



COS30002 - AI for Games

Robot Navigation Search

Nguyen Khanh Toan

104180605

Tutor class: 12:30 Friday

104180605@student.swin.edu.au

Words count: 2717

II. Table of Content

Robot Navigation Search.....	0
I. Instruction.....	1
II. Table of Content.....	2
III. Introduction.....	3
IV. Program Design.....	3
Development Tools:.....	3
Programming Language.....	3
GUI Framework.....	3
Libraries.....	3
Glossary:.....	4
UML Diagram:.....	5
Algorithm Implementation:.....	6
Breadth-first search algorithms.....	6
Depth-first search algorithms.....	7
Greedy Best-first search algorithms.....	8
A star (A*) search algorithms.....	9
Bi-Directional search algorithms.....	10
Bi-Directional with A* search algorithms.....	11
V. Intended Learning Outcomes (ILO).....	13
Understand Key Search Algorithms:.....	13
Develop and Implement Search Algorithms:.....	13
Understand Search Mechanisms Beyond Unit Content:.....	13
Analyze and Evaluate Algorithm Performance:.....	13
Design and Implement a Graphical User Interface (GUI):.....	13
VI. Summary.....	14
VII. Acknowledgement.....	14
VIII. References.....	15

III. Introduction

The robot navigation project focuses on creating algorithms and structures for a robot to navigate a grid-based environment. It involves implementing search algorithms like BFS, DFS, GBFS, and A* to find the best path from the robot's current position to its goal while avoiding obstacles. Data structures such as grids and nodes represent the environment and locations within it, while priority queues manage exploration efficiently. Additionally, heuristics like Manhattan distance guide the robot towards the goal. Input/output operations handle reading grid configurations from files and displaying path and exploration statistics. Overall, the project aims to develop a robust navigation system enabling the robot to autonomously plan and execute its path, avoiding obstacles and minimizing exploration time.

IV. Program Design

Development Tools

Programming Language

- **Python 3.12:** The project is developed using Python 3.12, which offers modern features and optimizations suitable for both algorithmic development and GUI design.

GUI Framework

- **PyGame:** PyGame is used for developing the graphical user interface of the project. It provides functionalities for handling graphics, user input, and events, making it ideal for visualizing the grid and the robot's navigation process.

Libraries

- **PyGame:** Utilized for GUI development, event handling, and visualization of the grid and robot movements.
- **math:** Provides mathematical functions and constants used for calculations within the project.
- **utils:** A custom utility module that includes functions for number extraction, format validation, and other general-purpose operations.
- **sys:** Provides access to system-specific parameters and functions, used for handling file operations and system exit.

- **numpy**: Used for handling numerical data, creating and manipulating arrays and matrices, which are integral for grid representation and manipulation.
- **time**: Utilized for time-related functions, such as delays in displaying the grid initialization or measuring execution time.
- **collections.deque**: Provides a double-ended queue that supports appending and popping from both ends, used in various queue operations.
- **functools**: Includes higher-order functions for functional programming, such as memoization.
- **ast**: Used for abstract syntax tree operations, specifically for safely evaluating expressions.
- **heapq**: Implements a heap queue algorithm, used for priority queue operations.
- **re**: Provides regular expression matching operations, useful for input validation.
- **statistics.mean**: Used for calculating the mean value of a set of numbers, which can be useful in heuristic evaluations and statistical analyses.

Glossary

- **BFS**: Breadth-first search
- **DFS**: Depth-first search
- **GBFS**: Greedy Best-first search
- **A***: A star search
- **CUS1**: Bi-directional BFS (Custom Search)
- **CUS2**: Bi-directional A Star (Custom Search)
- **Heuristic function**: A function calculating the distance from a node to the goal.
- **Manhattan distance**: Calculating the distance between two points based on the given formula:
 - $abs(a.x - b.x) + abs(a.y - b.y)$
- **PriorityQueue**: A queue type that stores elements based on their priority, arranging them in ascending or descending order.
- **Frontier**: A data structure that maintains a set of nodes ready for expansion and exploration. This report focuses on three queue types: Stack, Queue, and PriorityQueue.
- **Node**: A class representing a node, including its state, parent, path, and path cost.
- **Grid**: A class that defines a grid with attributes like size, initial state, goal state, and a set of obstacles.
- **GridProblem**: A class that represents a problem specific to a Grid object.

- Utils: A collection of utility functions available for use by other classes or functions.
- Visual: A class for visualizing the program's display, with all visual elements modifiable within this class.
- Button: A class representing an individual button, including its X and Y coordinates, text, and functionality.

UML Diagram

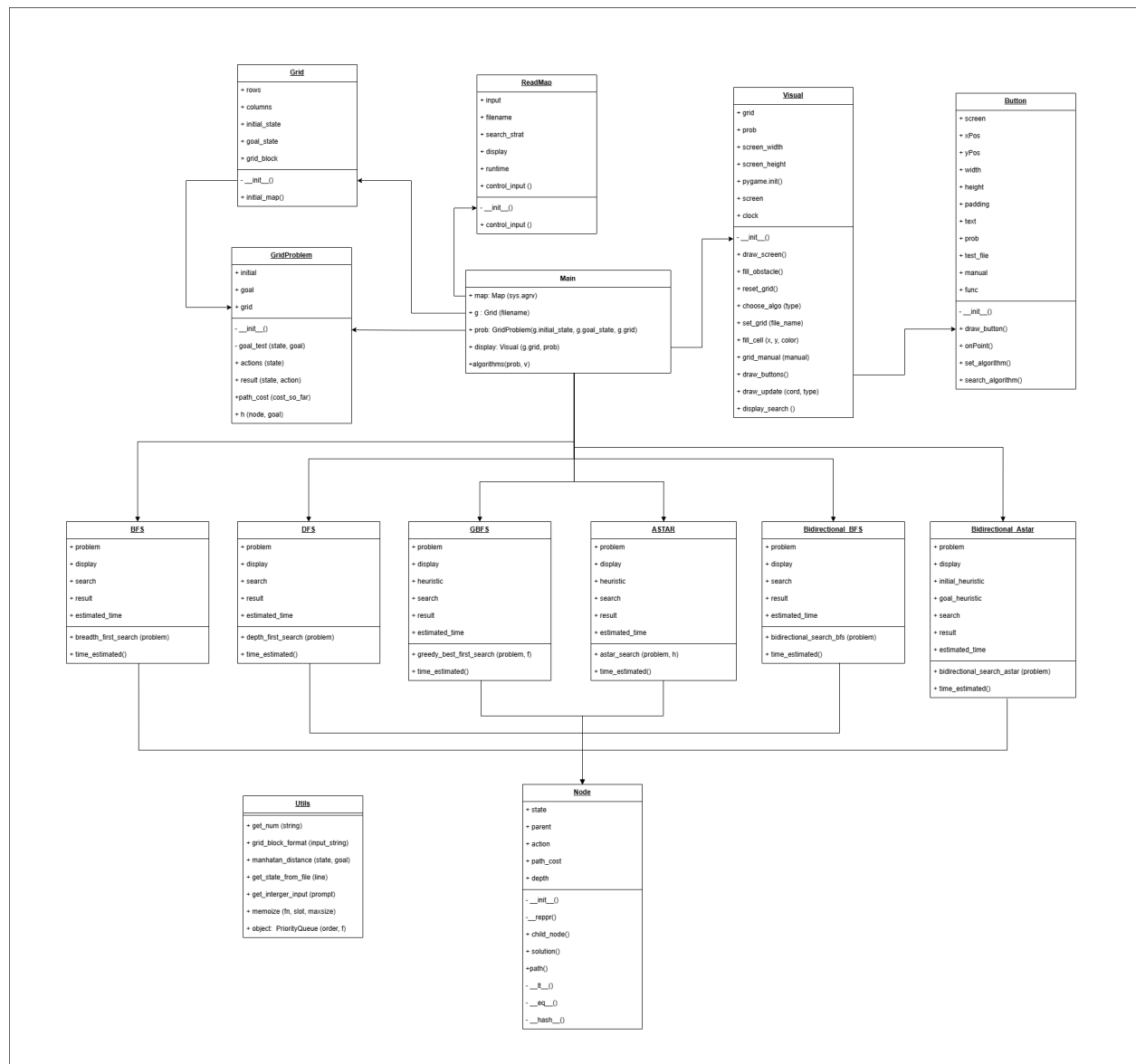


Figure 1: UML class Diagram

Algorithm Implementation

Breadth-first search algorithms


Breadth-first search (BFS) is an algorithm that explores all neighbor nodes at the current depth level before proceeding to nodes at the next depth level. It starts at the root node and systematically explores each neighbor of the current node, adding them to the frontier queue. It continually removes the node at the head of the queue, checks if it is the goal, and, if not, adds its neighbors to the back of the queue, repeating this process level by level until it finds the goal node or all nodes being explored.

Node expansion in BFS is achieved using a First-In-First-Out (FIFO) queue, which ensures that nodes are explored in the order they were added, maintaining the level order of exploration.

If the goal node is at a certain finite depth d , BFS will explore all shallower nodes before successfully finding the goal node. This characteristic means BFS can find a path to the goal with the fewest actions. However, it does not guarantee that the found path is optimal in terms of cost, as BFS does not consider action costs.

The time complexity of BFS is determined by the size of the state space. Given a branching factor b , BFS explores all nodes up to depth d , resulting in a time complexity of $O(b^d)$. This complexity arises because BFS explores all nodes at level d , which amounts to $O(b^d) = b + b^2 + b^3 + \dots + b^d$ nodes.

Regarding space complexity, BFS requires storing all nodes at depth d to generate their child nodes at depth $d + 1$. Consequently, the space complexity is also $O(b^d)$, proportional to the total number of nodes. This high memory requirement is a significant drawback of BFS, could lead to buffer overflow issues when dealing with problems that have large depths.



```

1 def breadth_first_search():
2     Initialize an empty queue frontier
3     Add the initial state of the problem to the frontier
4     Initialize an empty set explored to keep track of explored states
5
6     for each initial node in initial list:
7         while frontier not empty:
8             node = frontier take value put in first to search
9
10            if node state == goal state:
11                # To check if the goal at the same with initial node
12                return Path is found
13            add node to explored
14            for each neighbour in neighbours list of node:
15                if neighbour not in frontier or explored
16                    if node state is goal state:
17                        return Path is found
18                add neighbour to frontier
19    return Path is not found

```

Figure 2: BFS Pseudocode

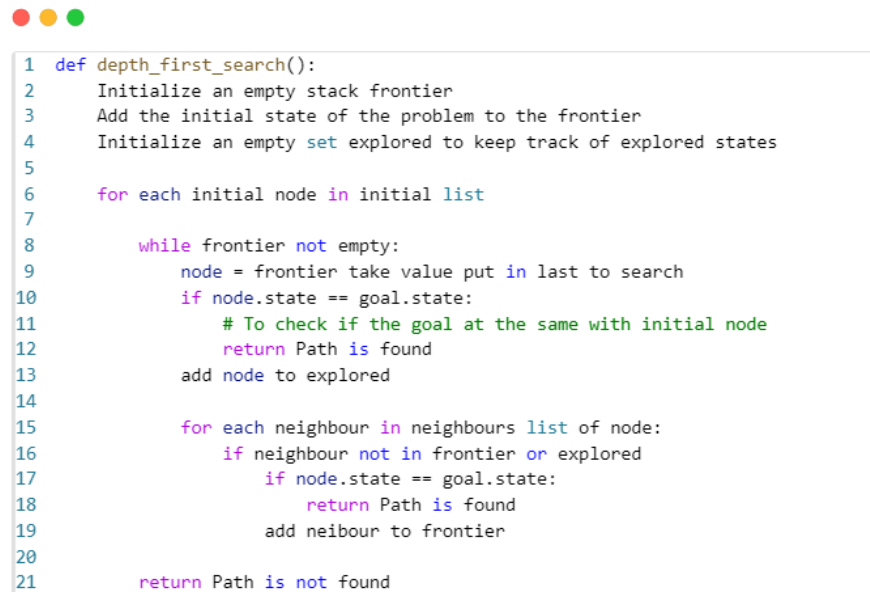
Depth-first search algorithms

Depth-first search (DFS) is an algorithm that explores each branch of a search tree as deeply as possible before backtracking. It begins at the root node and traverses to the deepest node where no successors exist. Then, it backtracks to explore the next branch, continuing this process until it finds the goal or runs out all nodes in the frontier.

Node expansion in DFS is managed using a Last-In-First-Out (LIFO) stack, which always pops and explores the most recently added node, pushing new nodes at the next depth level ($d + 1$) onto the stack.

Time complexity of DFS depends on the size of state space and in worst case scenario, the search branch b could expanded all of the node available in the search tree, which means DFS could search $O(b^d)$ where d is the maximum depth of any node.

Regarding space complexity, DFS stores only a single path from the initial node to the current node. Consequently, it requires memory for $O(bd)$ nodes, where b is the branching factor and d is the maximum depth of the tree.



```

1 def depth_first_search():
2     Initialize an empty stack frontier
3     Add the initial state of the problem to the frontier
4     Initialize an empty set explored to keep track of explored states
5
6     for each initial node in initial list
7
8         while frontier not empty:
9             node = frontier take value put in last to search
10            if node.state == goal.state:
11                # To check if the goal at the same with initial node
12                return Path is found
13            add node to explored
14
15            for each neighbour in neighbours list of node:
16                if neighbour not in frontier or explored
17                    if node.state == goal.state:
18                        return Path is found
19                    add neighbour to frontier
20
21    return Path is not found

```


Figure 3: DFS Pseudocode

Greedy Best-first search algorithms

Greedy Best-First Search (GBFS) is an algorithm that selects the node estimated to be closest to the goal using a heuristic function $f(n) = h(n)$, which means the heuristic function $h(n)$ estimates the cost from node n to the goal. GBFS expands the node with the lowest heuristic value, prioritizing paths that seem promising in terms of proximity to the goal. While GBFS can quickly find a solution, it does not guarantee that the solution is optimal.

The time complexity of GBFS is dependent on the quality of the heuristic function. In the worst case, if the heuristic function significantly overestimates the distance to the goal, GBFS may explore many unnecessary nodes, leading to exponential time complexity. However, with a well-crafted heuristic, GBFS can find solutions much faster than uninformed search algorithms, making it efficient in many practical scenarios. The time complexity in such cases is typically $O(b^d)$, where b is the branching factor and d is the depth of the solution.

The space complexity of GBFS is determined by the number of nodes it needs to store in memory during the search process. This complexity is generally similar to the time complexity, $O(b^d)$, as it must keep track of the frontier of nodes to be expanded. Therefore, although GBFS can be efficient in terms of time, it can still require substantial memory, especially for large and complex search spaces.



```

1 def greedy_best_first_search():
2     Initialize an empty priority queue frontier
3     Add the initial state of the problem to the frontier with priority determined by a heuristic function h
4     Initialize an empty set explored to keep track of explored states
5     Initialize heuristic function of estimated cost to reach goal
6     if node.state == goal.state:
7         # To check if the goal at the same with initial node
8         return Path is found
9     for each initial node in initial list
10
11     while frontier is not empty:
12         node = frontier take value with lower heuristic function in Priority Queue
13
14         if node.state == goal.state:
15             return Path is found
16         add node to explored
17
18         for each neighbour in neighbours list of node expanded:
19             if neighbour not in frontier or explored
20                 add neighbour to frontier
21             if heuristic of neighbour < heuristic of node in frontier with same state
22                 Update the priority of node in frontier with same state to h(neighbour)
23     return Path is not found

```

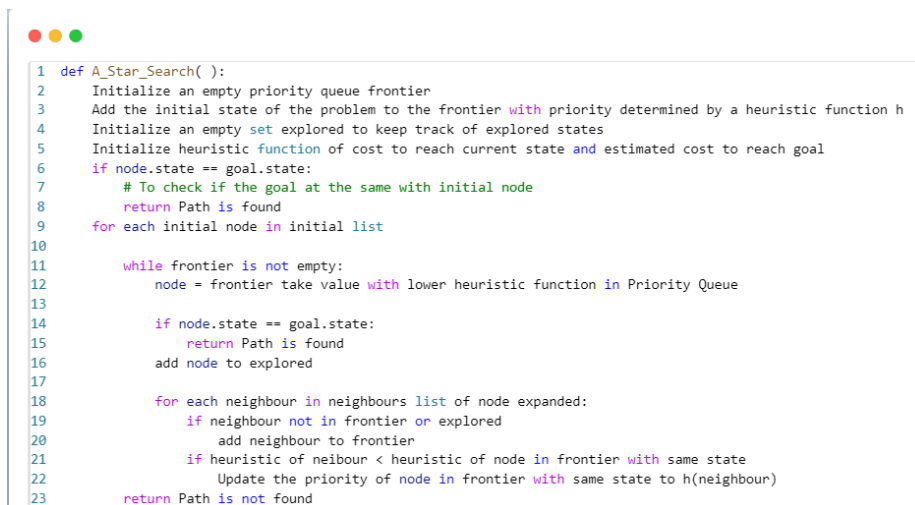
Figure 4: GBFS Pseudocode

A star (A) search algorithms*

A* (Astar) builds as an extension of the principles of Greedy Best-First Search (GBFS). Unlike GBFS, which focuses solely on minimizing the estimated distance to the goal, A* combines both the actual cost to reach a node from the start $g(n)$ and the estimated cost from that node to the goal $h(n)$. The evaluation function $f(n) = g(n) + h(n)$ ensures that the search not only moves towards the goal but also considers the path cost incurred so far. This combination of distance and cost estimation allows A* to find the lowest cost of path to the goal, often resulting in optimal and efficient search performance.

The time complexity of the A* algorithm depends on several factors, including the accuracy of the heuristic function and the structure of the search space. In the best-case scenario, where the heuristic function $h(n)$ perfectly estimates the cost to the goal and is consistent, A* can explore the search space efficiently with a time complexity of $O(bd)$, where b is the branching factor and d is the depth of the optimal solution. However, in the worst-case scenario, where the heuristic is poor or the search space is highly branching, A* can degrade to an exponential time complexity of $O(b^d)$. This occurs because the algorithm may end up exploring a significant portion of the search space, especially when the goal is far from the start or when many paths have similar costs.

A* search algorithm's space complexity is influenced by the number of nodes it needs to keep in memory during the search process. In general, A* maintains a frontier of nodes to be explored, stored in a priority queue sorted by the $f(n)$ values. This means that, in the worst case, the algorithm could require memory proportional to the number of nodes in the explored region of the search space. Specifically, the space complexity is $O(b^d)$, where b is the branching factor and d is the depth of the optimal solution. This high memory requirement can be a significant limitation for A*, especially in large or highly branching search spaces, as it needs to store all the generated nodes in memory until the goal is found. Nonetheless, managing memory efficiently remains a crucial aspect of implementing A* in practice, particularly for applications involving large search spaces.



```

1 def A_Star_Search( ):
2     Initialize an empty priority queue frontier
3     Add the initial state of the problem to the frontier with priority determined by a heuristic function h
4     Initialize an empty set explored to keep track of explored states
5     Initialize heuristic function of cost to reach current state and estimated cost to reach goal
6     if node.state == goal.state:
7         # To check if the goal at the same with initial node
8         return Path is found
9     for each initial node in initial list
10
11     while frontier is not empty:
12         node = frontier take value with lower heuristic function in Priority Queue
13
14         if node.state == goal.state:
15             return Path is found
16         add node to explored
17
18         for each neighbour in neighbours list of node expanded:
19             if neighbour not in frontier or explored
20                 add neighbour to frontier
21             if heuristic of neighbour < heuristic of node in frontier with same state
22                 Update the priority of node in frontier with same state to h(neighbour)
23     return Path is not found

```

Figure 5: Astar Pseudocode


Bi-Directional search algorithms

Bi-Directional Search is a search algorithm that simultaneously operates from both the starting node and the goal node, progressing towards each other with the aim of intersecting at a common node in frontier. It explores nodes from both ends until they meet each other in the middle. The search method used in this algorithm is inherited from the breadth-first search. However, instead of finding a path from start to goal, it seeks to find intersecting paths from both ends by meeting somewhere in between.

Compared to traditional breadth-first search, Bi-Directional Search exhibits significant runtime efficiency improvements. In breadth-first search, reaching the goal node requires expanding all nodes up to depth d , resulting in a time complexity of $O(b^d)$, where b represents the branching

factor, and d is the depth to the goal node. In contrast, the bi-directional approach divides the search space into two smaller sub-problems, significantly reducing the required search depth. By intersecting the search paths from both the start and goal, it effectively halves the depth of the search space. Consequently, the time complexity for bi-directional search is reduced to $O(b^{d/2})$, offering a substantial improvement over traditional methods.

In addition to its time efficiency, Bi-Directional Search also benefits from reduced space complexity. By limiting the search depth to $d/2$ on each side, the algorithm decreases the number of nodes that need to be stored in memory. This results in a space complexity of $O(b^{d/2})$, which is significantly lower than the $O(b^d)$ space requirement of a full breadth-first search. This reduction in memory usage makes Bi-Directional Search a highly efficient option for solving problems with large search spaces, where memory management is a critical concern.



```

1 def bi_directional():
2     Initialize a list of initial nodes and list of goal nodes
3     Initialize an empty queue frontier for initial and goal
4     Add the initial state of the problem to the initial frontier and goal state to goal frontier
5     Initialize an empty set explored to keep track of explored states
6
7     for each initial_node in initial_nodes:
8         if node.state == goal.state:
9             # To check if the goal at the same with initial node
10            return Path is found
11
12    while initial_frontier and goal_frontier are not empty:
13        initial_node = initial_frontier take value put in first to search
14        Add initial_node to explored
15
16        goal_node = goal_frontier take value put in first to search
17        Add goal_node to explored
18
19        for each neighbour in neighbours list of initial_node expanded:
20            if neighbour not in initial_frontier or explored:
21                Add neighbour to initial_frontier
22
23        for each neighbour in neighbours list of goal_node expanded:
24            if neighbour not in goal_frontier or explored:
25                Add neighbour to goal_frontier
26
27        if initial_frontier and goal_frontier intersection:
28            #Check if 2 frontier have the same element
29            return Path is found
30
31    return Path is not found

```

Figure 6: Bidirectional BFS Pseudocode

Bi-Directional with A search algorithms.*

Bi-Directional with A* Search is a technique that combines the principles of bi-directional search with the heuristic-based exploration of the A* algorithm. Unlike traditional bi-directional search, which typically uses breadth-first search, this method employs A* from both the start and goal nodes, integrating heuristic functions to guide the search process. The algorithm concurrently

expands nodes from both directions, aiming to intersect in the middle and effectively reduce the search space.

This advanced search technique merges the efficiency of bi-directional search with the informed pathfinding of A*. It explores the search space from both ends using A*'s heuristic function, which estimates the cost to the goal. By doing so, it prioritizes paths that are more likely to lead to the goal, thereby reducing the number of nodes explored. Although this method guarantees completeness, meaning it will find a solution if one exists, it does not always ensure optimality. The intersection check in bi-directional A* cannot always guarantee the shortest path, unlike a single-directional A* search that expands based on the minimal cost to the goal.

Despite its potential for suboptimal solutions, Bi-Directional with A* Search offers significant advantages in terms of execution speed and memory efficiency. The heuristic function helps to narrow down the search space more effectively than traditional bi-directional search, leading to a faster solution. Additionally, because it reduces the depth of the search by dividing it between two directions, it requires less memory compared to other exhaustive search methods. This makes it a practical choice for large-scale problems where computational resources are a critical concern.

```

1 def bi_directional_astar():
2     Initialize a list of initial nodes and list of goal nodes
3     Initialize an empty Priority Queue frontier for initial and goal
4     Add the initial state of the problem to the initial frontier and goal state to goal frontier
5     Initialize an empty set explored to keep track of explored states
6     Initialize heuristic function of cost to reach current state and estimated cost to reach goal for both initial and goal nodes
7
8     for each initial_node in initial_nodes:
9         if node.state == goal.state:
10             # To check if the goal at the same with initial node
11             return Path is found
12
13     while initial_frontier and goal_frontier are not empty:
14         initial_node = initial_frontier take value put in first to search
15         Add initial_node to explored
16         for each neighbour in neighbours list of initial_node expanded:
17             if neighbour not in initial_frontier or explored:
18                 Add neighbour to initial_frontier
19
20         for each neighbour in neighbours list of initial_node expanded:
21             if neighbour not in initial_frontier or explored:
22                 add neighbour to initial_frontier
23             if heuristic of neighbour < heuristic of initial_node in initial_frontier with same state:
24                 Update the priority of initial_node in initial_frontier with same state to h(neighbour)
25
26     if initial_frontier and goal_frontier intersection:
27         #Check if goal_frontier contains initial_node
28         return Path is found
29
30     goal_node = goal_frontier take value put in first to search
31     Add goal_node to explored
32     for each neighbour in neighbours list of goal_node expanded:
33         if neighbour not in goal_frontier or explored:
34             add neighbour to goal_frontier
35         if heuristic of neighbour < heuristic of goal_node in goal_frontier with same state:
36             Update the priority of goal_node in goal_frontier with same state to h(neighbour)
37
38     if initial_frontier and goal_frontier intersection:
39         #Check if initial_frontier contains goal_node
40         return Path is found
41

```

Figure 7: Bidirectional A star Pseudocode

V. Intended Learning Outcomes (ILO)

Understand Key Search Algorithms:

- Gain a comprehensive understanding of fundamental search algorithms such as Breadth-First Search (BFS), Depth-First Search (DFS), Greedy Best-First Search (GBFS), A* (A-star), and custom algorithms like Bi-Directional BFS and Bi-Directional A*. Understand their principles, applications, strengths, and limitations in the context of robotic navigation in a grid-based environment.

Develop and Implement Search Algorithms:

- Develop proficiency in designing and implementing various search algorithms using Python. Apply these algorithms to enable a robot to autonomously navigate in a grid environment, avoiding obstacles and finding the optimal path to a specified goal.

Understand Search Mechanisms Beyond Unit Content:

- Acquire the ability to understand and implement search mechanisms that extend beyond the course content, allowing for the exploration and application of advanced algorithmic techniques in diverse contexts.

Analyze and Evaluate Algorithm Performance:

- Analyze the time and space complexity of different search algorithms. Evaluate their performance in terms of runtime efficiency, memory usage, and solution optimality, especially in scenarios involving large and complex grid environments.

Design and Implement a Graphical User Interface (GUI):

- Gain experience in designing and implementing a graphical user interface using PyGame. This includes visualizing the robot's navigation process on a grid, handling user input, and displaying real-time updates of the robot's pathfinding progress.

VI. Summary

The **Robot Navigation Search** project for **COS30002 - AI for Games** aims to implement various search algorithms, including BFS, DFS, GBFS, A*, Bi-Directional BFS, and Bi-Directional A*, to enable a robot to navigate a grid efficiently. Using Python 3.12 and PyGame for a graphical interface, the project evaluates each algorithm's time and space complexity, demonstrating their application in finding optimal paths while avoiding obstacles. The project combines practical programming skills with theoretical analysis, providing a comprehensive understanding of AI techniques in game development and robotic navigation, emphasizing efficiency, documentation, and ethical considerations.

VII. Acknowledgement

This project owes a great deal to the invaluable insights provided by the book *"Artificial Intelligence: A Modern Approach"* (4th edition), which significantly deepened my understanding of various algorithms and their implementations. And I also appreciate the COS30002 - AI for Games course from James Bonner, with insightful content and practical with strong guidance from tutor class.

VIII. References

Russell, S., & Norvig, P. (2020). *Artificial Intelligence: A Modern Approach* (4th ed.). Pearson.