



# **Introduction to Artificial Intelligence**

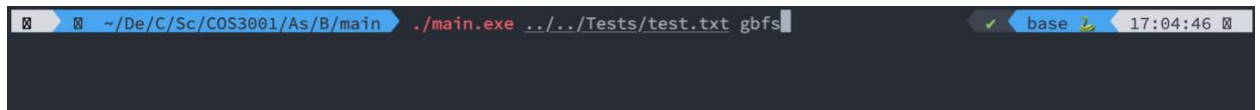
## **Assignment 1 Report Tree-based Search**

Xuan Tuan Minh Nguyen  
103819212

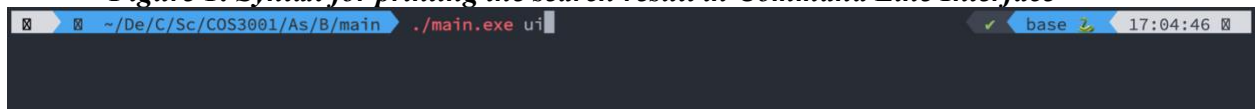
<b>INSTRUCTION.....</b>	<b>3</b>
<b>INTRODUCTION.....</b>	<b>4</b>
1. INTRODUCTION.....	4
2. GLOSSARIES.....	4
<b>SEARCH ALGORITHMS .....</b>	<b>5</b>
1. GENERAL CONCEPT.....	5
2. DEPTH-FIRST SEARCH (DFS) .....	5
Overview.....	5
Time Complexity.....	5
3. BREADTH-FIRST SEARCH (BFS) .....	6
Overview.....	6
Time Complexity.....	6
4. ITERATIVE DEEPENING DEPTH LIMITED SEARCH (IDDLs) .....	6
Overview.....	6
Time Complexity.....	6
5. GREEDY BEST-FIRST SEARCH (GBFS) .....	7
Overview.....	7
Time Complexity.....	7
6. A* SEARCH (A*).....	7
Overview.....	7
Time Complexity.....	8
7. WEIGHTED A* SEARCH (WA*) .....	8
Overview.....	8
Time Complexity.....	8
<b>IMPLEMENTATIONS.....</b>	<b>8</b>
1. UML CONCEPT.....	8
2. BREADTH-FIRST SEARCH (BFS) .....	10
3. DEPTH-FIRST SEARCH (DFS) .....	11
4. ITERATIVE DEEPENING DEPTH-LIMITED SEARCH (IDDLs).....	11
5. GREEDY BEST-FIRST SEARCH (GBFS) .....	12
6. A* SEARCH (A*).....	12
7. WEIGHTED A* SEARCH (A*) .....	13
<b>TESTING.....</b>	<b>13</b>
<b>FEATURES / BUGS.....</b>	<b>14</b>
1. FEATURES.....	14
2. BUGS.....	15
<b>RESEARCH.....</b>	<b>15</b>
<b>ACKNOWLEDGMENT / RESOURCES .....</b>	<b>16</b>
<b>REFERENCES.....</b>	<b>16</b>

## Instruction

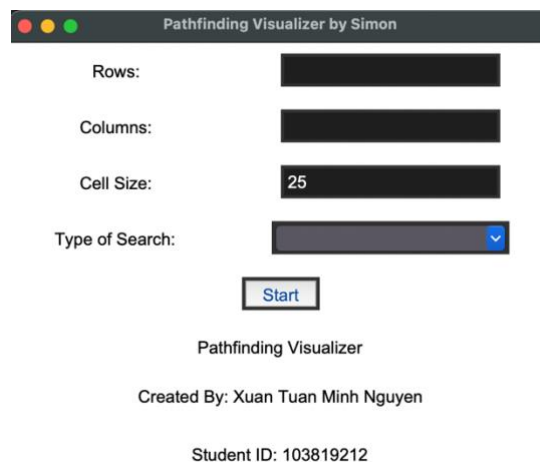
In order to output the moving steps from the start point to the goal point, simply navigate to the Command Line Interface, change the directory to “\Build\main”, and use the following syntax as provided in Figure 1: “main <filename> <method>” (or “python main.py <filename> <method>”), where <filename> is the text file contains the grid size (m x n), agent’s initial coordinates, goal coordinates separated by a “|” character, and the coordinates of the walls. And the <method> is the searching algorithm, it could be entered in both uppercase and lowercase and must follows the following list: dfs, bfs, gbfs, as, cus1, cus2. Besides, the program also have the ability to visualize the pathfinding algorithm using tkinter and the six implemented algorithm. Simply heads back to Command Line Interface (in the same directory of “\Build\main”, and type in the syntax that is provided in Figure 2: “main UI” (or “python main.py ui”). Figure 3 illustrates the first interface of the program, which let the user customizing the size of the grid (rows, columns and cell size) and choose which algorithm to executes. Figure 4 on the other hand is the main interface of the program, where user can chooses the starting and ending state of the agent and customizes the amount of walls (self-created or random-generated). During searching, the nodes in the frontier (not visited yet) are shown in the dark pink while the expanded nodes will be displayed in light pink. After the search is finished, the path will be illustrated with a green color.



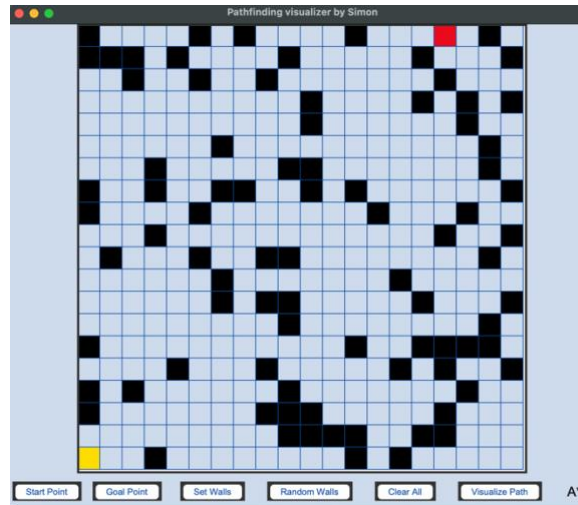
**Figure 1: Syntax for printing the search result in Command Line Interface**



**Figure 2: Syntax for running the GUI**



**Figure 3: The first user-interface of the program**



*Figure 4: The main user-interface of the program*

# Introduction

## 1. Introduction

In the field of Artificial Intelligence, using tree-based search algorithms for solving the Robot Navigation problem has been one of the most significant attentions in the context of pathfinding solutions. Throughout the years, multiple algorithms and solutions has been proposed to the public for solving the problem of Robot Navigation. The proposed solutions could be divided into two types of searching algorithms: Uninformed Search and Informed Search. This report is written to provide a comparison between the two given algorithmic approaches, which give an overview on the algorithms, the complexity, and the performance of each algorithm on a sample map. Furthermore, this report hands out the implementation of each algorithm on Python code, some encountered features and bugs and short research of visualizing the pathfinding algorithms using Tkinter library and the implemented algorithms.

## 2. Terminologies

- BFS: Breadth-First Search
- DFS: Depth-First Search
- GBFS: Greedy Best-First Search
- A\*: A-Star Search
- IDDLs: Iterative-Deepening Depth Limited Search
- Frontier: A data structure that helps keeping node that are going to be explored / expanded. Many different kinds of frontier have been introduced, however the tree frontier that is using in this report is queue, stack and priority queue.
- Visited Set: A set that helps keeping nodes that have been explored / expanded.

- $f(n)$ : Also known as heuristic function, is a function to calculate the distance between node(n) to the goal node.
- $h(n)$ : The cost from node(n) to goal.
- $g(n)$ : The cost that took to reach node(n).
- Manhattan Distance: A kind of heuristic function, which calculates the distance between two nodes ( $|ax - bx| + |ay - by|$ ).

## Search Algorithms

### 1. General Concept

This part aims to introduce six different popular pathfinding algorithms, which are Depth-First Search, Breadth-First Search, Iterative Deepening Depth Limited Search, Greedy Best-first Search, A\* Search, and Weighted A\* search. In addition, each given algorithm will be thoroughly examined in terms of space complexity (memory usage) and time complexity (efficiency).

### 2. Depth-First Search (DFS)

#### Overview

One of the most known uninformed search algorithms that has been the premise of multiple search algorithms, such as matching algorithm, Hopcroft-Kop, is the Depth-First search algorithm. Firstly, DFS will try to expand into the deepest node of the search tree until there is no further child nodes or any successors for a node. The algorithm then tries to trace back into the next deepest node where still has the unvisited nodes. Depth-First search algorithm could be implemented in two different ways, using a Last-In-First-Out data structure (stack) as the frontier of the search, or using recursive. Normally, the Depth-First search algorithm will not return the most optimal path as the rest of the proposed algorithms since it works by cramping into the deepest node as possible. But the space complexity when comparing with another uninformed search, such as Breadth-First search, is lower.

#### Time Complexity

Suppose  $m$  is the deepest depth of the node and  $b$  is the branching factor of each node.

- Time complexity:  $O(b^m)$
- Space complexity:  $O(bm)$

The space complexity of Depth-First Search when compared to Breadth-First Search is lower and the time complexity stays the same. DFS is a non-complete searching algorithm in spaces that have infinite depth spaces and spaces that contains loop over back to the previous node. To overcome this problem, the

functionalities of checking repeated state must be added to check if the successors is repeated with the visited nodes or not.

### 3. *Breadth-First Search (BFS)*

#### Overview

Aside from Depth-First Search algorithm, Breadth-First Search Algorithm is also a well-known search algorithm, which is also the premise algorithm for Uniform-Cost Search, Dijkstra's Search, and the rest of the proposed pathfinding algorithms. The algorithms will start from the root node and expand to all the child nodes. As the algorithm traverses to each depth of the node, it continuously expands all the nodes in the same depth before moving on to the next depth level. Breadth-First Search uses a First-In-First-Out data structure (Queue) as a frontier so all nodes could be expanded. In addition, by maintaining a set of visited nodes, the algorithm will make sure that the visited node will not be added back to the frontier, thus eliminate for any repeated states.

#### Time Complexity

Suppose  $d$  is the depth of the tree and  $b$  is the branching factor of each node.

- Time complexity:  $O(b^d)$
- Space complexity:  $O(b^d)$

With both space and time complexity of  $O(b^d)$ , the algorithm will result in a notable runtime and memory consumption, especially with a deep tree. However, according to Russel and Norvig (2010), due to the nature of the Breadth-First Search, the search will find the shallowest path to every node on the frontier, thus guarantees the optimality of the algorithm (if the cost to traverse between each node is equal to 1 then the search could be optimal).

### 4. *Iterative Deepening Depth Limited Search (IDDLs)*

#### Overview

Iterative Deepening Depth-Limited Search (IDDLs) is a search algorithm that combine the power of optimality and completeness from Breadth-First Search (BFS) and the space-efficiency from the Depth-First Search (DFS). The algorithm will start from the root node of the tree and then expand to all of its child nodes, it then performs depth-limited search repeatedly (using Last-In-First-Out data structure as the frontier) and at the same time increasing the depth limit of each iteration until the result is found. This strategy helps the algorithm explores all the nodes in the current depth before going down to the next depth.

#### Time Complexity

Suppose  $d$  is the depth of the tree and  $b$  is the branching factor of each node.

- Time complexity:  $O(b^d)$

- Space complexity:  $O(d)$

With the time complexity of  $O(b^d)$ , the algorithm will result in a significant runtime. However, the space complexity when compares to the two uninformed algorithms, is increasingly lower. Since Iterative Deepening Depth-Limited Search also owns the optimality and completeness from the Breadth-First Search, the search will always find the shallowest path to every node on the frontier thus is optimal when the path cost is equal.

## 5. Greedy Best-First Search (GBFS)

### Overview

Greedy Best-First Search (GBFS) is the simplest pathfinding algorithm in the Informed search group. It was inspired by Breadth-First Search, but with a heuristic function to filter out unnecessary nodes and focusing on finding the closest node to the goal. With a good heuristic function could result in an efficient and less complexity result, so the heuristic function that is chosen for the current problem is Manhattan Distance. The algorithm then appends the node into a priority queue based on the calculated heuristic value and the search will try to proceed to the most promise value node.

### Time Complexity

Suppose  $s$  is the maximum tree depth and  $b$  is the branching factor of the node.

- Time complexity:  $O(b^s)$
- Space complexity:  $O(b^s)$

It is clear to see that the worst-case time complexity of Greedy Best-First Search is equal to Depth-First Search and since all the nodes are kept in the queue and the visited set, the memory consumption is very large. Thus, this algorithm is not the most optimal algorithm since it does not track the cost of the past and it does not assure the optimal path as well.

## 6. A\* Search (A\*)

### Overview

A-Star search (A\*) is in fact, the most popular informed algorithms that is used widely in the world. Instead of evaluating, like Greedy Best-First Search heuristic function  $f(n) = h(n)$  where  $h(n)$  is the cost from node  $n$  to the goal, A\* on the other hand adds in the  $g(n)$  function, which is the cost to reach the node  $n$ . Therefore, the heuristic function of A\* is  $f(n) = h(n) + g(n)$ . Since the heuristic function is improved from Greedy Best-First Search, the algorithm is now complete and optimal. However, A\* has a big drawback, which the heuristic function must be admissible. The heuristic function must make sure that it never

overestimates the cost to the goal, which  $h(n) \leq h^*(n)$  where  $h^*(n)$  is the most optimal cost to reach the goal.

### Time Complexity

Suppose  $d$  is the depth of best solution and  $b$  is the branching factor of the node.

- Time complexity:  $O(b^d)$
- Space complexity:  $O(b^d)$

Since A\* must expand even more nodes than Greedy Best-First Search and all the nodes must be kept, the time and space complexity of A\* is generally bigger than Greedy Best-First Search but it will assure a most optimal and shortest path to the goal.

## 7. *Weighted A\* Search (WA\*)*

### Overview

In general, Weighted A\* Search (WA\*) is the A\* Search algorithm. However, instead of having the heuristic function of  $f(n) = h(n) + g(n)$ , weighted A\* will add a weight ( $\epsilon$ ) into the heuristic function, making it as  $f(n) = h(n) + \epsilon \times g(n)$ . Which if the weight ( $\epsilon$ ) value is greater than 1 which could speed up the search but will result in a suboptimal path (if weight ( $\epsilon$ ) is equal to 1 then this function is the normal A\* search).

### Time Complexity

Suppose  $d$  is the depth of best solution and  $b$  is the branching factor of the node.

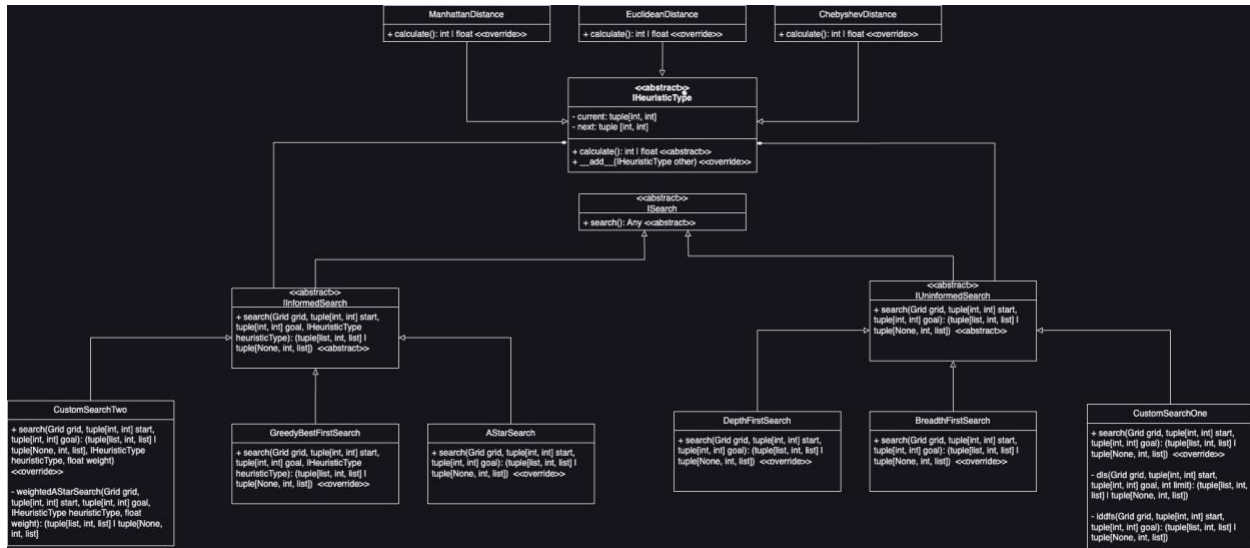
- Time complexity:  $O(b^d)$
- Space complexity:  $O(b^d)$

In general, the time and space complexity of the Weighted A\* is equal to the A\*. However, since the space complexity of A\* heavily dictated by the number of nodes it kept, thus increasing the weight ( $\epsilon$ ) value could result in a lower worst-case time complexity.

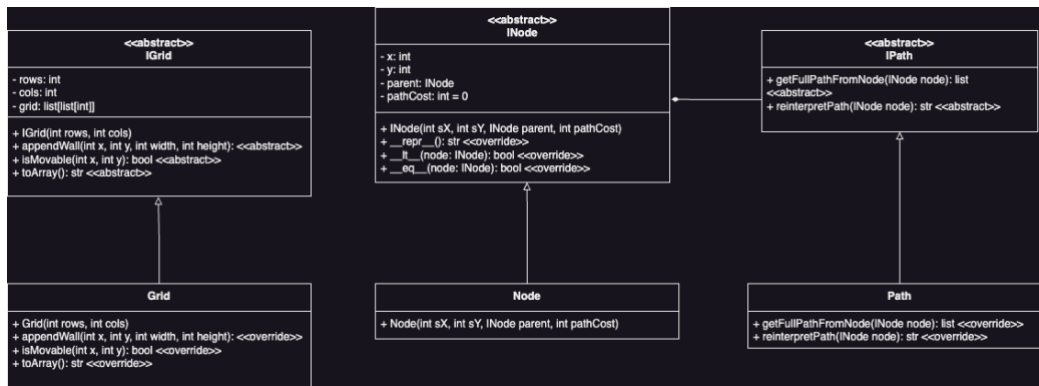
## Implementations

### 1. *UML Concept*





**Figure 5: Search Algorithm and Heuristic Function UML Diagram of the Program**



**Figure 6: Grid, Node and Path UML Diagram of the Program**

## 2. Breadth-First Search (BFS)

The primary data structure that Breadth-First Search uses as the primary frontier is a First-In-First-Out queue, which is deque in this case, to ensure a level-by-level exploration. Firstly, the root node must be kept into the queue. After that, the program will use a while loop to loop the procedure until the queue is empty or if the value is returned. While in the loop, the first element of the queue will be pop out and check if it matches the goal. If it matches the goal, then returns the value and terminates the program. Otherwise, it will continuously loop through all neighbors and cells to check if it is a 'valid' cell (not a wall) then push those 'valid' cells into the queue. The program must terminate if the goal is found or if there are no paths found, which mean no cell left in the queue.

A screenshot of a code editor with a dark background and light-colored text. The code is a Python class named 'BreadthFirstSearch' that inherits from 'IUninformedSearch'. It defines a 'search' method that takes a grid, start coordinates, and goal coordinates as input. The method initializes a 'frontierQueue' as a deque containing the start node, a 'countedNodes' counter, a 'visitedNodes' set, and a 'changes' list. It then enters a while loop that continues as long as the 'frontierQueue' is not empty. Inside the loop, it pops the first element from the queue, checks if it is the goal, and if so, returns the full path, counted nodes, and changes. If not the goal, it adds the node to the visited set and the changes list. It then iterates over the four possible directions (up, down, left, right) and for each direction, it calculates the next coordinates. If the next coordinates are within the grid and the cell is movable (not a wall) and has not been visited, it creates a new node with the updated coordinates and path cost, and adds it to the frontier queue. Finally, it returns None, counted nodes, and changes if no path is found.

```
1 class BreadthFirstSearch(IUninformedSearch):
2     def search(self, grid: Grid, start: tuple[int, int], goal: tuple[int, int]):
3         frontierQueue = deque([Node(start[0], start[1])])
4         countedNodes = 0
5         visitedNodes = set()
6         changes = []
7         while frontierQueue:
8             node = frontierQueue.popleft()[0]
9             countedNodes += 1
10            if (node.x, node.y) == goal:
11                return Path.getFullPathFromNode(node), countedNodes, changes
12            visitedNodes.add((node.x, node.y))
13            changes.append(((node.x, node.y), 'visited'))
14            for directionX, directionY in Direction.getTupleValues():
15                nextX, nextY = directionX + node.x, directionY + node.y
16                if (nextX, nextY) not in visitedNodes and grid.isMovable(nextX, nextY):
17                    childNode = Node(nextX, nextY, node, node.pathCost + 1)
18                    frontierQueue.append(
19                        Path.getFullPathFromNode(node) + [childNode])
20                    visitedNodes.add((nextX, nextY))
21                    changes.append(((nextX, nextY), 'frontier'))
22            return None, countedNodes, changes
```

*Figure 7: Breadth-First Search Implementation in Python*

### 3. Depth-First Search (DFS)

Theoretically, the Depth-First Search is identical to the Breadth-First Search. However, the main different is instead of using First-In-First-Out queue, Depth-First Search uses Last-In-First-Out stack to ensure a depth-by-depth exploration.



```

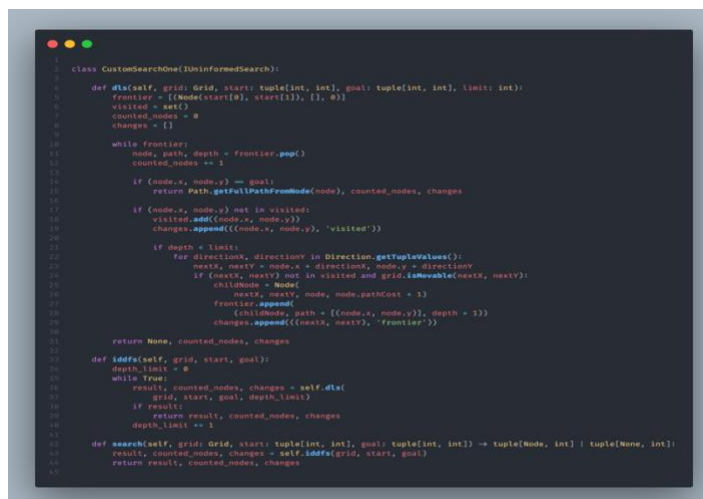
1 class DepthFirstSearch(IUninformedSearch):
2
3     def search(self, grid: Grid, start: tuple[int, int], goal: tuple[int, int]):
4         frontierStack = [(Node(start[0], start[1]), [])]
5         countedNodes = 0
6         visitedNodes = set()
7         changes = []
8         while frontierStack:
9             node, _ = frontierStack.pop()
10            countedNodes += 1
11            if (node.x, node.y) == goal:
12                return Path.getFullPathFromNode(node), countedNodes, changes
13            visitedNodes.add((node.x, node.y))
14            changes.append(((node.x, node.y), 'visited'))
15
16            unvisitedNodes = []
17            for directionX, directionY in Direction.getTupleValues():
18                nextX, nextY = directionX + node.x, directionY + node.y
19                if (nextX, nextY) not in visitedNodes and grid.isMovable(nextX, nextY):
20                    childNode = Node(nextX, nextY, node, node.pathCost + 1)
21                    unvisitedNodes.append(
22                        (childNode, Path.getFullPathFromNode(node)))
23            changes.append(((nextX, nextY), 'frontier'))
24            frontierStack.extend(unvisitedNodes[::-1])
25        return None, countedNodes, changes

```

Figure 8: Depth-First Search Implementation in Python

### 4. Iterative Deepening Depth-Limited Search (IDDLs)

Based on the Depth-First Search algorithm, Iterative Deepening Depth-Limited Search is pretty much identical to the Depth-First Search. However, it will have a side function to set the depth limit to 0 and then increasing it once by once until it reaches the goal node.



```

1 class CustomSearchNone(IUninformedSearch):
2
3     def dls(self, grid: Grid, start: tuple[int, int], goal: tuple[int, int], limit: int):
4         frontier = [(Node(start[0], start[1]), [], 0)]
5         visited = set()
6         counted_nodes = 0
7         changes = []
8
9         while frontier:
10            node, path, depth = frontier.pop()
11            counted_nodes += 1
12
13            if (node.x, node.y) == goal:
14                return Path.getFullPathFromNode(node), counted_nodes, changes
15
16            if (node.x, node.y) not in visited:
17                visited.add((node.x, node.y))
18                changes.append(((node.x, node.y), 'visited'))
19
20            if depth < limit:
21                for directionX, directionY in Direction.getTupleValues():
22                    nextX, nextY = node.x + directionX, node.y + directionY
23                    if (nextX, nextY) not in visited and grid.isMovable(nextX, nextY):
24                        childNode = Node(
25                            nextX, nextY, node, node.pathCost + 1)
26                        frontier.append(
27                            (childNode, path + [(node.x, node.y)], depth + 1))
28                changes.append(((nextX, nextY), 'frontier'))
29
30        return None, counted_nodes, changes
31
32     def iddfs(self, grid: Grid, start: tuple[int, int], goal: tuple[int, int]):
33         depth_limit = 0
34         while True:
35             result, counted_nodes, changes = self.dls(
36                 grid, start, goal, depth_limit)
37             if result:
38                 return result, counted_nodes, changes
39             depth_limit += 1
40
41     def search(self, grid: Grid, start: tuple[int, int], goal: tuple[int, int]) -> tuple[Node, int] | tuple[None, int]:
42         result, counted_nodes, changes = self.iddfs(grid, start, goal)
43         return result, counted_nodes, changes

```

Figure 9: Iterative Deepening Depth-Limited Search Implementation in Python

## 5. Greedy Best-First Search (GBFS)

The context of Greedy Best-First Search is similar to Breadth-First Search, but it uses a priority queue instead of a normal queue to store the nodes. Priority queue ensures that nodes must be stored based on the cost and the smallest cost node will be pops out first. In the current solution, Manhattan distance has been chosen as the heuristic function to calculate the cost from node to node. In case of multiple goal, it will return the one that has the shortest distance.

```

class GreedyBestFirstSearch(InformedSearch):
    def search(self, grid: Grid, start: tuple[int, int], goal: tuple[int, int], heuristicType: DHeuristicType | int) -> (tuple[list, int, list] | tuple[None, int, list]):
        if isinstance(heuristicType, DHeuristicType):
            nodeCost = heuristicType(start, goal).calculate()
        else:
            nodeCost = Heuristic(
                start, goal).getHeuristicFunction(heuristicType)
        frontierQueue = [(nodeCost, Node(start[0], start[1]))]
        heapq.heappush(frontierQueue)
        visitedNodes = set()
        changes = []
        countedNodes = 0
        while frontierQueue:
            cost, node = heapq.heappop(frontierQueue)
            countedNodes += 1
            if (node.x, node.y) == goal:
                return Path.getFullPathFromNode(node), countedNodes, changes
            visitedNodes.add((node.x, node.y))
            changes.append((node.x, node.y), 'visited')
            for direction, directionInDirection in Direction.getTuplesValues():
                nextx, nexty = direction + node.x, direction + node.y
                if (nextx, nexty) not in visitedNodes and grid.isMovable(nextx, nexty):
                    childNode = Node(nextx, nexty, node, node.getCost() + 1)
                    if isinstance(heuristicType, DHeuristicType):
                        nodeCost = heuristicType(start, goal).calculate()
                    else:
                        nodeCost = Heuristic(
                            start, goal).getHeuristicFunction(heuristicType)
                    heapq.heappush(frontierQueue, (nodeCost, childNode))
                    visitedNodes.add((nextx, nexty))
                    changes.append((nextx, nexty), 'frontier')
        return None, countedNodes, changes

```

Figure 10: Greedy Best-First Search Implementation in Python

## 6. A\* Search (A\*)

Theoretically, A\* has the same implementation to Greedy Best-First Search in terms of the context. However, the major difference between A\* and Greedy Best-First Search is the way it evaluates the costs of a node. Instead of calculating the cost  $h(n)$  from node  $n$  to goal, A\* will also add in the cost  $g(n)$  to reach the node  $n$ .

```

class AStarSearch(InformedSearch):
    def search(self, grid: Grid, start: tuple[int, int], goal: tuple[int, int], heuristicType: DHeuristicType | int) -> (tuple[list, int, list] | tuple[None, int, list]):
        if isinstance(heuristicType, DHeuristicType):
            nodeCost = heuristicType(start, goal).calculate()
        else:
            nodeCost = Heuristic(
                start, goal).getHeuristicFunction(heuristicType)
        frontierQueue = [(nodeCost, Node(start[0], start[1]))]
        heapq.heappush(frontierQueue)
        visitedNodes = set()
        changes = []
        countedNodes = 0
        while frontierQueue:
            cost, node = heapq.heappop(frontierQueue)
            countedNodes += 1
            if (node.x, node.y) == goal:
                return Path.getFullPathFromNode(node), countedNodes, changes
            visitedNodes.add((node.x, node.y))
            changes.append((node.x, node.y), 'visited')
            for direction, directionInDirection in Direction.getTuplesValues():
                nextx, nexty = direction + node.x, direction + node.y
                if (nextx, nexty) not in visitedNodes and grid.isMovable(nextx, nexty):
                    childNode = Node(nextx, nexty, node, node.getCost() + 1)
                    if isinstance(heuristicType, DHeuristicType):
                        nodeCost = heuristicType(start, goal).calculate()
                    else:
                        nodeCost = Heuristic(
                            start, (nextx, nexty)).getHeuristicFunction(
                                heuristicType) + Heuristic((nextx, nexty), goal).getHeuristicFunction(heuristicType)
                    heapq.heappush(frontierQueue, (nodeCost, childNode))
                    visitedNodes.add((nextx, nexty))
                    changes.append((nextx, nexty), 'frontier')
        return None, countedNodes, changes

```

Figure 11: A\* Search Implementation in Python

## 7. Weighted A\* Search (A\*)

Weighted A\* Search is A\* search but has a weight ( $\epsilon$ ) value to let the user determine how much weight they wanted to add into the heuristic function in order to change the time complexity of algorithm.

```

1 class CustomSearchTwo(InformedSearch):
2
3     def search(self, grid: Grid, start: tuple[int, int], goal: tuple[int, int], heuristicType: IHeuristicType | int, weight: int = 0):
4
5         if weight == 0:
6             _weight = 1.5
7         else:
8             _weight = weight
9
10        result, nodeCount, changes = self._weightedAStarSearch(
11            grid, start, goal, heuristicType, _weight)
12
13        return result, nodeCount, changes
14
15    def _weightedAStarSearch(self, grid: Grid, start: tuple[int, int], goal: tuple[int, int], heuristicType: IHeuristicType | int, weight: float):
16        start_node = Node(start[0], start[1], None, 0)
17
18        if isinstance(heuristicType, IHeuristicType):
19            start_heuristic = heuristicType(start, goal).calculate()
20        else:
21            start_heuristic = Heuristic(
22                start, goal).getHeuristicFunction(heuristicType)
23
24        start_cost = start_node.pathCost + weight * start_heuristic
25        frontierQueue = [(start_cost, start_node)]
26        heapq.heappush(frontierQueue)
27
28        visitedNodes = set()
29        countedNodes = 0
30        changes = []
31
32        while frontierQueue:
33            current_cost, node = heapq.heappop(frontierQueue)
34            countedNodes += 1
35
36            if (node.x, node.y) == goal:
37                return Path.getFullPathFromNode(node), countedNodes, changes
38
39            visitedNodes.add((node.x, node.y))
40            changes.append(((node.x, node.y), 'visited'))
41
42            for direction, direction_v in Direction.getTupleValues():
43                nextx, nexty = node.x + direction.x, node.y + direction.y
44                if (nextx, nexty) not in visitedNodes and grid.isMovable(nextx, nexty):
45                    childNode = Node(nextx, nexty, node, node.pathCost + 1)
46                    if isinstance(heuristicType, IHeuristicType):
47                        child_heuristic = heuristicType(
48                            (nextx, nexty), goal).calculate()
49                    else:
50                        child_heuristic = Heuristic(
51                            (nextx, nexty), goal).getHeuristicFunction(heuristicType)
52
53                    # Apply the weighting factor only to the heuristic component
54                    child_cost = childNode.pathCost + weight * child_heuristic
55                    heapq.heappush(frontierQueue, (child_cost, childNode))
56                    visitedNodes.add((nextx, nexty))
57                    changes.append(((nextx, nexty), 'frontier'))
58
59        return None, countedNodes, changes

```

Figure 12: Weighted A\* Search Implementation in Python

## Testing

Given the test environment as in the Figure 13, the tested result of each algorithm, which is the average after executing the algorithm six times, can be illustrated in the Table 1 below.

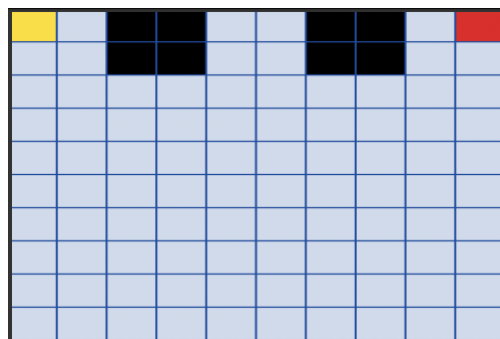


Figure 13: Test environment

Algorithm	BFS	DFS	IDDLS	GBFS	A*	Weighted A* ( $\epsilon = 1.25$ )
Time(ms)	0.68	1.64	1.51	0.543	0.508	0.342
Nodes	77	83	20	19	19	19

*Table 1: Node expanded and time complexity of each algorithm.*

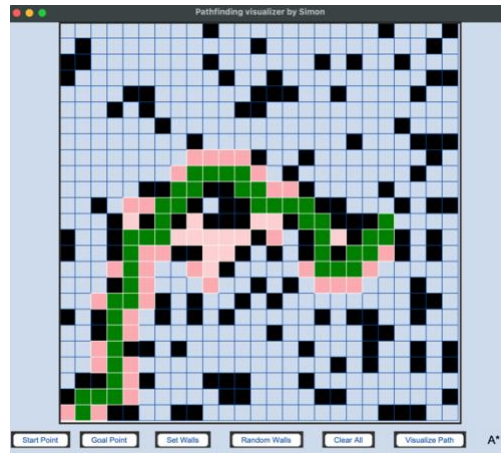
For the uninformed search group, it could be clearly seen that while the Iterative Deepening Depth-Limited Search (IDDLS) expanded the least nodes and Depth-First Search (DFS) expanded the most nodes, it is also notable to capture that Breadth-First Search takes the smallest amount of time to execute. On the other hand, all of the three informed search algorithms have the same number of nodes taken. The only different statistic is the amount of taken time, which could be clearly seen that Weighted A\* (with  $\epsilon = 1.25$ ) consumed the least amount of time compared to the two other algorithms while still delivers an optimal and completed path.

## Features / Bugs

### 1. Features

Features that have been implemented

- Command Line Arguments that can read the text file containing grid size (n x m), start and goal coordinates, multiple walls coordinates and the searching algorithms, parsing it and passing it to the algorithm classes.
- Six proposed algorithms have been implemented, which are Breadth-First Search, Depth-Limited Search, Greedy Best-First Search, A\*, Iterative Deepening Depth-Limited Search and Weighted A\*. The Iterative Deepening Depth-Limited Search and Weighted A\* are the two custom search algorithms, while the other four algorithms are required.
- A Graphical User Interface (GUI) has been created for user to visualize the algorithm in a NxM grid environment. As seen in the Figure 4, the yellow and red cells are the start and goal point of the agent, and the blacked cells are the wall that user could manually generates or let the computer randomly generates. In the Figure 14, the darker pink cells are the nodes that is in the frontier queue (stack) while the lighter pink cells are the nodes that has been visited, and the final path are the green cells joining together.



*Figure 14: The main User-Interface after finished searching*

## 2. Bugs

Throughout the process of implementation, I have encountered lots of errors regarding the logic of the code. For example, I have used the wrong data structure for Breadth First Search, which is the normal array instead of a queue, thus outputted the wrong results. In addition, the GUI also caused several non-sense bugs, which slower my process of implementing the GUI. However, after taking a revision on the code and having a look at the documentations, I have managed to overcome the bugs and have successfully implemented the program in my intended plan.

## Research

In this research section, I have chosen the topic of “In addition to the mandatory command-line-based user interface, can you build a GUI to display the environment and how the algorithm is trying to find the solution? This option should also include a visualizer to show the changes happening to the search tree”.

In advance to implement this research topic, the Graphical User Interface library that is my main library is tkinter, which is one of the well-known Python GUI Library. The visualizer program simulates the same aspect just like the Command Line Interface program, such as a grid, the ability to customize the start and goal coordinates, and customize the wall. However, one advance of the Graphical User Interface program is that it can visualize the process of searching and illustrating the search process, which the nodes in the darker pink color are the node in the frontier, the nodes in the lighter pink are the visited nodes and the green nodes that are joined as a path will be the path generated by the algorithm. For more reference, refer to Figure 3, 4 and 14.



## Conclusion

In conclusion, this report has delivered a specific analyze on six different popular pathfinding algorithms. All of the proposed algorithms are being used, with the Robot Navigation problem, to distinguish which algorithms is better. Moreover, the report also covers the implementation process of the algorithms and the program, which includes GUI and Command Line Interface. In the group of uninformed searches, it could be clearly seen that the Iterative Deepening Depth-Limited Search algorithm has a better performance in term of time and space complexity. However, if the user prioritizes the solution that provides the shortest path as possible but not caring about the memory usage, informed search, especially Weighted A\* Search, is the best choice for user to go with. With Weighted A\* Search, user can dynamically adjust the weight value based on the demand of time complexity or the completion of the path.

## Acknowledgment / Resources

While the first six articles, which are published by Brilliant.org, Wikipedia, AI Master and Ebendt, R. and Drechsler, R. are the articles that helps me understanding the concept and the mechanism of the six proposed pathfinding algorithms. The last reference, which is a Python document, is the one that helped me got a closer look at the functions or classes that are provided by tkinter and the usage of each classes/function, thus has helped me creating the Graphical User Interface program.

## References

- [1] Depth-First Search (DFS). *Brilliant.org*. Retrieved 23:27, April 9, 2024, from <https://brilliant.org/wiki/depth-first-search-dfs/>
- [2] Breadth-First Search (BFS). *Brilliant.org*. Retrieved 23:27, April 9, 2024, from <https://brilliant.org/wiki/breadth-first-search-bfs/>
- [3] Wikimedia Foundation. (2023, November 24). *Iterative deepening depth-first search*. Wikipedia. [https://en.wikipedia.org/wiki/Iterative\\_deepening\\_depth-first\\_search](https://en.wikipedia.org/wiki/Iterative_deepening_depth-first_search)
- [4] Master, AI. (2024). *What is greedy best-first search?*. What is Greedy Best-first Search? · Heuristic Search. <https://ai-master.gitbooks.io/heuristic-search/content/what-is-greedy-best-first-search.html>
- [5] Master, AI. (2024). *What is A-search?*. What is A-search? · Heuristic Search. <https://ai-master.gitbooks.io/heuristic-search/content/what-is-a-search.html>



[6] Ebendt, R., & Drechsler, R. (2009, June 16). *Weighted a\* search – unifying view and application*. Artificial Intelligence.

<https://www.sciencedirect.com/science/article/pii/S000437020900068X>

[7] Python, D. (n.d.). *Tkinter - Python interface to TCL/TK*. Python documentation. <https://docs.python.org/3/library/tkinter.html>