Student Name: Thanh Tam Vo
Student ID: 103487596

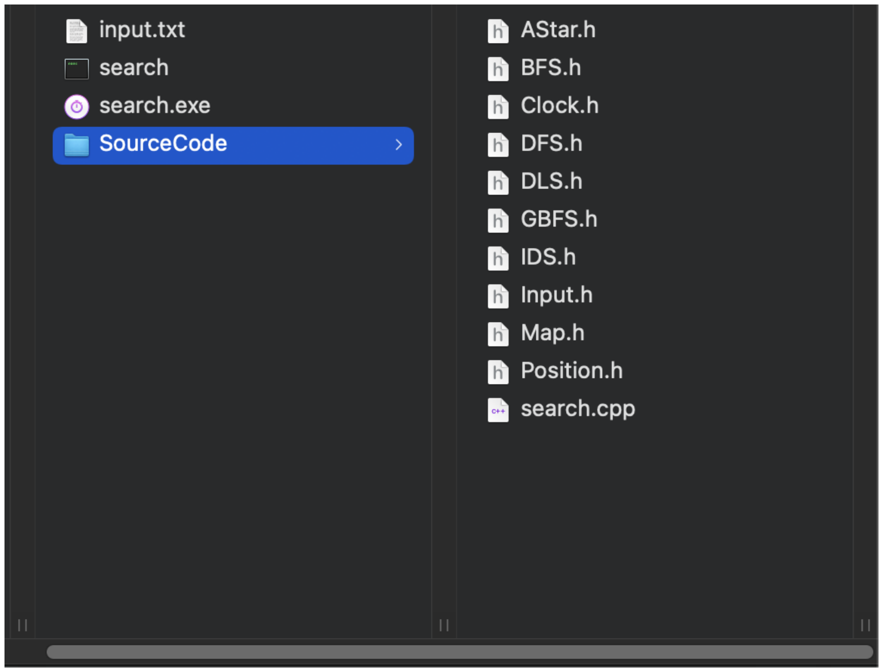
COS30019 - Introduction to Artificial Intelligence
Assignment 1

Contents

1 Program Instruction

1.1 File Organization

The submission contains a **.exe** file for Microsoft Windows OS, a **.exec** file for MacOS, a **.txt** file for data manipulation, and a folder containing source codes for the program as Figure 1 below:



input.txt
search
search.exe
SourceCode

AStar.h
BFS.h
Clock.h
DFS.h
DLS.h
GBFS.h
IDS.h
Input.h
Map.h
Position.h
search.cpp

Figure 1: File Organization

1.2 MacOS User

For those who use MacOS systems, here is the command line to run the program on the terminal

```
Submission % ./search input.txt BFS
```

The search algorithm can be chosen by replacing 'BFS' with 'DFS', 'GBFS', 'ASTAR', and 'IDS'. Note that if the algorithm is Iterative deepening depth-first search (IDS), the command line should be followed by the number of iterations as below

```
Submission % ./search input.txt IDS 17
```

1.3 Microsoft Windows 11 User

For those who use the Microsoft Windows Operating Systems, especially Windows 11, here is the command line to run the program in the **cmd**

```
PS D:\Desktop\Submission> ./search input.txt BFS
```

1.4 Data manipulation

The input data may be changed by altering the **input.txt** file, but first, we must unify the Coordinate System for our Robot Navigation problem.

1.4.1 Coordinate Systems Define

Suppose that we have a map as below:

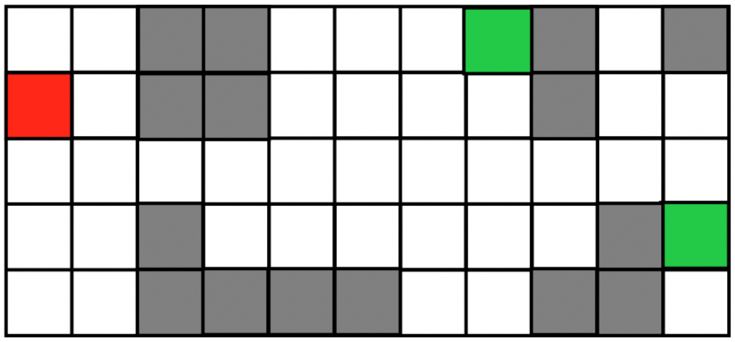


Figure 2: Default map

where the Red cell is the source of the Robot, the Green cell is the Destination, and the Grey cell is the obstacle/wall, which means that the Robot can not go through this cell. The position of each cell follows the rule of 2D Array, for example, in Figure 2 above, the coordinate of the Red cell is **(1, 0)**.

Overall, given a map with width and height, the position of a cell is (x, y), where $0 \leq x <$ height and $0 \leq y <$ width.

1.4.2 Data manipulation

Here is the default **input.txt** file:

```
5 11
1 0
3 10
0 2 2 2
3 2 2 1
4 3 1 3
0 8 2 1
0 10 1 1
4 8 1 1
3 9 2 1
```

Listing 1: input.txt

The first three lines are the basic information of the map:

- The first line contains 2 integers: height and width of the map, respectively
- The second line contains the position of the Source of the Robot
- The third line contains the destination for the Robot to find out the path

From the fourth line, each line contains 4 integers n, m, k, q where (n, m) is the coordinate of the leftmost top corner of a wall, k and q are the height and width of that wall, respectively. By following this rule, the user can modify the map in the **input.txt** file.

2 Introduction

Robot Navigation is one of the ideal problems in Artificial Intelligence. To solve this problem, human creates an Agent that has the ability to find out the path from a given source to the destination by 2 major searching techniques: uninformed and informed search. This report aims to explain these search methods also giving the implementation in programming language **C++**. Also, this report will cover research about the mechanism of the priority queue, which reduces the time complexity for heuristic search.

3 Search Algorithm and Its Implementation

3.1 Breadth-first Search (BFS)

3.1.1 BFS Idea

In this Robot Navigation problem, the algorithm aims to explore other cells horizontally, which means that it starts visiting the source cell, then expanding its neighbor, and again, expanding the neighbors of each neighbor cell and so on until all cells are visited. To achieve this idea, the algorithm uses a queue to store the waiting cells. Frankly speaking, given a tree graph with infinite depth, the BFS algorithm can find out the goal at the d^{th} depth of the graph. Here is the BFS pseudocode to solve our Robot Navigation problem.

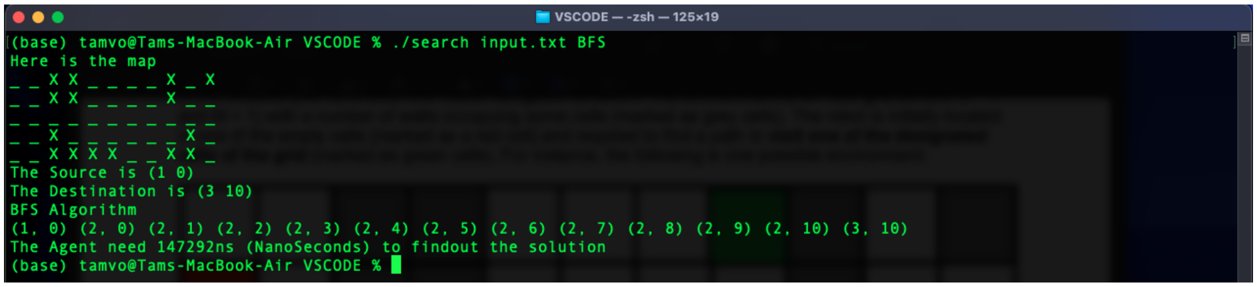
```
cell ← Source
queue ← push cell to the frontier
visited visited list
while queue is not empty do
  currentCell ← pop queue
  Append currentCell to visited
  for cell in neighbours of currentCell do
    if cell is not in visited then
      queue ← push cell to the queue
    end if
  end for
end while
```

Figure 3: Pseudocode for BFS algorithm

3.1.2 Quality

Since the algorithm will explore cells horizontally, in other words, it will expand all the cells, therefore the algorithm requires a large amount of memory to be allocated. However, it still guarantees that it always finds the shortest path from the goal to the target.

In my implementation with the default input, the BFS algorithm takes 147,292 nanoseconds to find out the solution as Figure 4, since our problem is simple, I decided to use nanoseconds to measure the performance, other algorithms such as DFS, GBFS, etc. will be used nanosecond to measure too.



```
VSCODE — -zsh — 125×19
(base) tamvo@Tams-MacBook-Air VSCODE % ./search input.txt BFS
Here is the map
_ _ X X _ _ _ _ X _ X
_ _ X X _ _ _ _ X _ _
_ _ _ _ _ _ _ _ _ _ _
_ _ X _ _ _ _ _ _ X _
_ _ X X X X _ _ X X _
The Source is (1 0)
The Destination is (3 10)
BFS Algorithm
(1, 0) (2, 0) (2, 1) (2, 2) (2, 3) (2, 4) (2, 5) (2, 6) (2, 7) (2, 8) (2, 9) (2, 10) (3, 10)
The Agent need 147292ns (NanoSeconds) to findout the solution
(base) tamvo@Tams-MacBook-Air VSCODE %
```

Figure 4: Search result using BFS

3.2 Depth-first Search (DFS)

3.2.1 DFS Idea

The algorithm works the same as BFS, however, cells will be explored vertically, which means that it starts visiting the source, then expands a neighbor cell, and expand a neighbor of that neighbor cell, and so on until it reaches the limitation. It is easy to see that for a problem with infinite depth, DFS will never find out the solution. To achieve this idea, the algorithm uses a stack to store the waiting cells. The pseudocode to solve our problem using DFS in Figure 5 below:

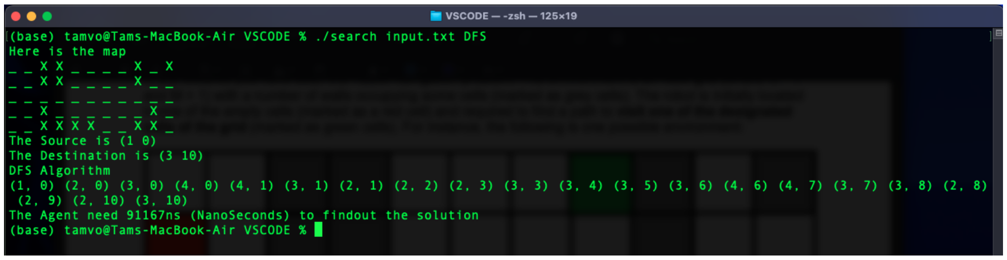
```
cell ← Source
stack ← push cell to the frontier
visited visited list
while stack is not empty do
  currentCell ← pop stack
  Append currentCell to visited
  for cell in neighbours of currentCell do
    if cell is not in visited then
      stack ← push cell to the stack
    end if
  end for
end while
```

Figure 5: Pseudocode for a DFS algorithm

3.2.2 Quality

As we discussed above, DFS can not return the solution for a tree graph with infinite depth. Supposed that our problem has a limited depth, DFS can find out the solution, but it does not guarantee that the solution is optimal. My implementation as Figure 6 will prove that, although the taken time is less than BFS, the path is not optimal.

In my perspective for this simple problem, I prefer the optimal solution rather than the taken time, so DFS is not the best choice for me.



```
(base) tamvo@Tams-MacBook-Air VSCODE % ./search input.txt DFS
Here is the map
_ _ X X _ _ _ _ X _ X
_ _ X X _ _ _ _ X _ _
_ _ _ _ _ _ _ _ _ _ _
_ _ X _ _ _ _ _ _ X _
_ _ X X X X _ _ X X _
The Source is (1 0)
The Destination is (3 10)
DFS Algorithm
(1, 0) (2, 0) (3, 0) (4, 0) (4, 1) (3, 1) (2, 1) (2, 2) (2, 3) (3, 3) (3, 4) (3, 5) (3, 6) (4, 6) (4, 7) (3, 7) (3, 8) (2, 8)
 (2, 9) (2, 10) (3, 10)
The Agent need 91167ns (NanoSeconds) to findout the solution
(base) tamvo@Tams-MacBook-Air VSCODE %
```

Figure 6: Searching result using DFS

3.3 Iterative Deepening depth-first Search (Custom Search 1)

The Iterative Deepening depth-first Search (IDS) solves the problem of DFS: infinite depth graph, before having a deep understanding about IDS, let's take a look at Depth Limited Search (DLS)

3.3.1 Depth Limited Search (DLS)

The idea behind DLS is the same as DFS, however, in case it reaches the given limit depth of the graph, the algorithm will stop. Supposed the target is at level h^{th}, but the limitation is level $(l)^{th}$, where $l < h$, the agent will not find out the solution.

Here is the pseudocode for DLS:

```
$(cell, level) \leftarrow (Source, 0)$
$limit \leftarrow$ assign limitation
$stack \leftarrow$ push $(cell, level)$ to the frontier
$visited$ visited list
**while** $stack$ is not empty **do**
  $currentCell \leftarrow$ pop $stack$
  **if** $currentCell.cell$ is target **then**
    return Solution
  **end if**
  Append $currentCell$ to $visited$
  **for** $cell$ in neighbours of $currentCell$ **do**
    $nextLevel \leftarrow currentCell.level + 1$
    **if** $cell$ is not in $visited$ and $nextLevel < limit$ **then**
      $stack \leftarrow$ push $(cell, nextLevel)$ to the $stack$
    **end if**
  **end for**
**end while**
```

Figure 7: Pseudocode for DLS algorithm

3.3.2 IDS Idea

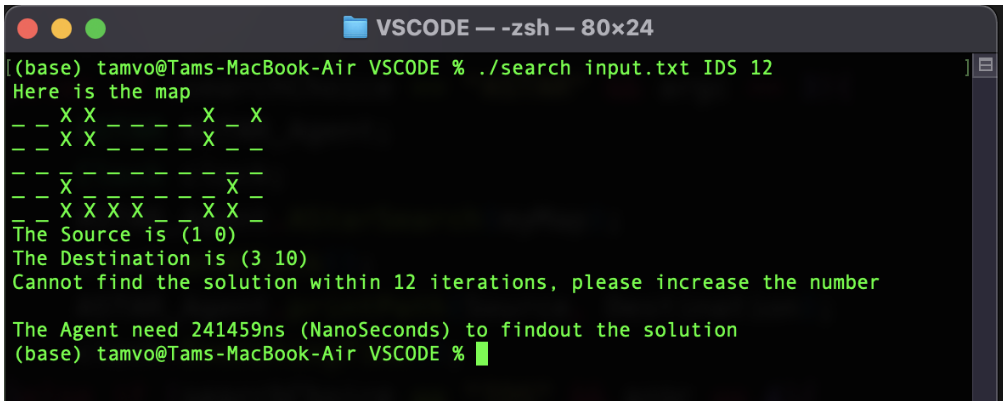
The DLS algorithm can find out the solution as long as the limitation is greater than the depth of the target, so the IDS algorithm will try all the limitation depth from 0 to n, for $n \geq 0, n \in \mathbb{N}$. Here is the pseudocode for Iteravite Deepening depth-first Search:

```
limitation ← assign limitation
for limit in range(0, limitation) do
    DLS(graph, target, source , limit)
end for
```

Figure 8: Pseudocode for IDS algorithm

3.3.3 Quality

The solution is based on the idea of DFS, so it is not optimal. Plus, the algorithm still depends on a factor: the number of iterations, or the limitation. As Figure 9 below, with 12 iterations, the path can not be found:

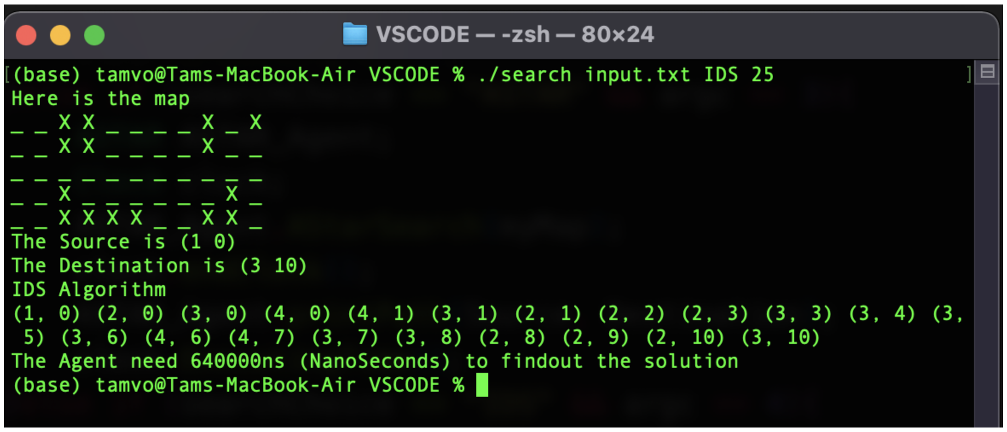


```
(base) tamvo@Tams-MacBook-Air VSCODE % ./search input.txt IDS 12
Here is the map
_ _ X X _ _ _ _ X _ X
_ _ X X _ _ _ _ X _ _
_ _ _ _ _ _ _ _ _ _ _
_ _ X _ _ _ _ _ _ X _
_ _ X X X X _ _ X X _
The Source is (1 0)
The Destination is (3 10)
Cannot find the solution within 12 iterations, please increase the number

The Agent need 241459ns (NanoSeconds) to findout the solution
(base) tamvo@Tams-MacBook-Air VSCODE % 
```

Figure 9: Searching result using IDS with 12 iterations

Now this is the terrible result after increasing the iteration to 25:



```
(base) tamvo@Tams-MacBook-Air VSCODE % ./search input.txt IDS 25
Here is the map
_ _ X X _ _ _ _ X _ X
_ _ X X _ _ _ _ X _ _
_ _ _ _ _ _ _ _ _ _ _
_ _ X _ _ _ _ _ _ X _
_ _ X X X X _ _ X X _
The Source is (1 0)
The Destination is (3 10)
IDS Algorithm
(1, 0) (2, 0) (3, 0) (4, 0) (4, 1) (3, 1) (2, 1) (2, 2) (2, 3) (3, 3) (3, 4) (3,
 5) (3, 6) (4, 6) (4, 7) (3, 7) (3, 8) (2, 8) (2, 9) (2, 10) (3, 10)
The Agent need 640000ns (NanoSeconds) to findout the solution
(base) tamvo@Tams-MacBook-Air VSCODE % 
```

Figure 10: Searching result using IDS with 12 iterations

With those uninformed search techniques above, I prefer BFS algorithm to find out the solution despite its costly space required.

3.4 Greedy Best-first Search (GBFS)

3.4.1 Idea

The idea of GBFS is the same as BFS, however, instead of using a queue to store the waiting cells, the algorithm uses a priority queue and a heuristic function $h(x)$ to evaluate the current cell. In our Robot Navigation problem, the heuristic function is the Manhattan method. Below is the Manhattan method which takes the positions of two cells: the current cell and the goal cell:

```cpp
#include <iostream>
using namepace std;

int Manhattan(Position x, Postion Goal){
    return (abs(x.row - Goal.row) + abs(x.col - Goal.col));
}
```

Listing 2: Manhattan Function in C++

For example, supposed the current cell is $(1, 0)$ and our goal is $(3, 10)$. The next cells that will be explored are $(0, 0)$, $(2, 0)$, and $(1, 1)$, by passing them to the Manhattan heuristic function, the values we will get are 13,11, and 11, respectively. Next, we push cells $(0, 0)$, $(2, 0)$, and $(1, 1)$ to the priority queue, if we pop a cell from the queue above, we can get $(1, 1)$ or $(2, 0)$ because it seems that those cells are closer to the goal and more potential to find the path.

Here is the pseudocode for GBFS:

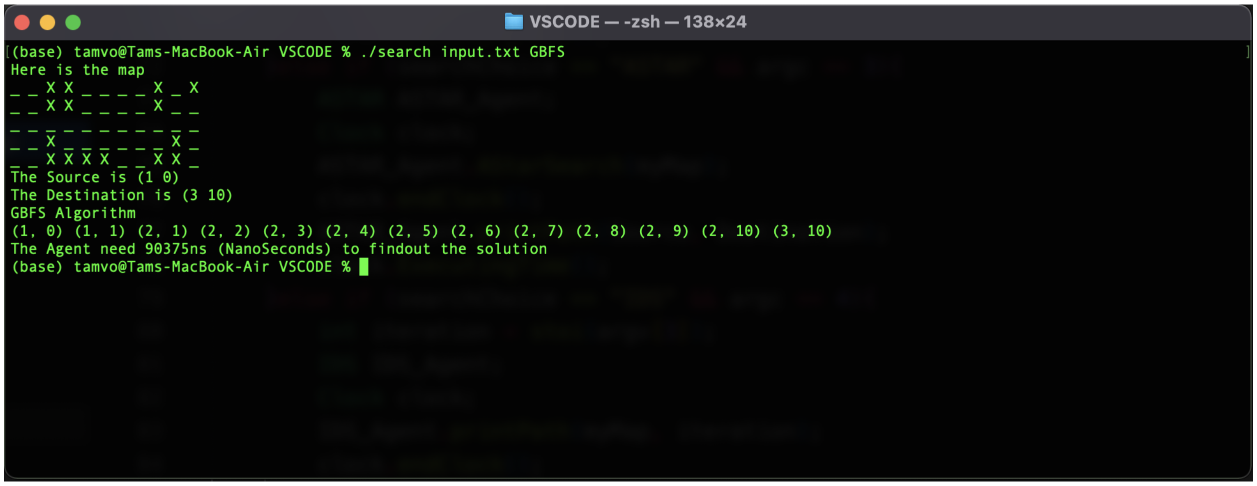
```
$cell \leftarrow Source$
$priotityQueue \leftarrow$ push $cell$ to the frontier
$visited$ visited list
while $priorityQueue$ is not empty do
  $currentCell \leftarrow$ pop $priorityQueue$
  if $currentCell$ is goal then
    return solution
  end if
  Append $currentCell$ to $visited$
  for $cell$ in neighbours of $currentCell$ do
    if $cell$ is not in $visited$ then
      $priotiyQueue \leftarrow$ push $cell$ to the $priorityQueue$
    end if
  end for
end while
```

Figure 11: Pseudocode for GBFS algorithm

3.4.2 Quality

GBFS algorithm performance depends on the heuristic function, we can use the Euclidean method as a heuristic function but it is not optimal as the Manhattan method. My result after using GBFS to find the solution is in Figure 12, the result is better than other search methods.

It seems that the agent will quickly find out the solution with a heuristic search, although the path is not better than BFS, which always gives the optimal path, the performance of GBFS is still impressive.



```
VSCODE — -zsh — 138×24
(base) tamvo@Tams-MacBook-Air VSCODE % ./search input.txt GBFS
Here is the map
_ _ X X _ _ _ _ X _ X
_ _ X X _ _ _ _ X _ _
_ _ _ _ _ _ _ _ _ _ _
_ _ X _ _ _ _ _ _ X _
_ _ X X X X _ _ X X _
The Source is (1 0)
The Destination is (3 10)
GBFS Algorithm
(1, 0) (1, 1) (2, 1) (2, 2) (2, 3) (2, 4) (2, 5) (2, 6) (2, 7) (2, 8) (2, 9) (2, 10) (3, 10)
The Agent need 90375ns (NanoSeconds) to findout the solution
(base) tamvo@Tams-MacBook-Air VSCODE %
```

Figure 12: Searching result using GBFS

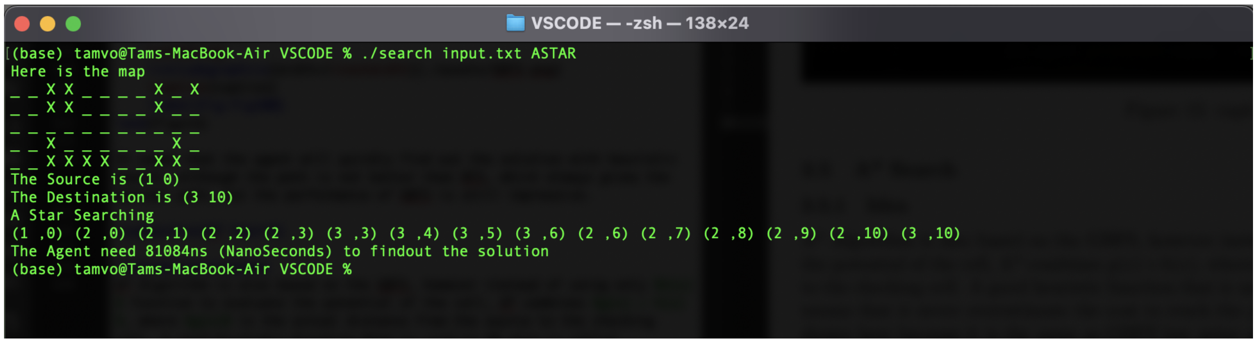
3.5 A* Search

3.5.1 Idea

A* Algorithm is also based on the GBFS, however instead of using only the $h(x)$ function to evaluate the potential of the cell, A* combines $g(x) + h(x)$, where $g(x)$ is the actual distance from the source to the checking cell. A good heuristic function in A* star is called "Admissible", which means that it never overestimates the cost to reach the goal. Pseudocode for A* Search will not be shown here because it is the same as GBFS but uses a different evaluation function.

3.5.2 Quality

In our problem, the distance between two adjacent cells is always 1, therefore, A* Searching will not return the optimal solution to find the path, although the executing time is less than other uninformed searches. Here is my result after implementing this searching technique:



```
VSCODE — -zsh — 138×24
(base) tamvo@Tams-MacBook-Air VSCODE % ./search input.txt ASTAR
Here is the map
_ _ X X _ _ _ _ X _ X
_ _ X X _ _ _ _ X _ _
_ _ _ _ _ _ _ _ _ _ _
_ _ X _ _ _ _ _ _ X _
_ _ X X X X _ _ X X _
The Source is (1 0)
The Destination is (3 10)
A Star Searching
(1 ,0) (2 ,0) (2 ,1) (2 ,2) (2 ,3) (3 ,3) (3 ,4) (3 ,5) (3 ,6) (2 ,6) (2 ,7) (2 ,8) (2 ,9) (2 ,10) (3 ,10)
The Agent need 81084ns (NanoSeconds) to findout the solution
(base) tamvo@Tams-MacBook-Air VSCODE %
```

Figure 13: Searching result using A*

4 Programming Overview Design

The UML diagram below demonstrates the relationship in the program. There are 11 classes. This section will not explain the function of these classes: BFS, DFS, GBFS, ASTAR, IDS, and DLS because obviously, they are in charge of searching the path. Meanwhile,

- Input class is responsible for reading data from **input.txt**
- Clock class is responsible for measuring the performance of a searching technique
- Map class is responsible for storing information about the environment

Here is the UML diagram:

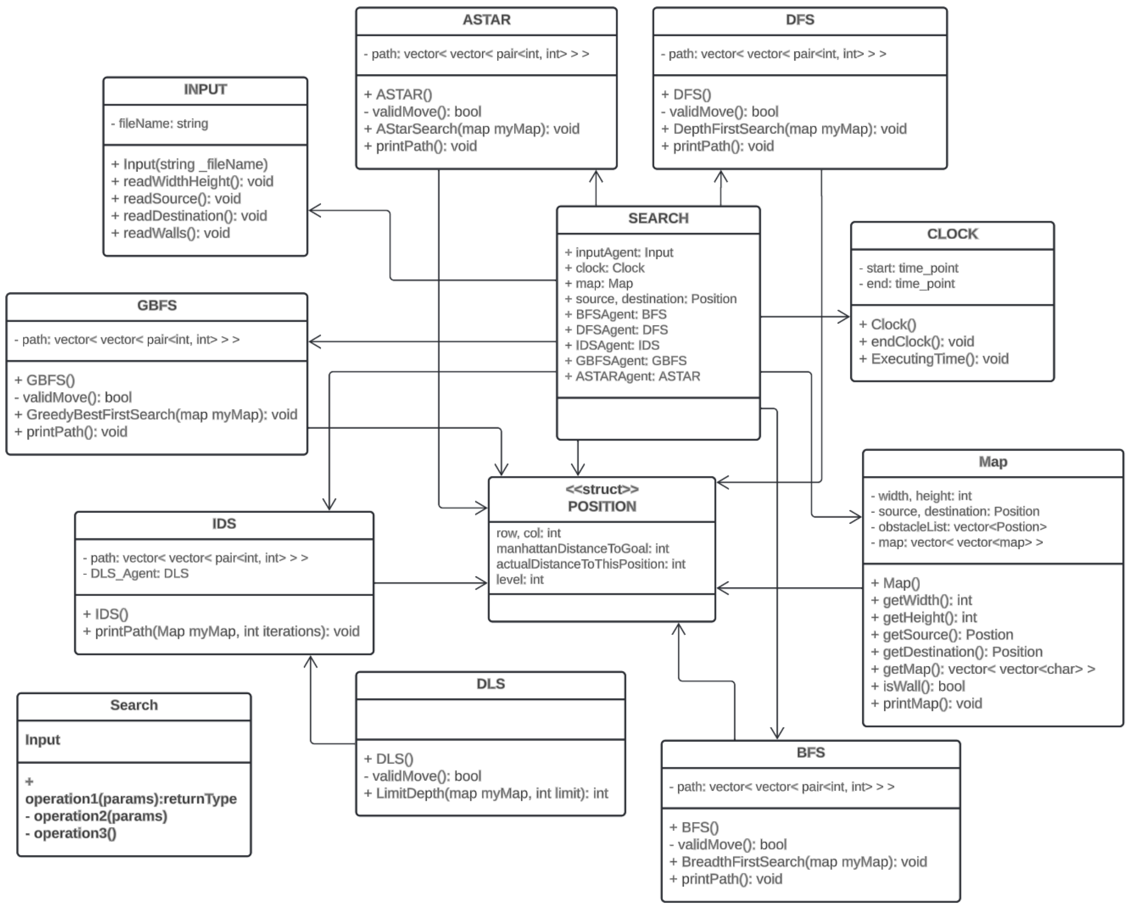

ASTAR
- path: vector< vector< pair<int, int> > >
+ ASTAR()
- validMove(): bool
+ AStarSearch(map myMap): void
+ printPath(): void
DFS
- path: vector< vector< pair<int, int> > >
+ DFS()
- validMove(): bool
+ DepthFirstSearch(map myMap): void
+ printPath(): void
INPUT
- fileName: string
+ Input(string _fileName)
+ readWidthHeight(): void
+ readSource(): void
+ readDestination(): void
+ readWalls(): void
SEARCH
+ inputAgent: Input
+ clock: Clock
+ map: Map
+ source, destination: Position
+ BFSAgent: BFS
+ DFSAgent: DFS
+ IDSAgent: IDS
+ GBFSAgent: GBFS
+ ASTARAgent: ASTAR
CLOCK
- start: time_point
- end: time_point
+ Clock()
+ endClock(): void
+ ExecutingTime(): void
GBFS
- path: vector< vector< pair<int, int> > >
+ GBFS()
- validMove(): bool
+ GreedyBestFirstSearch(map myMap): void
+ printPath(): void
<<struct>>
POSITION
row, col: int
manhattanDistanceToGoal: int
actualDistanceToThisPosition: int
level: int
Map
- width, height: int
- source, destination: Position
- obstacleList: vector<Postion>
- map: vector< vector<map> >
+ Map()
+ getWidth(): int
+ getHeight(): int
+ getSource(): Postion
+ getDestination(): Position
+ getMap(): vector< vector<char> >
+ isWall(): bool
+ printMap(): void
IDS
- path: vector< vector< pair<int, int> > >
- DLS_Agent: DLS
+ IDS()
+ printPath(Map myMap, int iterations): void
Search
Input
+ operation1(params):returnType
- operation2(params)
- operation3()
DLS
+ DLS()
- validMove(): bool
+ LimitDepth(map myMap, int limit): int
BFS
- path: vector< vector< pair<int, int> > >
+ BFS()
- validMove(): bool
+ BreadthFirstSearch(map myMap): void
+ printPath(): void

Figure 14: UML Diagram

5 Research field: Mechanism Behind Priority Queue

5.1 Introduction

In informed search implementation, the main difference is the frontier, instead of using a queue or stack to store the waiting cells, the priority queue is chosen to reduce the time complexity. The priority queue allows getting the maximum or minimum element with time complexity $O(1)$ and pushing/popping an element with the time complexity $O(logN)$. The idea behind the priority queue is based on the Heap data structure, which will be discovered in this session.

5.2 Heap Data Structure Overview

There are two types of Heap: Min Heap and Max Heap. The implementation of this program is prioritizing the cell with the lowest Manhattan Distance to the Goal, so Min Heap will be the main topic to be discussed. Below is one of the implementations of the priority queue in the program:

```cpp
#include <iostream>
#include <queue>
#include <vector>
using namepace std;
class GBFS{
    /// code

    void GreedyBestFirstSearch(Map myMap){
    /// declare the built-in priority queue
    priority_queue< Position, vector<Position>, optionManhattan > frontier;

    /// code
    }
};
```

Listing 3: Using priority queue in program

5.2.1 Properties

First, Min Heap is a complete binary tree, whose nodes at each level (except the level containing leaf nodes or the lowest level) are always fully filled. At the lowest level, leaf nodes should be filled from the left-hand side. Figure 16 below shows the demonstration of a complete binary tree:

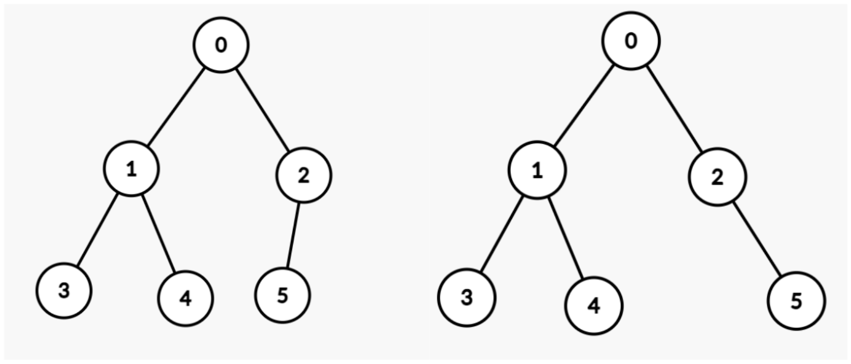


Figure 15: Complete Binary Tree and Incomplete Binary Tree

Second, in Mean Heap, the root is always the smallest element and every parent node is always less than its child nodes as Figure 17 below:

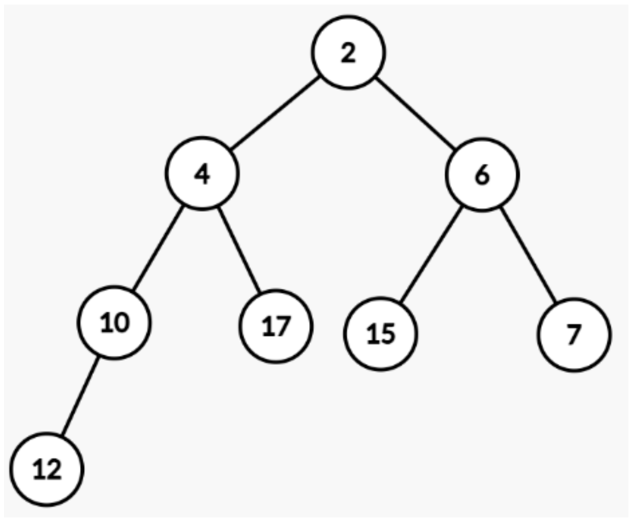


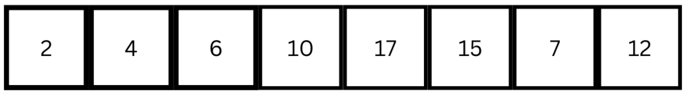
Figure 16: Example of a Min Heap

5.2.2 Using array to present Min Heap Data structure

We can use an array to present the Heap data structure: the first element at index 0 is always the root of the binary tree, and its child nodes are at index 1 and 2. Following that, the child nodes of index 1 are at 3 and 4, and so on. Hence we have:

Given an index a, its child nodes are at $(2a + 1)$ and $(2a + 2)$.

For example, Figure 17 below is a array representation for the Min Heap in Figure 16



2	4	6	10	17	15	7	12

Figure 17: Array representation for Figure 16

5.3 Min Heap Operation

There are 3 main operations that we can interact with a Min Heap including Heapify, Insertion, and Deletion.

5.3.1 Heapify

Heapify is an action that converts a normal array into a heap data structure. Beginning with the last node that has the leaf, then we check and swap its order with its child nodes' order in the array.

Here is the pseudocode for Heapify

```
Function Heapify(index, array)
    // Calculates the sum of two numbers
    current = index
    leftChild = 2 * index + 1
    rightChild = 2 * index + 2
    // Swap current position with its left child node
    if leftChild < size of array and array[leftChild] < array[current] then
        current = leftChild
    end if
    // Swap current position with its right child node
    if rightChild < size of array and array[rightChild] < array[current] then
        current = rightChild
    end if
    // Recursively checking
    if current ! = index then
        swap(array[current], array[index])
        Heapify(current, array)
    end if

Function HeapifyArray (array)
    for i = size of array / 2 - 1; i $\geq$ 0; i– do
        Heapify (i, array)
    end for
```

Figure 18: Pseudocode for Heapify an array

The algorithm above has a time complexity of $O(logN)$ and the space complexity is $O(N)$.

5.3.2 Insertion

Given a min heap array, adding a new element is pushing back that element at the end of the array, then we modify its position in the array to satisfy the requirement of a heap data structure. This operation is easy to understand because we do not need to use recursion to do it. Here is the pseudocode for inserting an element into a heap array:

```
Function insert(element, array)
    // Pushing back the element at the end of the array
    array ← pushing back element
    currentIndex ← size of array - 1

    While cuurrentIndex != 0 and array[(currentIndex - 1) / 2] > array[currentIndex] do
        swap(array[(currentIndex - 1) / 2], array[currentIndex])
        currentIndex = (currentIndex - 1) / 2
    end while do
```

Figure 19: Pseudocode for inserting an element to a heap

The algorithm above has a time complexity of $O(logN)$ and the space complexity is $O(N)$.

5.3.3 Deletion

Deletion of an element of a heap array is deleting the root of the binary tree or the first element of the array. then we must heapify the array again to adapt the rule of array representation for Heap. The idea is to replace the first element with the last element, next, we delete the last element and heapify the array from the first position.

Here is the Pseudocode for deleting an element of a heap data structure:

```
Function delete(array)
    // Pushing back the element at the end of the array
    Swap the first element and the last element
    Delete the last element of the array
    Heapify(0, array)
```

Figure 20: Pseudocode for deleting an element from a heap

The algorithm above has a time complexity of $O(logN)$ and the space complexity is $O(N)$.

5.4 Conclusion

Instead of sorting the queue to get the smallest element, by implementing the priority queue in informed searching, the time complexity is reduced significantly. Therefore, the performance of the Agent will be more impressive.

6 Reference

Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein, C. (2009). *Introduction to Algorithms* (3rd ed.). MIT Press.