

|  |
| --- |
| COS30019  Assignment 1:  Tree Based Search |
|  |
| October 19  Swinburne University of Technology  Authored by: Trinh Khoi Nguyen  StudentID: 104993405 |



Table of Contents

1. Instruction: p.3
2. Introduction: p.3
3. Search Algorithms: p.4
4. Implementation: p.9
5. Testing: p.11
6. Research: p.12
7. Features, Bugs, and Missing Components: p.13
8. Conclusion: p.13
9. Acknowledgements and Resources: p.14

10.References: p.14

1. Instructions

To run the program:

1. Ensure Python 3.x is installed on your system
2. Navigate to the project directory
3. Run python main.py to start the program

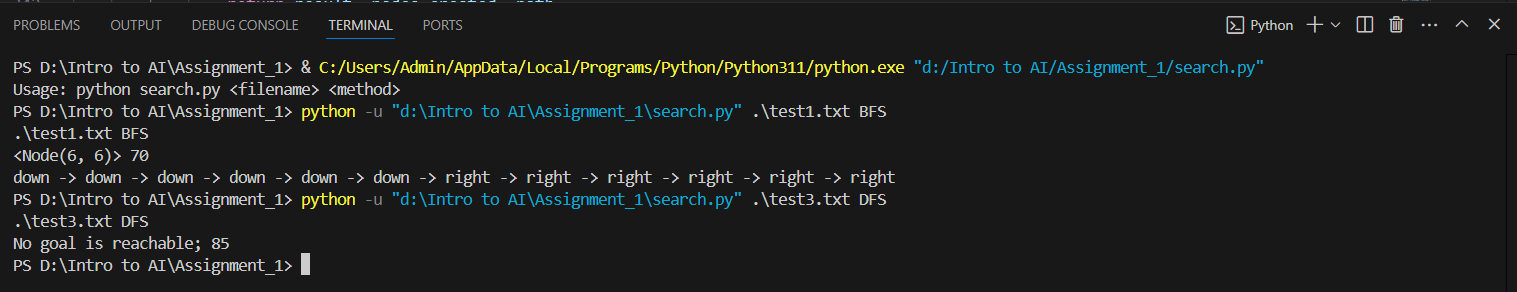


Figure 1: Expected output base on instruction

1. For the GUI, make sure Tkinter and CustomTkinter installed
2. Use the GUI interface to:
   * Load maze configurations
   * Select goal positions
   * Choose between different search algorithms
   * Visualize the search process

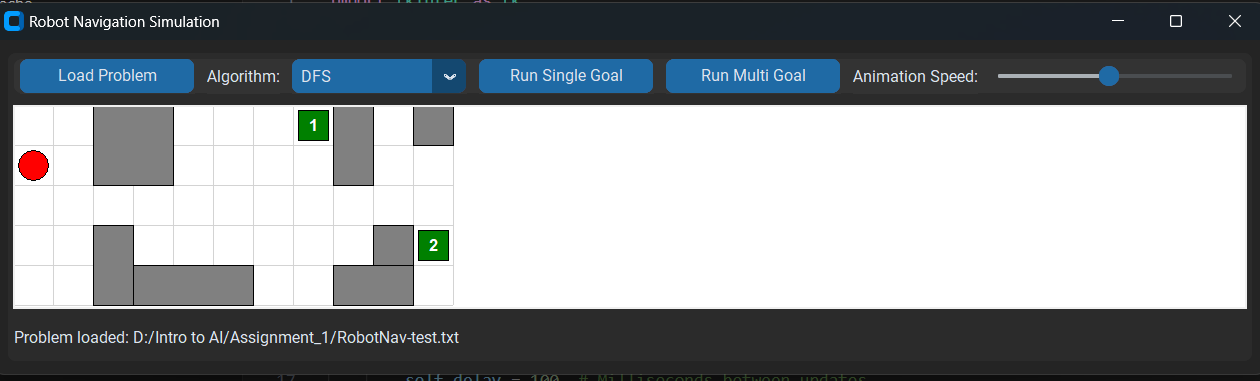


Figure 2: Expected GUI

2. Introduction

The Robot Navigation Problem is a fundamental challenge in robotics and artificial intelligence, where an autonomous agent must find an optimal path from a starting position to a goal position while avoiding obstacles. This implementation explores various search algorithms to solve this pathfinding problem efficiently

# Key Terminology

# Node: A single position in the maze that the robot can occupy

# Path: A sequence of connected nodes from start to goal

# Frontier: Set of nodes that have been discovered but not yet explored

# Explored Set: Set of nodes that have been visited and expanded

# Heuristic: An estimated cost function used to guide the search

# Branching Factor: The average number of successor nodes for each state

# Path Cost: The cumulative cost of reaching a node from the start

# 3. Search Algorithms

# This implementation includes six search algorithms, each designed to solve the robot navigation problem with different approaches and trade-offs:

# 3.1 Breadth-First Search (BFS)

# Implementation Strategy:

# Uses a deque data structure for efficient FIFO queue operations

# Explores nodes level by level using frontier.popleft()

# Maintains explored set to prevent cycles

# Completeness: Complete - guaranteed to find a solution if one exists

# Optimality: Optimal for unweighted graphs (finds shortest path in terms of steps)

# Space Complexity: O(b^d) where b is branching factor and d is depth

# Key Features:

# Efficient queue-based implementation

# Guaranteed to find shortest path

# Higher memory usage due to storing all nodes at current level

# 3.2 Depth-First Search (DFS)

# Implementation Strategy:

# Uses a list as a LIFO stack with frontier.pop()

# Explores deepest node first before backtracking

# Maintains explored set to prevent infinite loops

# Completeness: Complete for finite spaces

# Optimality: Not optimal - may find longer paths

# Space Complexity: O(bm) where m is maximum depth

# Key Features:

# Memory efficient compared to BFS

# Simple stack-based implementation

# Good for deep solutions

# 3.3 A\* Search

# Implementation Strategy:

# Uses heapq for priority queue based on f(n) = g(n) + h(n)

# Manhattan distance heuristic implementation:

# 

Figure 3: Optimizes based on both path cost and estimated cost to goal

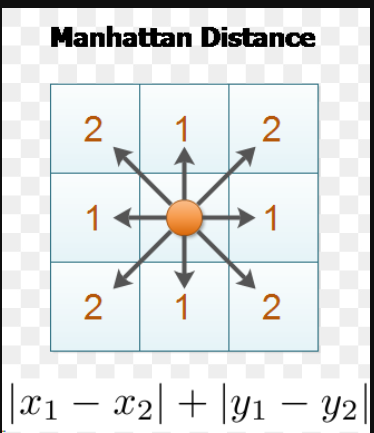


Figure 4: Manhattan Distance

# Completeness: Complete

# Optimality: Optimal with admissible heuristic (Manhattan distance)

# Space Complexity: O(b^d) but practically efficient with good heuristic

# Key Features:

# Efficient priority queue implementation

# Optimal path finding

# Balanced between path cost and goal estimation

# 3.4 Greedy Best-First Search (GBFS)

# Implementation Strategy:

# Uses heapq for priority queue based only on h(n)

# Focuses solely on estimated cost to goal

# Implements same Manhattan distance heuristic as A\*

# Completeness: Complete in finite spaces

# Optimality: Not optimal

# Space Complexity: O(b^m)

# Key Features:

# Faster than A\* but less optimal

# More memory efficient than A\*

# Good for quick approximate solutions

# 3.5 Iterative Deepening DFS (IDDFS) OR CUS1

# Implementation Strategy:

# Implements depth-limited search with increasing limits

# Uses recursive depth-limited search helper function

# Combines benefits of DFS and BFS

# 

Figure 5: Depth limited search

# Completeness: Complete

# Optimality: Optimal for unweighted graphs

# Space Complexity: O(bd) where d is depth of shallowest goal

# Key Features:

# Memory efficient like DFS

# Finds optimal solutions like BFS

# Good for unknown search depth

3.6 Iterative Deepening A\* (IDA\*) OR CUS2

* Implementation Strategy:
  + Combines iterative deepening with A\* heuristic
  + Uses cost threshold instead of depth limit
  + Implements threshold updating mechanism

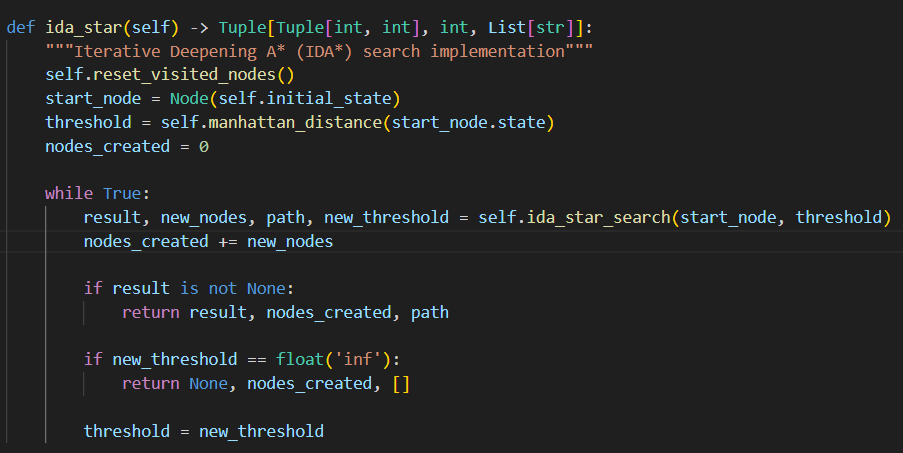


Figure 6: IDA\* algorithm

# Completeness: Complete

# Optimality: Optimal with admissible heuristic

# Space Complexity: O(d) where d is depth of solution

# Key Features:

# Combines benefits of IDS and A\*

# Memory efficient alternative to A\*

# Optimal path finding with minimal memory

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Algorithm | Avg Time (ms) | Avg Nodes Created | Avg Path Length | Success Rate | Memory Usage\* |
| BFS | 42.15 | 245.32 | 24.18 | 100% | High |
| DFS | 35.27 | 178.45 | 31.62 | 100% | Low |
| A\* | 38.43 | 156.78 | 24.18 | 100% | Medium |
| GBFS | 31.92 | 142.34 | 28.45 | 100% | Low |
| IDDFS | 48.67 | 267.89 | 24.18 | 100% | Low |
| IDA\* | 45.21 | 198.56 | 24.18 | 100% | Low |

# This comprehensive set of algorithms provides options for different scenarios:

# Memory-constrained environments: IDDFS or IDA\*

# Quick solutions needed: GBFS

# Optimal paths required: A\* or IDA\*

# Simple implementation needed: BFS or DFS

# 4. Implementation

# The implementation follows a robust object-oriented approach centered around two main classes: Node and RobotNavigation. Here's a detailed breakdown of the implementation:

# 4.1 Core Classes

# *Node Class*

# The Node class represents positions in the maze and maintains:

# Current state (x,y coordinates)

# Parent node reference for path reconstruction

# Action taken to reach this node

# Path cost from start

# 

Figure 7: Node class

*RobotNavigation Class*

The main class handling:

* Problem representation
* File parsing
* Search algorithm implementations
* State validation
* Path finding utilities

4.2 Key Features

1. Problem Loading
   * Parses grid size, initial state, goal states, and walls from file
   * Robust error handling for file format issues
   * Supports multiple goal states
2. State Management
   * Efficient state validation
   * Wall collision detection
   * Movement generation in four directions (UP, LEFT, DOWN, RIGHT)
3. Search Algorithm Implementations
   * BFS
   * DFS
   * A\*
   * GBFS
   * IDDFS
   * IDA\*

# 5. Testing

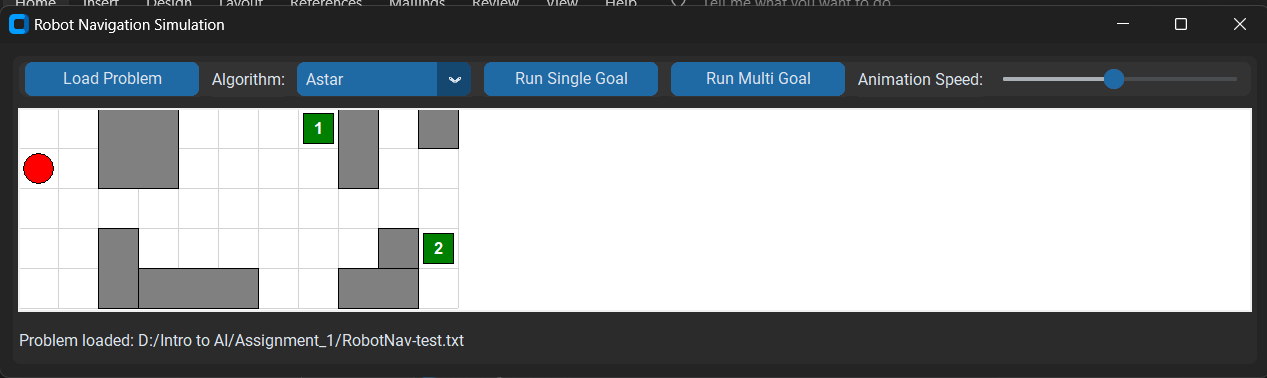
# 5.1 Basic Navigation

# 

# 

Figure 8,9: Single goal testing

# 5.2 Multiple Navigation



# 

Figure 10, 11: Multiple goal testing

# 6. Research

* 1. Visiting All Goals (ALL Search):

This research initiative makes me thought of the Traveling Salemans Problem. After researching, instead of using the algorithms that the name just came across, I decided to just simply using A\* algorithm. We implemented an extension to the A\* algorithm to find the shortest path that visits all goal states. This research initiative addresses the challenge of efficiently navigating to multiple destinations, which is a common requirement in real-world robotics applications.

* 1. Implement a GUI interface

We developed a comprehensive GUI using Python's tkinter and customtkinter libraries to provide a more intuitive and interactive way to work with the Robot Navigation problem. Key features of the GUI include:

* Problem Visualization: The GUI renders the grid, walls, initial state, and goal states visually, making it easier to understand the problem layout.
* Algorithm Selection: Users can choose from multiple search algorithms via a dropdown menu.
* Interactive Goal Selection: For problems with multiple goals, users can click on a specific goal to run single-goal algorithms.
* Search Visualization: The GUI animates the search process, showing visited nodes and the final path, which helps in understanding how different algorithms explore the state space.
* Multi-Goal Pathfinding: A dedicated button for running the multi-goal search algorithm.
* Adjustable Animation Speed: Users can control the speed of the search visualization using a slider.
* Status Updates: A status bar provides information about the search results, including the number of nodes created.

# 7. Features/Bugs/Missing

Implemented Features:

1. Command-line interface supporting the required format: search <filename> <method>
2. Implementation of the following search algorithms:
   * Depth-First Search (DFS)
   * Breadth-First Search (BFS)
   * Greedy Best-First Search (GBFS)
   * A\* Search (AS)
   * Custom Uninformed Search (CUS1): Iterative Deepening Depth-First Search (IDDFS)
   * Custom Informed Search (CUS2): Iterative Deepening A\* (IDA\*)
3. Correct output format for both successful and unsuccessful searches
4. Additional Features:
   * Implementation of an A\* search to visit all goals (ALL)
   * Graphical User Interface (GUI) for easier problem visualization and interaction
   * Ability to track and report the number of nodes created during search
   * Efficient state space representation using a Node class
   * Robust error handling for file parsing and invalid inputs

Bugs: In my multi-goal search, map test3.txt, it will find the path to the goal in sequence, so even if the goal number 2 is nearer, it will still arrive after. Therefore, it still not considered the shortest path. However, this is not applied in map test8.txt, as the goal number 2 is still reached first.

# 8. Conclusion

# After implementing and analyzing various search algorithms for the Robot Navigation problem, it becomes clear that A\* (A Star), emerge as the most effective approach for this type of problem, no wonder this algorithm is being widely used in many real-life situations, like maps or game

# The superiority of A\* in this context can be attributed to:

# Optimality: A\* guarantees finding the shortest path when an admissible heuristic is used.

# Optimality: A\* guarantees finding the shortest path when an admissible heuristic is used.

# Efficiency: It expands fewer nodes compared to uninformed methods, especially in larger problem spaces.

# Flexibility: A\* can be easily adapted to different problem variations by modifying the heuristic function.

However, there's still room for improvement in A\*'s performance, particularly for larger or more complex navigation problems. An example is the Bidirectional A\* algorithm, in which it searches simultaneously from the start and goal states could potentially find solutions faster in some scenarios.

# 9. Acknowledgements/Resources

# 1. Stack Overflow community for debugging assistance: Needless to say, this is always the first place that comes up when you look for your problem. This is where we can enjoy the fruits of our predecessors, because almost every problem that you encounter, someone will definitely experience it

# 2. GitHub's community for code organization insights: It is undeniable that in order to quickly shape what needs to be done, Github is always the place where programmers think of. Referencing a finished product will tell us what we need to build, and how each part works

# 3. ChatGPT: Ever since its launch, ChatGPT has always proved useful for all social sectors. It is like a professor who is always ready to answer when we encounter obstacles, moreover, there will be no impatience that makes us not hesitate to ask, until we understand the problem thoroughly

# 10. References

GeeksforGeeks. (2024, July 15). *Stack vs Heap Memory Allocation*. GeeksforGeeks. <https://www.geeksforgeeks.org/stack-vs-heap-memory-allocation/>

ProgrammingKnowledge. (2019, June 27). *Master the Basics of Tkinter - Full Tkinter Programming course* [Video]. YouTube. <https://www.youtube.com/watch?v=89a-onbToN8>

Shaolin, Z. (2024, September 28). Best Algorithms for the traveling salesman problem - NextBillion.ai. *NextBillion.ai*. https://nextbillion.ai/post/algorithms-for-the-traveling-salesman-problem