissibility for the heuristic function. In short, h(n) can be computed as the following formula: h(n) = Manhanttan(current_node, closest_goal) + MST (unvisited_goals). However, it turned out that this h(n) is not good at all.

Let's take the map below as an example to clarify this claim. To get the shortest path in this map, robot must head left to visit green 1 → 4 at its very first step. From the table below, we can see that using MST will yield green 5 as our first goal cell to be visited. This is against what we had expected. Therefore, MST is not the good choice for our MGPF.

Cell	h(n)	g(n)	f(n)
1	51	10	61
2	50	9	59
3	52	6	58
4	53	3	56
5	**52**	**2**	**54**
6	51	5	56
7	52	6	58
8	50	7	57
9	51	28	79
10	51	29	80
11	50	32	82
12	51	31	82
13	51	30	81

Figure 38. Table to show the result of computing f(n), g(n), h(n)

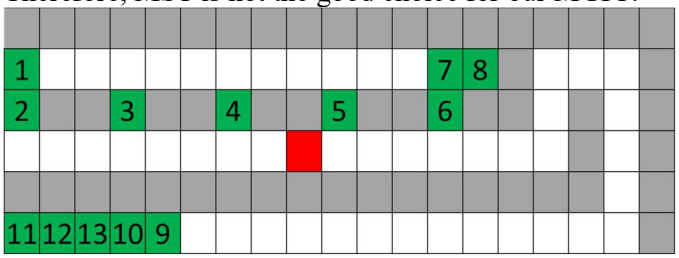

Figure 39. Map to illustrate the efficiency of MST

- A*, which is known as the best search algorithm in Robot Navigation Problem, cannot be used to find the shortest path through all goal nodes, regardless of how good the heuristic function is. The reason for this is that A* relies on the assumption that each node has a unique parent, which allows the algorithm to keep track of the optimal path from the starting node to each node in the search space.

Let's take the map below as an example to clarify this claim. Obviously, a closer look at the map reveals that there is no way that we can go through all green cells without re-visiting a node. The shortest path would be from starting position to green 2, then green 1, and finally green 3. This path visits white 4 two times: down from red cell and up from green 2. Therefore, a child node may have multiple parents in MGPF. In such cases, it becomes impossible to determine which path to follow to reach the goal nodes. When a child node has multiple parents, there is no longer a clear way to determine the optimal path to that node, as there may be multiple paths that lead to it.

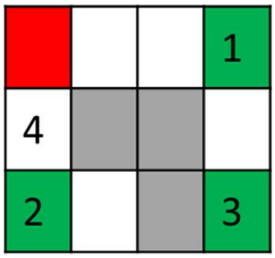

Figure 40. Map to illustrate the inappropriate of A in MGPF*

→ Therefore, instead of treating the current environment as normal, I put the shortest distance between every two goal nodes and the distance between each of them and start node in a graph. The edges among goal nodes will be 2-way with the same value of cost while the edge from start node to every goal node will be one way with the value of the shortest path length between them. From now on, the only thing I worry about is the freshly generated graph.

Proof that MGPF is a variant of TSP:

- When having a graph that stores the shortest path length of all nodes of interest, my problem becomes finding the shortest possible path that a robot can take to visit a set of vertices, while visiting each vertex exactly once. Why is there a constraint of visiting a vertex exactly once? Because if not, the solution found is not the optimal

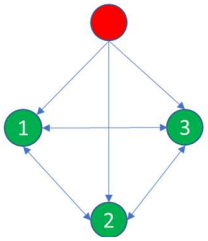

Figure 41. The transformation of our MGPF from a grid map into a graph

one. As my graph carries the shortest path length between any two vertices, it means that there is always a direct edge between any two vertices. Therefore, due to triangle inequality, it will cost us more time if we go back to one vertex that has been already visited. The illustration can give you a clearer understanding of transforming a typical grid map figure 40. into a graph.

- When edges between every goal node to our start node are added with a cost value of 0, our MGPF has finally turned into Travelling Salesman Problem. TSP is the problem that finds the shortest possible route that a salesman can take to visit a set of cities and return to the starting point, while visiting each city only once. This problem is NP-hard because it can be reduced to the Hamiltonian cycle problem, which is known to be NP-hard. The illustration can give you a clearer understanding of the extra edges added.

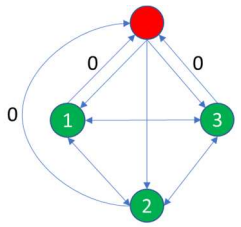

Figure 42. The image shows how the graph looks like after adding some 0-value edges

In the case of MGPF problem, the difficulty arises from the fact that the problem space is often large and complex, with many possible paths that the robot can take. There is no general algorithm that can solve it optimally for all cases at the moment.

Some additional test-cases together with test cases in section 3.2 is used to justify the goodness of the proposed algorithm. Some test cases are excluded because of the invalid input.

No	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Test case identifier	3.2.1	3.2.2	3.2.3	3.2.4	3.2.5	3.2.6	3.2.7	3.2.8	3.2.13	3.2.15	3.2.16	17	18	19	20	21	22	23	24	25
My length	25	25	28	69	22	45	38	**38**	46	60	**110**	106	28	162	27	34	60	**232**	7	**320**
Actual h	25	25	28	69	22	45	36	**36**	46	60	**93**	106	28	162	27	34	60	**198**	7	**280**

Figure 43. Result table for testing MGPF algorithm

In 20 test cases, there are 16 cases that my algorithm returns the optimal path.

In the 4 sub-optimal cases, my path exceeds the shortest solution an average of: 5.56%, 18.28%, 17.17% and 14.29% for test case 3.2.8, 3.2.9, 23 and 25 respectively.

7. Conclusion

When it comes to the best type of search algorithm, I would say informed search. Informed search dominates its counterpart by offering better time efficiency and flexibility. The pure of informed search, heuristic functions, can help to guide the search towards the goal node, reducing the number of nodes visited during the search.

When it comes to the best search algorithm for Robot Navigation Problem, I would say A Star with a good heuristic function. With a good heuristic function, A* can prune large portions of the search space and stay focus on the most promising paths towards the goal. In this problem where there are several goal nodes, if the chosen heuristic function only takes one goal node into account at a time, I believe an uninformed search such as BFS can easily outperform A*. However, my heuristic function of minimum distance from the start node to every goal node makes A* supremely stand out.

Due to Robot Navigation Problem's NP-hardness and similarity to the Travelling Salesman Problem, it is unlikely that a general algorithm can be developed to solve all instances of the problem. My own proposed algorithm has found the optimal path in most cases while in other cases, it can still yield a good suboptimal path with the cost exceeding under 20% of the actual shortest length.

For this assignment, I believe I already give my best on improving them so there are not much further improvements left to be made. With the restriction on page limit, I cannot present the correctness of my proposed algorithm. That's one of my regrets. Besides, if I had more time, I would re-factor my code and design a better data structure for my program. For example, in most of search algorithms, I used a 2D vector to track the visited nodes while a linked list with a hash map may help reduce space and time complexity. Basically, most of search algorithms follow the same basic framework of visiting nodes and expanding the fringe, I would to re-design the code so that I only need to implement one single search() method that accepts different types of data structure of fringe.

8. Acknowledgements/Resources

I would like to express my sincere appreciation to John DeNero, Dan Klein, Pieter Abbeel for creating The Pac-Man Projects as a valuable reference and source of inspiration for my project. Your related work provided me with important insights and ideas that helped me to develop my own approach and understand much more about search algorithms.

I also would like to express my sincere gratitude to my tutor for the invaluable advice and guidance provided during the completion of my assignment. Your insights and recommendations were instrumental in helping me to understand the material and improve the quality of my work. Thank you for your support and encouragement throughout the process.

Finally, I would like to thank the StackOverflow community for providing a valuable resource where I can search for solutions to problems encountered during my project. Your contributions have been instrumental in helping me to overcome difficult challenges and find effective solutions. Thank you for your willingness to share your knowledge and expertise with the broader community, and for being a reliable source of support for developers like me.

9. References

[1] Russell, S.J., Norvig, P. and Davis, E. (2022) Artificial Intelligence: A modern approach. Harlow, England: Pearson Educación.

[2] Red Blob Games. (n.d.). A* Pathfinding for Beginners. Accessed April 1, 2023, from https://www.redblobgames.com/pathfinding/a-star/introduction.html

[3] GeeksforGeeks. (2021, March 4). A* Search Algorithm. Accessed April 1, 2023, from https://www.geeksforgeeks.org/a-search-algorithm/

[4] AI Shack. (n.d.). A* Algorithm. Accessed March 27, 2022, from http://www.aishack.in/tutorials/implementation-a-star-search/

[5] Gerhardt, M. (2018). Dynamic Programming in Python: A Beginner's Guide. Accessed from https://medium.com/@mohitgera1994/dynamic-programming-in-python-a-beginners-guide-a4542a62be18

[6] LeetCode. Shortest Path Visiting All Nodes. Accessed 21 April 2023 https://leetcode.com/problems/shortest-path-visiting-all-nodes/description/

[7] AI Shack. Greedy Best-First Search (GBFS). Accessed 21 April 2023 http://www.aishack.in/tutorials/greedy-best-first-search-gbfs/

[8] GeeksforGeeks. (n.d.). Greedy Best First Search (GBFS) Algorithm. Retrieved from https://www.geeksforgeeks.org/greedy-best-first-search-gbfs-informed-search/

[9] AI Shack. (n.d.). A* Pathfinding with Minimum Spanning Trees. Retrieved from http://www.aishack.in/tutorials/minimum-spanning-trees-a-star-pathfinding/

[10] Nitschke, G., & Kötter, M. (2011). Combining A* and Minimum Spanning Tree for Pathfinding in Games. In Proceedings of the International Conference on Entertainment Computing (pp. 323-328). doi: 10.1007/978-3-642-24500-8_34

[11] Bhuiyan, M. N. H., Islam, M. M., & Lee, M. (2012). Combining A* Algorithm with Minimum Spanning Tree for Shortest Path Finding in Large Scale Graphs. In Proceedings of the 2012 IEEE/ACIS 11th International Conference on Computer and Information Science (pp. 241-246). doi: 10.1109/ICIS.2012.30

[12] Marinakis, Y., & Marinaki, M. (2017). A* Algorithm and Minimum Spanning Tree: A Comparative Study. In Proceedings of the 6th International Conference on Modern Circuits and Systems Technologies (pp. 1-5). doi: 10.1109/MOCAST.2017.7953528

[13] Gao, S., & Xia, Y. (2018). An Improved A* Algorithm Based on Minimum Spanning Tree. In Proceedings of the 2018 4th International Conference on Control, Automation and Robotics (pp. 71-75). doi: 10.1109/ICCAR.2018.8381981