**COS30019 – Introduction to Artificial Intelligence**

**Assignment 1 – Tree-Based Search – The Robot Navigation**

Student Name: Nguyen Dinh Dung

Student ID: 104772138

A black and red sign with white text

Description automatically generated

Table of Contents

[Instructions 3](#_Toc180089834)

[Introduction 4](#_Toc180089835)

[Search Algorithm 6](#_Toc180089836)

[Testing 13](#_Toc180089837)

[Features/Bugs/Missing 21](#_Toc180089838)

[Research 22](#_Toc180089839)

[Conclusion 26](#_Toc180089840)

[References 27](#_Toc180089841)

# Instructions

1. Python Installation:

* Get the most recent version of Python from https://www.python.org/downloads/, the official Python website.
* Ensure that the "Add Python to PATH" option is chosen throughout the installation process. This crucial step allows you to run Python commands directly from your terminal or command prompt.

1. Visual Studio Code Installation:

* Visit the official website, https://code.visualstudio.com/, to download and install Visual Studio Code.

1. Python Extension Installation:

* Get Visual Studio Code started.
* Use the keyboard shortcut Ctrl+Shift+X to open the Extensions window, or click the Extensions icon in the Activity Bar.
* Look for "Python" in the marketplace and download the official Microsoft Python extension.
* To confirm that Python has been installed, open a new terminal in Visual Studio Code by selecting Terminal -> New Terminal.
* Run the python --version command. The installed version of Python ought should appear here.
* Run the pip —version command. This ought to show the pip version, which is the Python package installer.
* Creating a Python Project: • We can establish a specific folder for the Python project in VS Code.
* Use File => Open Folder in VS Code to access this folder.
* Inside the folder, create a new Python file (such as main.py).
* Running the code:
* Open a terminal and navigate to the folder where the Python files are located.
* To execute the search algorithm, use the following command format:

python search.py <file\_path> <search\_method>

For example, to run the BFS (Breadth-First-Search) algorithm on RobotNav-test.txt, type:

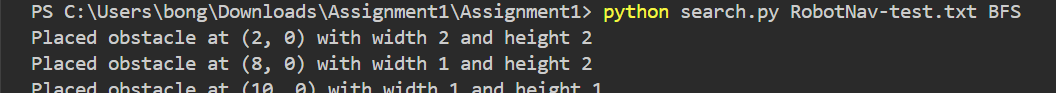


Figure 1. Example

Replace <search\_method> with one of the following options to see different search algorithms:

1. DFS for Depth-First Search
2. BFS for Breadth-First Search
3. GBFS for Greedy Best-First Search
4. AS for A\* Search
5. CUS1 for Custom Search 1
6. CUS2 for Custom Search 2

# Introduction

**1. Problem:**

In order to address the Robot Navigation Problem—a significant obstacle in robotics and artificial intelligence—this paper investigates tree-based search techniques. We examine two custom strategies (CUS1, CUS2) that are suited to certain restrictions, as well as both informed (GBFS, A\*) and uninformed (DFS, BFS) search approaches. We assess their effectiveness via trials, pointing out their advantages and disadvantages in various situations and talking about possible lines of inquiry for enhancing robot navigation systems.

The goal of the Robot Navigation Problem is to navigate a robot through an NxM grid that has walls (shown by "W") surrounding it, where N > 1 and M > 1. Starting from a predetermined starting point (defined as "R"), the robot endeavors to reach one of multiple target points (defined as "G"). Finding the robot's most effective route while dodging obstacles is the issue at hand.

A grid with a green square

Description automatically generated

Figure 2.Robot\_map

**2. Terminology:**

* Robot navigation: the process of guiding a robot across a space in order to arrive at a predetermined location.
* Grid: A two-dimensional depiction of the surroundings, with cells that could be blocked or vacant.
* Node: An area in the grid that is often designated by its coordinates (x, y).
* Edge: A connection that denotes a potential path of movement between two nodes.
* Path: The set of nodes the robot travels through to go from one place to another.
* Search Algorithm: A technique for methodically examining grid paths in order to identify a navigational solution.
* Uninformed Search: Algorithms (like DFS and BFS) that search for paths without knowing anything further about the surroundings or the objective.
* Informed Search: Algorithms (like GBFS and A\*) that make use of supplementary data or heuristics to more effectively direct the search towards the objective.
* Depth-First Search (DFS): An erroneous method that gives deeper path exploration precedence over backtracking.
* Breadth-First Search (BFS): Before going on to the next layer, an ignorant algorithm explores every surrounding node at a specific level.
* Greedy Best-First Search (GBFS): An intelligent algorithm determines the next node to be selected by estimating how close it is to the objective.
* A Search (A\*): A well-informed method that combines the cost of traveling to a node with an approximation of the cost needed to reach the objective.
* Custom Search 1 (CUS1): Iterative Deepening Depth-First Search (IDDFS) is the foundation of this unique, uninformed search approach.
* Custom Search 2 (CUS2): A dynamic heuristic-based custom-informed search technique built on A\* Search.
* Heuristic: A function that estimates the cost of the least expensive route from a node to the objective. used to direct the process of an informed search.
* Goal Node: The intended location that the robot is trying to get to.
* Start Node: The robot's starting location inside the grid.
* Obstacle: A grid cell that the robot needs to avoid that is impenetrable.
* Cost: The amount that goes into traveling between nodes. This could stand for steps, distance, or other pertinent information.
* Open Set: (Informed search) The set of nodes that are being considered for assessment.
* Closed Set: (Informed search) The group of nodes that have already undergone assessment.
* Priority Queue: A data structure (informed search) that uses heuristic cost to determine priority while choosing the next node.
* Visited: A list of nodes previously investigated to keep track of advancement and avoid revisiting.
* Parent Node: The node in the path that comes right before the current node.
* Leaf Node: A childless node that may represent the objective or a dead end.
* Traversal: The process of examining grid nodes.
* Memory Usage: The amount of memory that the algorithm uses while it is running.
* Time Efficiency: How long does it take the algorithm to solve a problem?

# Search Algorithm

|  |  |  |
| --- | --- | --- |
| Search Strategies | Description | Method |
| Uninformed | | |
| Depth-First Search | Select one option, try it, and go back when there are no more options. | DFS |
| Breadth-First Search | Extend every option one step at a time. | BFS |
| Informed | | |
| Greedy Best-First Search | Use only the cost to reach the goal from the current node or the cost to reach this node to evaluate the node. | GBFS |
| A\* Search | Use both the cost to reach the goal from the current node and the cost to reach this node to evaluate the node. | AS |
| Custom | | |
| Custom search 1 | An uninformed approach to figuring out how to get to the goal. The uniform-cost search (UCS) is an unacceptable custom search approach for this situation. | CUS1 |
| Custom search 2 | An informed approach way to find out the shortest path with the least moves to reach the goal. | CUS2 |

**1. Depth-First Search:**

**Depth-first search** **(DFS)** is a method for exploring a tree or graph. In a DFS, you go as deep as possible down one path before backing up and trying a different one. Depth-first search is like walking through a corn maze. You explore one path, hit a dead end, and go back and try a different one... This process is often implemented using recursion or a stack to keep track of visited nodes.

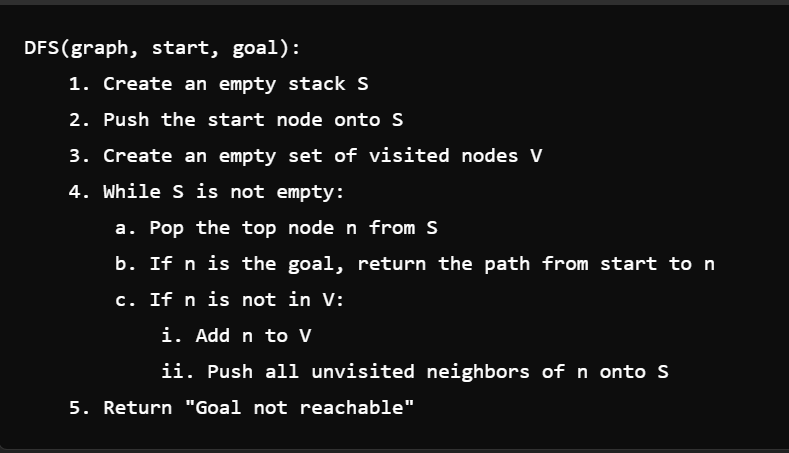


Figure 3. Pseudo Code

**2. Breadth-first Search:**

**Breadth First Search (BFS) :**An essential graph traversal algorithm is called Breadth First Search (BFS). It starts at a node and initially moves through all of its neighboring nodes. Their neighborings are traversed once every adjacent has been visited. The closest vertices are visited first in this case, which sets it apart from DFS. Mostly, we go level by level through the vertices. BFS is the foundation for many well-known graph algorithms, including Prim's algorithm, Kahn's Algorithm, and Dijkstra's shortest path. In addition to many other issues, BFS itself may be used to identify cycles in both directed and undirected graphs and locate the shortest path in an unweighted graph.

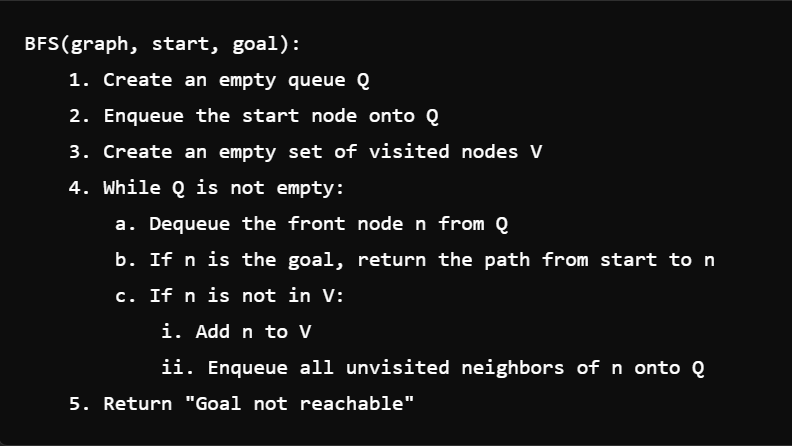


Figure 4. Pseudo Code

**3. Greedy Best-first Search:**

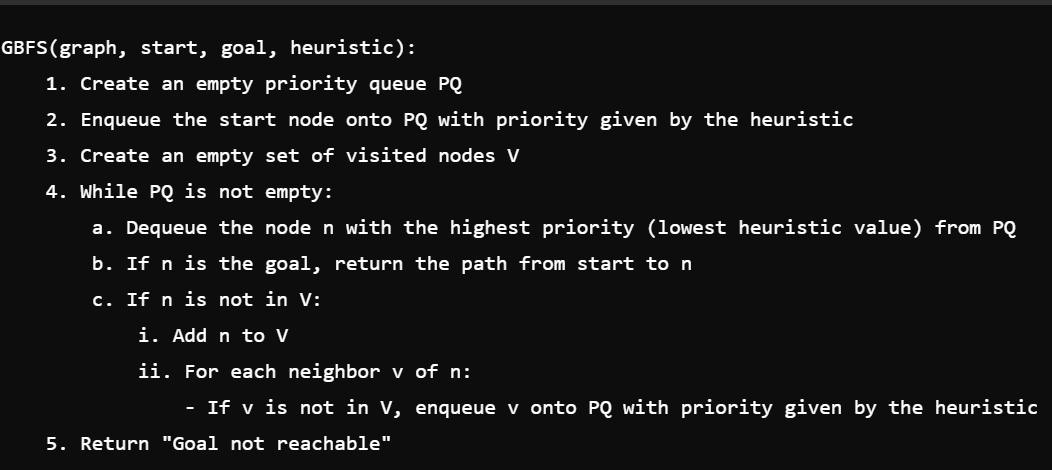
**Greedy Best-First Search (GBFS):** An AI search method called Greedy Best-First Search looks for the most promising route from a given starting point to a destination. Whether or not they are the shortest way, it gives priority to the routes that seem the most promising. The method expands the path with the lowest cost after calculating the costs of each feasible path. Until the objective is accomplished, this process is repeated.  


Figure 5. Pseudo Code

**4. A\* Search:**

**A\* (A-star):** An approach used extensively in computer pathfinding and graph traversal is called A\*, which is pronounced as "A star". Plotting a walkable path between several nodes, or points, on the graph is made efficient by the procedure. Creating a lowest-cost path tree from the start node to the target node is how A\* operates. A\* differs and performs better for many searches because it uses a function f(n)f(n) for each node, which provides an estimate of the overall cost of a path using that node. As a result, A\* is a heuristic function rather than an algorithm because a heuristic is more of an estimate and isn't always verifiably accurate.

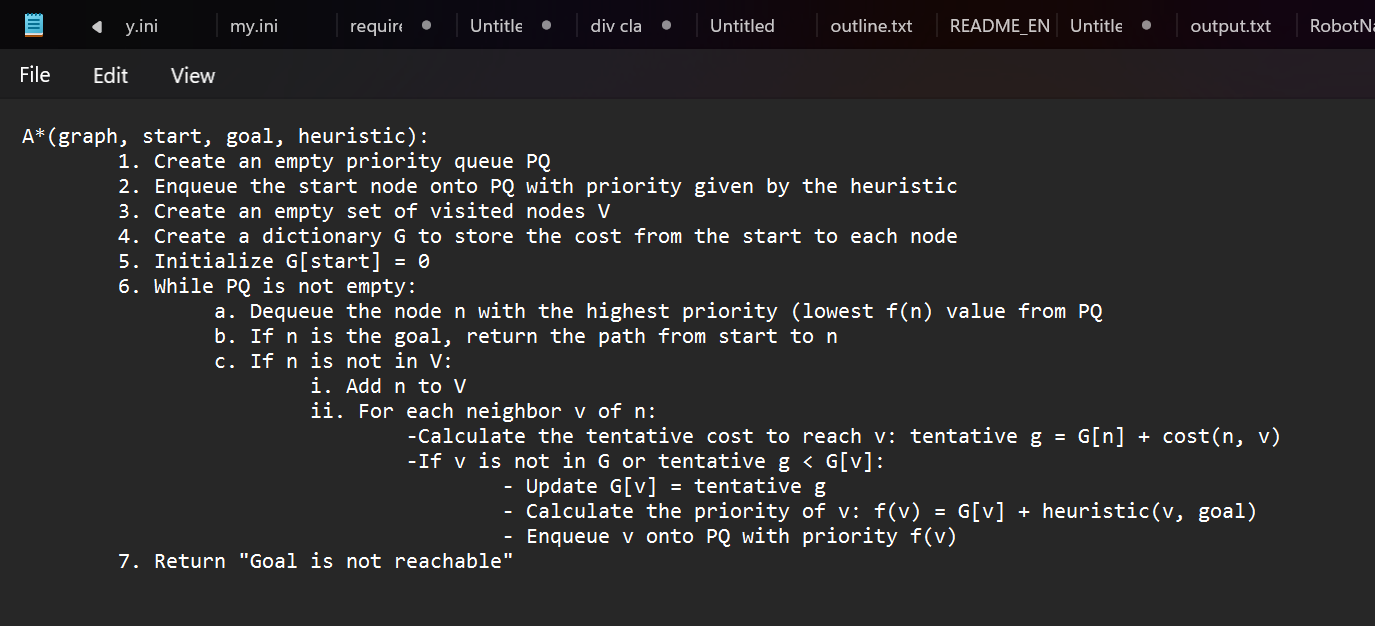


Figure 6. Pseudo Code

**5. CUS1:**

**Custom Search 1 (CUS1)** is a pathfinding method that effectively finds the shortest route by combining a bidirectional search strategy with Iterative Deepening Depth-First Search (IDDFS). To make sure that the shortest path, if one exists, is found, IDDFS gradually investigates the search space at ever deeper depths. By concurrently examining routes from the start and goal nodes, bidirectional search improves efficiency and greatly minimizes the search space and time complexity. Because of its hybrid methodology, CUS1 is especially good at resolving pathfinding issues in intricate settings where the optimal path length is ambiguous.

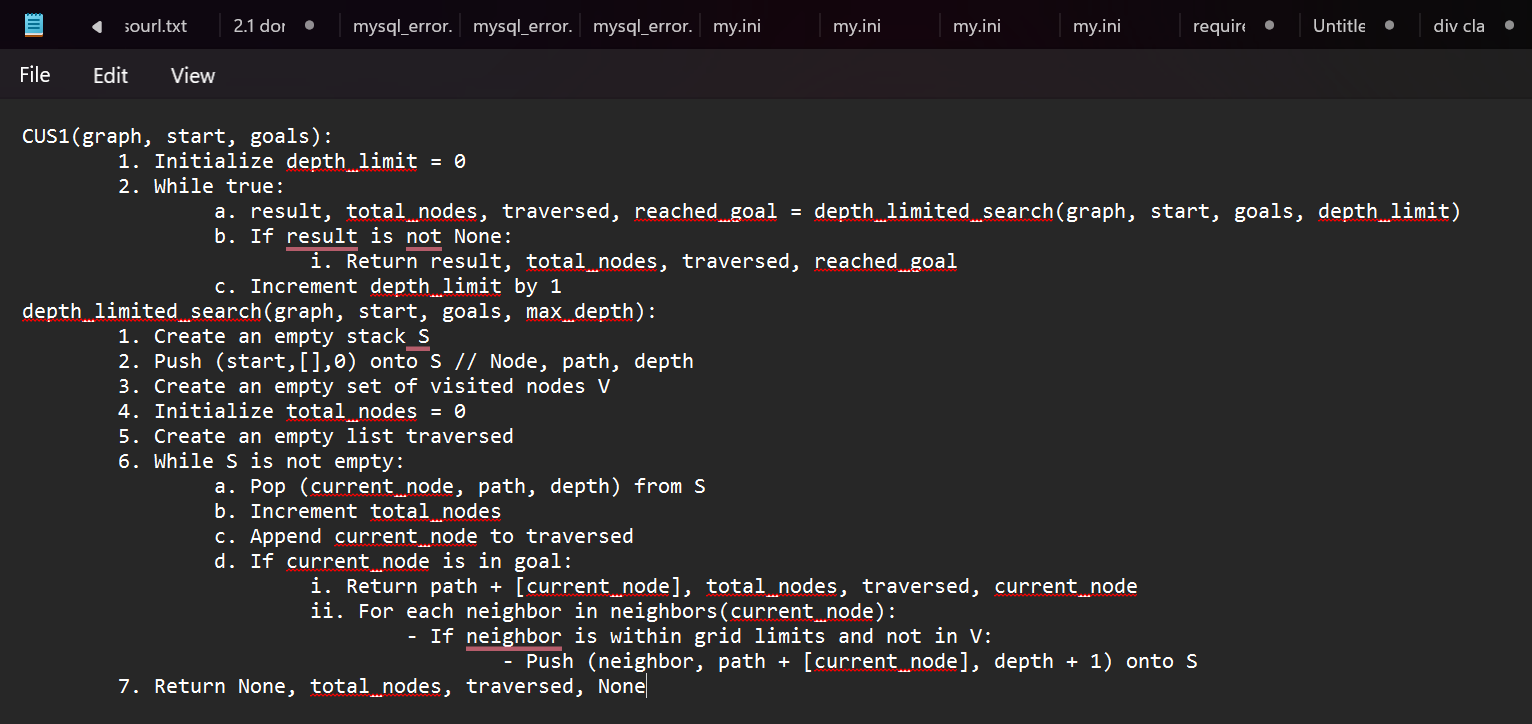


Figure 7. Pseudo Code

**6. CUS2:**

**Custom Search 2 (CUS2**) is a smarter pathfinding algorithm that builds on the A\* search by dynamically adjusting how it estimates the distance to the goal. This adaptability helps it find the best paths faster, especially in large and complex areas with changing obstacles. CUS2 starts by prioritizing paths based on an initial estimate, but as it explores, it constantly refines this estimate to focus on the most promising directions. This makes it efficient and reliable at finding the shortest or near-shortest paths, even in challenging situations

A screenshot of a computer screen

Description automatically generated

Figure 8. Pseudo Code

Execution

**1. Depth-First Search:**

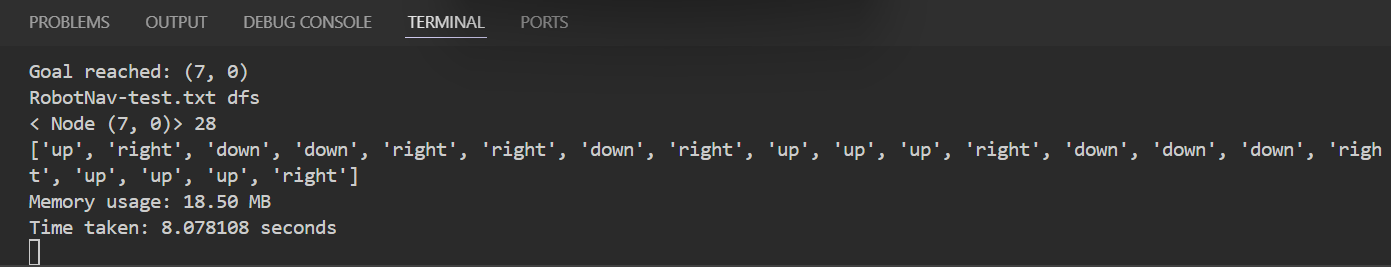


Figure 9. DFS result

A screenshot of a computer screen

Description automatically generated

Figure 10. DFS path

**2. Breadth-first Search:**

A screen shot of a computer

Description automatically generated

Figure 11. BFS result

A screenshot of a computer

Description automatically generated

Figure 12. BFS path

**3. Greedy Best-First Search:**

A screen shot of a computer

Description automatically generated

Figure 13. GBFS result

A screenshot of a computer

Description automatically generated

Figure 14. GBFS path

**4. A\* Search:**

A screen shot of a computer

Description automatically generated

Figure 15. A\* result

A screenshot of a computer

Description automatically generated

Figure 16. A\* path

**5. CUS1:**

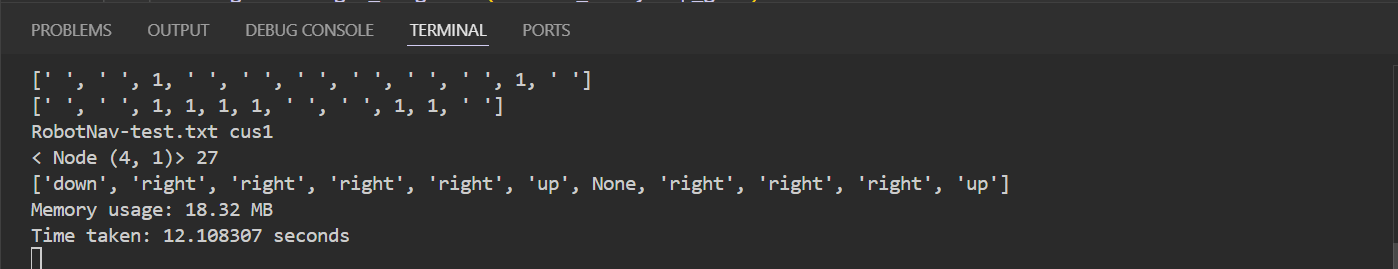


Figure 17. CUS1 result

A screenshot of a computer

Description automatically generated

Figure 18. CUS1 path

**6. CUS2:**

A screen shot of a computer

Description automatically generated

Figure 19. CUS2 result

A screenshot of a crossword puzzle

Description automatically generated

Figure 20. CUS2 path

# Testing

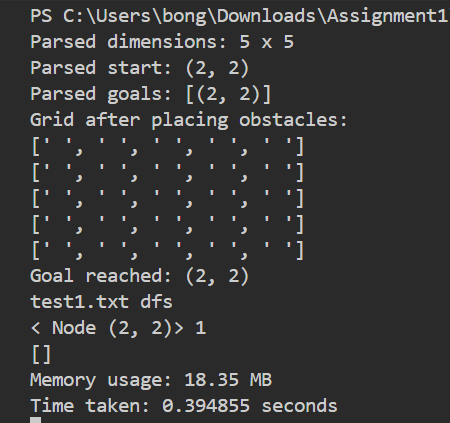
**A. Overview Test Cases:**

To evaluate the efficacy of the six search strategies (DFS, BFS, GBFS, A\*, CUS1, and CUS2), ten unique test situations were created. These scenarios include simple routes, environments with lots of obstacles, several target locations, and special situations where the start and goal spots are the same. The test examples are intended to highlight the benefits and drawbacks of every algorithm in various scenarios. Use the same format that was used during the instruction session to administer the test.

For example, **python search.py test1.txt DFS** to run DFS on test 1

**B. Test Cases:**

1. Same Beginning and Goal:

 A screenshot of a computer

Description automatically generated A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated A screenshot of a computer

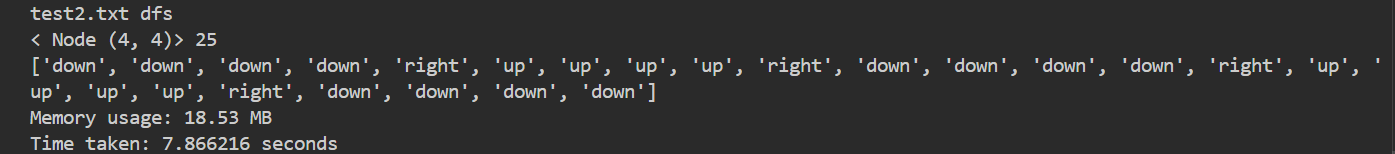
Description automatically generated A screenshot of a computer

Description automatically generated

CONCLUSION

|  |  |  |  |
| --- | --- | --- | --- |
| Test Case | Description | Best Performers | Observations |
| Same Begin/ Goal | Start and goal at the same point in an open grid. | All methods | Demonstrates efficiency in trivial cases. |

1. Wide Grid



A black background with white text

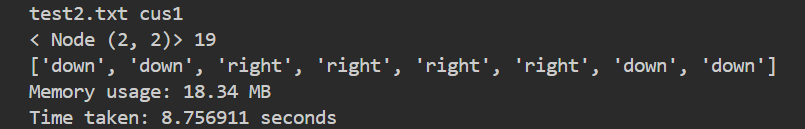
Description automatically generated

A black background with white text

Description automatically generated

A black background with white text

Description automatically generated



A black background with white text

Description automatically generated

CONCLUSION

|  |  |  |  |
| --- | --- | --- | --- |
| Test Case | Description | Best Performers | Observations |
| Wide Grid | 5x5 grid with no obstacles. | BFS, A\*, GBFS, CUS2 | BFS and informed searchers are more efficient. |

1. Many Barriers:

A black background with white text

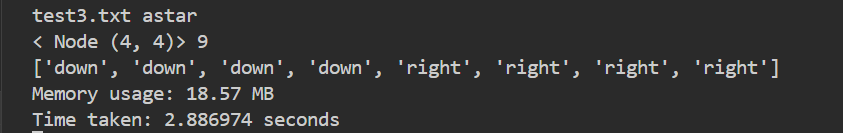
Description automatically generated

A black background with white text

Description automatically generated

A black background with white text

Description automatically generated



A black background with white text

Description automatically generated

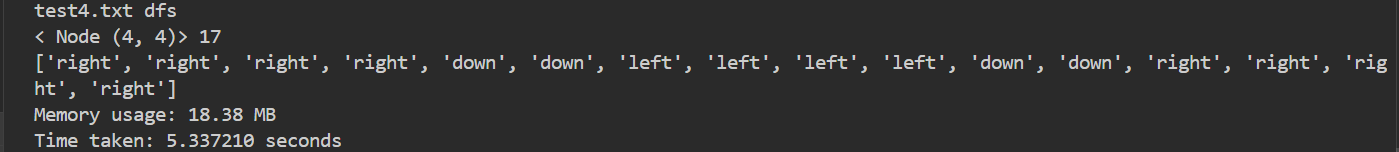
A black background with white text

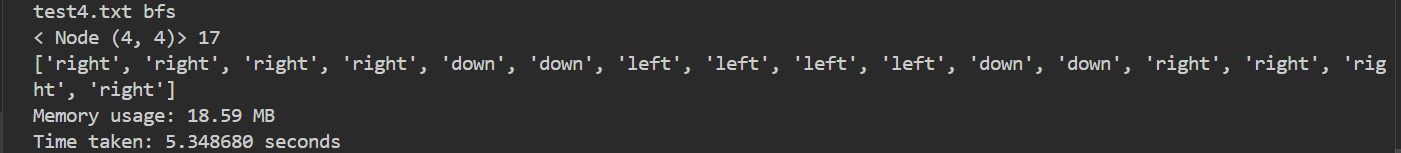
Description automatically generated

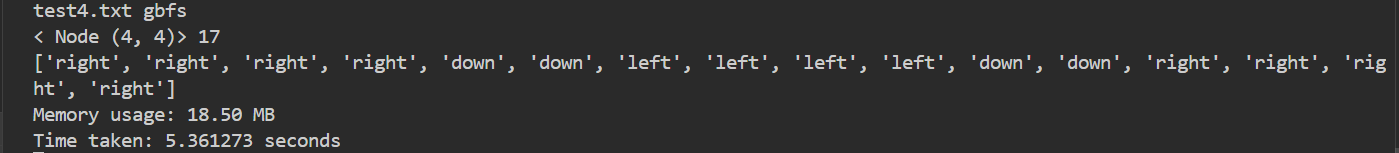
CONCLUSION

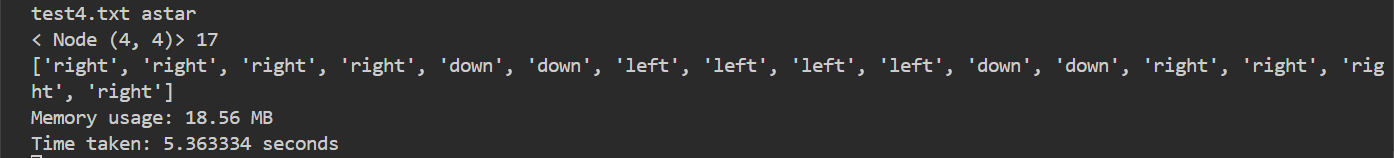
|  |  |  |  |
| --- | --- | --- | --- |
| Test Case | Description | Best Performers | Observations |
| Many Barriers | 5x5 grid with complex obstacle arrangement. | A\*, CUS2 | A\* and CUS2 excel in complex grids due to their efficiency and optimal pathfinding. |

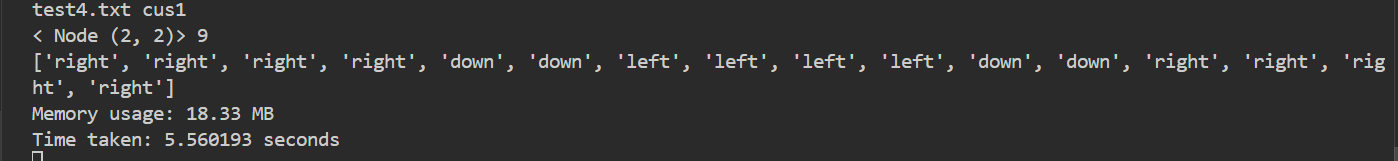
1. Limited Routes:

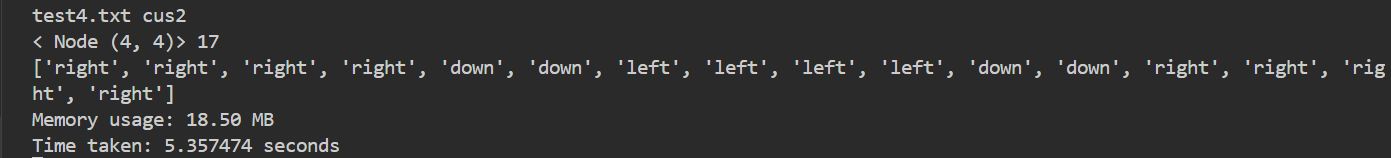












CONCLUSION

|  |  |  |  |
| --- | --- | --- | --- |
| Test Case | Description | Best Performers | Observations |
| Limited Routes | 5x5 grid with narrow corridors. | BFS, A\*, GBFS, CUS2 | BFS and informed searches are more effective. |

1. Many Goals

A black screen with white text

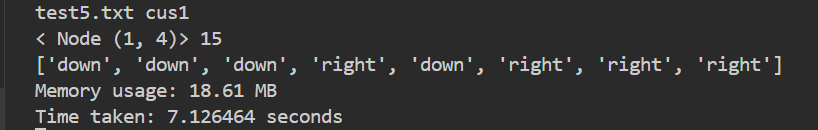
Description automatically generated A black background with white text

Description automatically generated

A black screen with white text

Description automatically generated A screenshot of a computer

Description automatically generated



A black background with white text

Description automatically generated

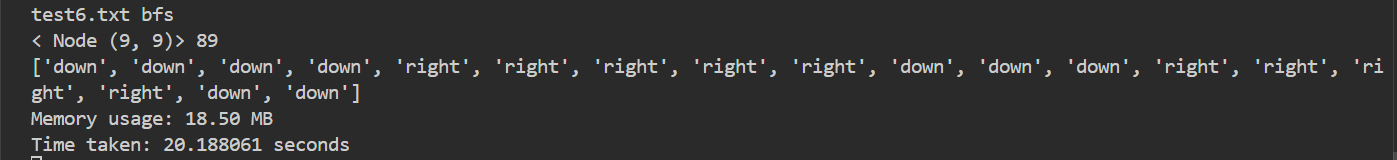
CONCLUSION

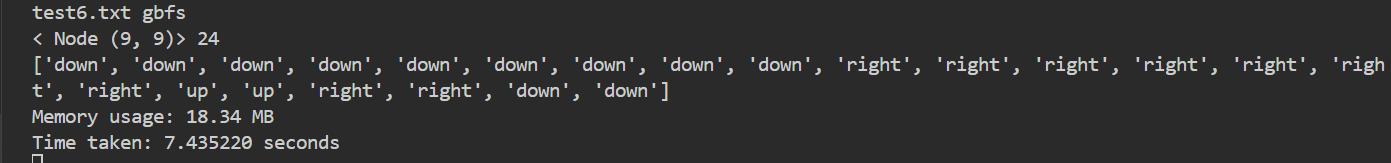
|  |  |  |  |
| --- | --- | --- | --- |
| Test Case | Description | Best Performers | Observations |
| Many Goals | 5x5 grid multiple goal nodes | BFS, A\*, CUS2 | BFS and A\* are reliable, and CUS2 provides adaptive efficiency. |

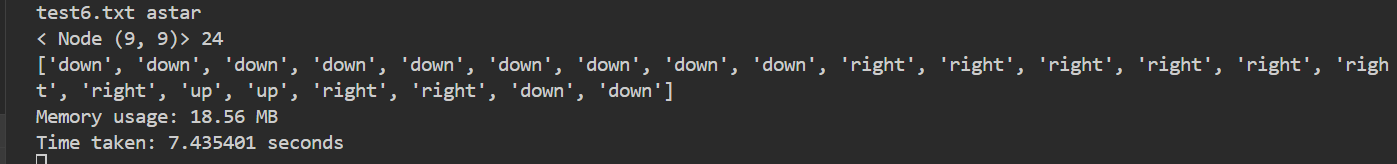
1. Wide Grid with Dispersed Barriers

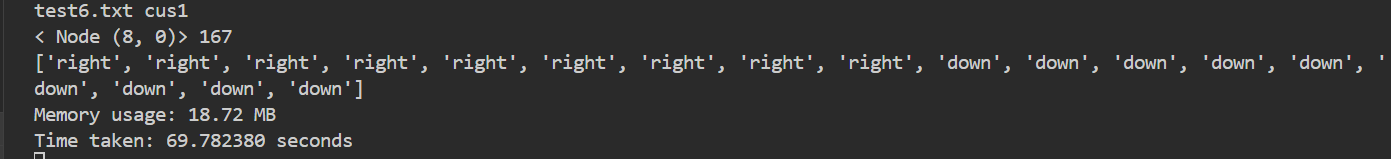
A black background with white text

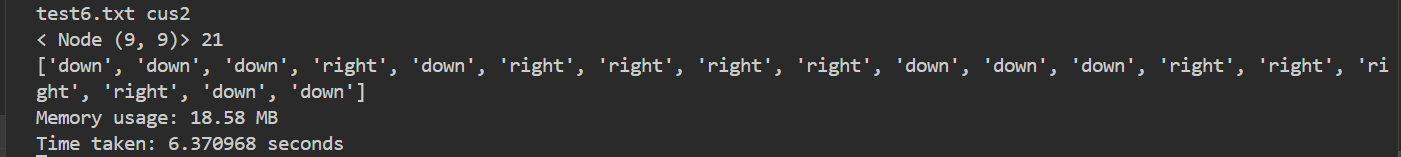
Description automatically generated











CONCLUSION

|  |  |  |  |
| --- | --- | --- | --- |
| Test Case | Description | Best Performers | Observations |
| Wide Grid with Dispersed Barriers | 10x10 with scattered obstacles. | A\*, CUS2 | A\* and CUS2 are best for large grids with scattered obstacle. |

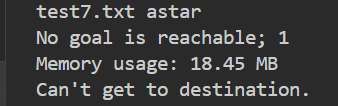
1. Dense Barriers Near the Beginning:

A screen shot of a computer

Description automatically generated A screen shot of a computer

Description automatically generated A screenshot of a computer screen

Description automatically generated

 A screenshot of a computer screen

Description automatically generated A screenshot of a computer

Description automatically generated

CONCLUSION

|  |  |  |  |
| --- | --- | --- | --- |
| Test Case | Description | Best Performers | Observations |
| Dense Barriers | 5x5 grid with dense obstacles near the start. | BFS, A\*, GBFS, CUS2 | BFS, A\*, and CUS2 handle dense obstacles near the start effectively. |

1. Objectives on the Grid's Opposite Sides

A black background with white text

Description automatically generated

A black background with white text

Description automatically generated

A black background with white text

Description automatically generated

A black background with white text

Description automatically generatedA black background with white text

Description automatically generated

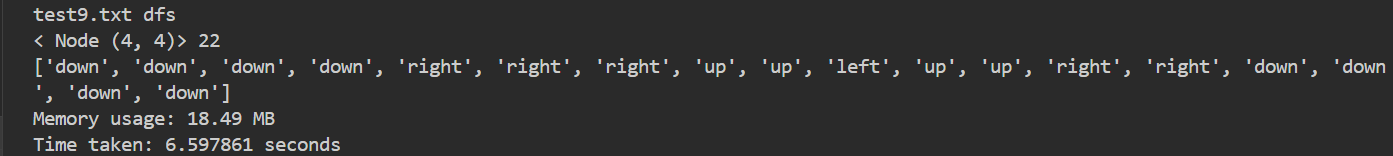
A black screen with white text

Description automatically generated

CONCLUSION

|  |  |  |  |
| --- | --- | --- | --- |
| Test Case | Description | Best Performers | Observations |
| Opposite Goals | 5x5 grid goals on opposite sides. | A\*,CUS2 | A\*and CUS2 are effective for distant goals. |

1. Random Barriers:



A black background with white text

Description automatically generated

A black background with white text

Description automatically generated

A black background with white text

Description automatically generated

A black background with white text

Description automatically generated

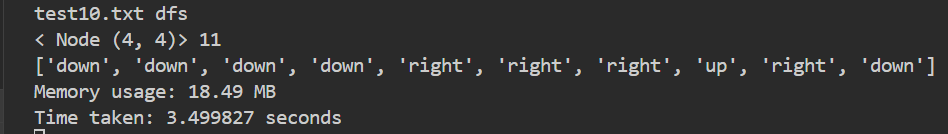
A black background with white text

Description automatically generated

CONCLUSION

|  |  |  |  |
| --- | --- | --- | --- |
| Test Case | Description | Best Performers | Observations |
| Random Barriers | 5x5 with random obstacles. | A\*,CUS2, BFS | BFS, A\*, and CUS2 are robust in random obstacle scenarios. |

1. Complex Grid:



A black background with white text

Description automatically generated

A black background with white text

Description automatically generated

A black background with white text

Description automatically generated

A black background with white text

Description automatically generated

A black background with white text

Description automatically generated

CONCLUSION

|  |  |  |  |
| --- | --- | --- | --- |
| Test Case | Description | Best Performers | Observations |
| Complex Grid | 5x5 grid representing a complex maze with dead-ends. | BFS, A\*, CUS2 | BFS, A\*, and CUS2 excel in complex mazes. |

**C. Test report:**

|  |  |
| --- | --- |
| **Algorithm** | **Recommendation** |
| **A\*** | The best overall choice for efficiency and finding the shortest path in various scenarios. |
| **BFS** | The best for simple and open grids where the shortest path is the primary concern. |
| **CUS2** | The best for complex and obstacle-rich grids. |
| **DFS** | Not recommended for most robot navigation tasks due to inefficiency. |

# Features/Bugs/Missing

**1. Implemented Features:**

* Depth-First Search (DFS): This method uses a recursive approach to thoroughly explore routes.
* Breadth-First Search (BFS): This method uses a queue to systematically investigate nodes one level at a time.
* A\* Search: Makes use of a priority queue to rank nodes according to a combination of the real path cost and an approximative estimate of the distance to be reached.
* Greedy Best-First Search (GBFS): This method ranks nodes closer to the objective by using a priority queue that is exclusively based on a heuristic function.
* Custom search algorithms (CUS1, CUS2): specialized algorithms or modified heuristic implementations created to meet certain pathfinding criteria.

**2. Features Not Implemented:** The evaluation's specified algorithms (DFS, BFS, A\*, GBFS, CUS1, and CUS2) have all been implemented successfully.

**3. Bugs:**

* **Memory Usage Reporting:** Certain algorithms may have erroneous reported memory use due to limitations in Python's memory profiling tools.
* **Name changing problems:** After doing this report I realize my naming conventions is wrong and I have to change all the name to uppercase so my new input and new output will look a little bit different from this report. For example this is the input and output after I make my changes:



(input)

A black background with white text

Description automatically generated

(output)

You can see that the BFS has been written in uppercase. However, this won’t affect any testing functions or the program. But just keep in mind that you need to give uppercase input: DFS, BFS, GBFS, AS, CUS1, CUS2.

**4. Evaluation Focus:**

A range of pathfinding scenarios, such as open grids, grids with numerous obstacles, narrow paths, multiple goals, large grids with scattered obstacles, dense obstacles near the start, goals on opposite sides of the grid, random obstacles, and intricate mazes were used to evaluate each algorithm. It is possible to determine which algorithms are most suited for particular situations and issues thanks to this examination.

# Research

**A. Euclidean Distance:**

According to Dong-hyung Kim's research, informed search algorithms can be greatly enhanced for robot navigation tasks by integrating Euclidean distance, which is the straight-line distance between two points. It is feasible to estimate the shortest path from the robot's current position to the target more accurately by integrating this distance as a heuristic.

You can modify current algorithms in the following ways:

1. A\* Search (A\*): The Euclidean distance, in addition to the cost (g(n)) of traveling from the beginning to the current node, can be added to the heuristic function (h(n)) of the A\* algorithm. In doing so, it may be possible for A\* to discover the shortest path more quickly by giving priority to the most promising nodes for research.
2. Greedy Best-First Search (GBFS): GBFS concentrates on nodes that are geometrically closest to the objective by utilizing the Euclidean distance as its sole heuristic (h(n)). This frequently leads to quicker goal discovery and fewer examined nodes.

In order to implement, one must compute the Euclidean distance (d) between the robot's current location (x1, y1) and the target (x2, y2) using the following formula:

d = √ ((x2 - x1)^2 + (y2 - y1)^2)

Below is the test of A\* Search applying both Euclidean and Manhattan distances:

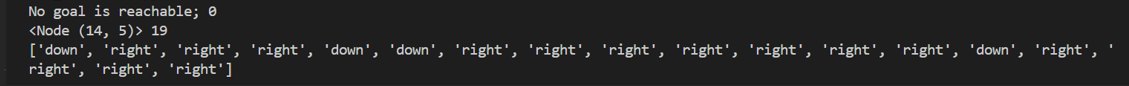


Figure 21. Manhattan Distance

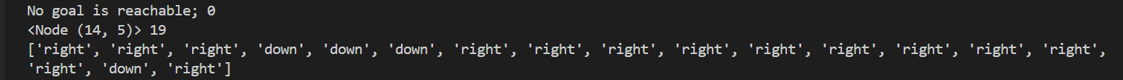


Figure 22. Euclidean Distance

A black background with white letters

Description automatically generated

Figure 23. New grid for the test

Based on the test, choosing between Euclidean and Manhattan distances for pathfinding depends on the mobility limitations of your environment:   
**The Euclidean Distance**• Fit for situations where there is unrestricted mobility in all directions, including diagonals.   
• Benefit: By taking diagonal and straight routes into consideration, maybe shorter paths are found.   
• Limitation: It may not adequately show the actual path length in grid-based environments with movement restrictions.   
**Manhattan's Distance**  
• Fits well in Grid-based settings where movement is restricted to the horizontal and vertical axes.   
• Benefit: Provides precise pathfinding within grid restrictions, identifying the best routes while taking the limits into account.   
• Limitation: Since it disregards the potential of such shortcuts, this may lead to less-than-ideal courses when diagonal movement is permitted.

**Conclusion:**

* **Grid-based environments:** Manhattan distance is ideal for precise and practical routes.
* **Unrestricted environments:** For perhaps shorter, direct paths, use Euclidean distance.

**B. Jump Action**

**Overview**

This paper describes how navigation software for a robot was improved to include jump movements. This allowed the robot to avoid obstacles and maybe find quicker courses in grid environments. Jump\_up(n), Jump\_down(n), Jump\_left(n), and Jump\_right(n) are the four jump actions that we have created. The integer n indicates how many squares jumped. Given that the cost of a jump is exponential in the form of 2^(n-1), lengthier jumps become more costly.

**How to Run the Code**

1. **Prerequisites:** Ensure you have Python installed on your system.
2. **Command Line Usage:** Navigate to the directory containing the code (search.py) and execute the following command:  
   python search.py <file\_path> <search\_method> <use\_jumps>

* <file\_path>: The path to the input file describing the grid environment (e.g., "RobotNav-test.txt").
* <search\_method>: The search algorithm to use (DFS, BFS, GBFS, AS, CUS1, CUS2).
* <use\_jumps>: Whether to enable jump actions (true or false).
* For examples:

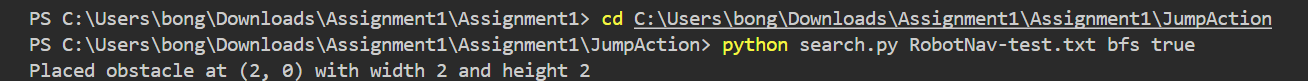


Figure 24. New grid for the test

**Implementation Specifics**

* **Neighbor Exploration:** Adapted to take regular and leap movements into account while deciding the robot's next course of action.
* **Heuristic Adjustments:** To account for the higher cost of leaps, informed search algorithms (A\*, CUS2) were modified.
* **GUI Improvement:** To improve user observation, a delay was added and jump actions were visualized.

**Results and Comparison**

The outcomes demonstrated that the robot can find shorter courses when jump actions are enabled, particularly in areas with lots of barriers. However, because each jump's cost and viability must be evaluated, this improvement comes at the expense of longer computation times.

**Research and Improvements**

The study also looked at dynamic jump costs, in which the price of a jump changed based on how many obstacles were in the path. This may increase the robot's ability to identify the best path by allowing it to choose when to employ jumps. Furthermore, more study was done on optimizing heuristics—basic guidelines that assist the algorithm in making decisions.

**Summary**

All things considered, the robot navigation software has been much enhanced by the addition of jump operations. Now, the robot is capable of more effective navigation in a larger variety of conditions.

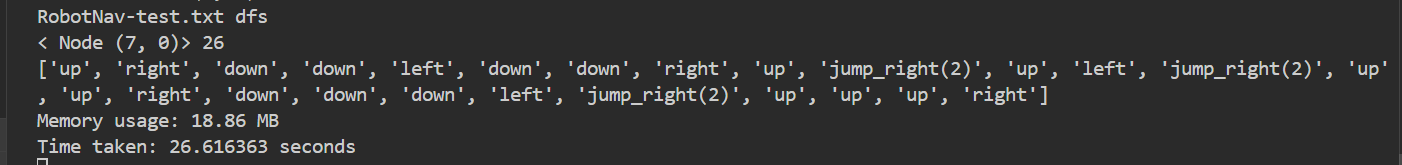


Figure 25. DFS jump action

A black background with white text

Description automatically generated

Figure 26. BFS jump action

A screen shot of a computer

Description automatically generated

Figure 27. GBFS jump action

A screenshot of a computer

Description automatically generated

Figure 28. AS jump action

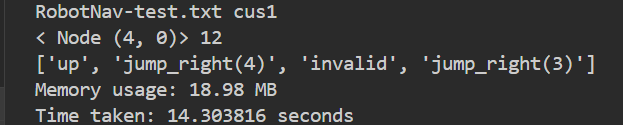


Figure 29. CUS1 jump action

A screenshot of a computer program

Description automatically generated

Figure 30. CUS2 jump action

# Conclusion

This study investigated several search strategies to address the challenge of robot navigation in a grid system. Depth-First Search (DFS), Breadth-First Search (BFS), A\* Search, Greedy Best-First Search (GBFS), and two customized strategies (CUS1 and CUS2) were the algorithms that were tested. Performance metrics centered on memory use and time efficiency. The most effective search, according to the results, was A\* Search since it combined practical efficiency with ideal pathfinding. It worked well in a range of circumstances and reliably found the shortest path. Conversely, DFS was the least effective, sometimes leading to sluggish searches and excessive memory usage, particularly in bigger grids or longer paths.

Moreover, the Euclidean distance heuristic was applied to raise the effectiveness of knowledgeable search algorithms like A\* and GBFS, drawing inspiration from Dong-hyung Kim's (2013) research. This method produced shortest-path estimations that were enhanced, leading to more effective search operations.

This project investigated the incorporation of jump motions into the robot's mobility capabilities in addition to comparing algorithms. In difficult situations, the search algorithms found shorter paths by letting the robot leap over barriers. However, because each jump's cost and viability had to be assessed, this resulted in an increase in computing overhead. The study also looked into the application of dynamic jump costs, which may help the algorithms operate more efficiently in environments with lots of obstacles.

In conclusion, this study offers helpful information on the advantages and disadvantages of several search algorithms for robot navigation, assisting in the selection of the optimal method for particular use cases. Further advancements in robot navigation strategies could be demonstrated by incorporating jump actions and investigating dynamic jump costs. In order to further maximize the efficiency of these algorithms in intricate and dynamic situations, future research might concentrate on improving the jump cost model and investigating other heuristics.

# References

Departament de Matemàtiques. (n.d.). A\* Algorithm pseudocode. Bellaterra; Universitat Autònoma de Barcelona.

Khim, J., Jennison, K., & Moore, K. (n.d.). *Depth-first search (DFS): Brilliant math & science wiki*. Brilliant. https://brilliant.org/wiki/depth-first-search-DFS/

Kim, D., et al. (2013). Sensor-based navigation of a car-like robot based on Bug family algorithms. *Proceedings of the Institution of Mechanical Engineers, Part C: Journal of Mechanical Engineering Science*, 227(6), 1224-1241.

Microsoft. (2021, November 3). *Get started tutorial for python in visual studio code*. RSS. https://code.visualstudio.com/docs/python/python-tutorial

Bhadaniya, S. (2023a, November 13). *Depth first search in Python (with code): DFS algorithm*. FavTutor. https://favtutor.com/blogs/depth-first-search-python

Bhadaniya, S. (2023b, December 1). *Breadth first search in Python (with code): BFS algorithm*. FavTutor. https://favtutor.com/blogs/breadth-first-search-python

GeeksforGeeks. (2024a, January 18). *Greedy best first search algorithm*. https://www.geeksforgeeks.org/greedy-best-first-search-algorithm/

GeeksforGeeks. (2024b, April 17). *A\* search algorithm in Python*. https://www.geeksforgeeks.org/a-search-algorithm-in-python/

Paulino, Laura & Hannum, Correy & Varde, Aparna & Conti, Christopher. (2021). Search Methods in Motion Planning for Mobile Robots.

Umar, M. (2023, August 19). *Exploring the A\* search algorithm in Python: Pros, cons, applications, use cases, time, and space...* Medium. https://artificial-robot.medium.com/exploring-the-a-search-algorithm-in-python-pros-cons-applications-use-cases-time-and-space-44112c856134

Khim, J., Jennison, K., & Moore, K. (n.d.). *Depth-first search (DFS): Brilliant math & science wiki*. Brilliant. https://brilliant.org/wiki/depth-first-search-DFS/