



# AIHW1 PROJECTS

---

Adam Bouafia – AIConDa UNIVAQ

# TABLE OF CONTENTS

- I. Donuts Finder (Agent Game)
  1. Project Description
  2. Project Structure (**Directory Tree**)
  3. Project Implementation Details
  4. Heuristic & Search Algorithms Implementation
  5. Performance Analysis
  6. Conclusion with Graphs and Stats

# 1. PROJECT DESCRIPTION:

The Donuts Finder game, created using Python and Pygame, is intended to offer an enjoyable and engaging method to learn about well-known path finding algorithms like Dijkstra's, A\*, Greedy best first search (GBFS), Breadth First Search (BFS), and Depth First Search (DFS). It incorporates five unique mazes (Recursive Division, Prim's Algorithm, Randomized DFS, BASIC Random Maze, BASIC Random weight) and uses Homer Simpson as the agent who desires to consume the donuts, which serve as the target nodes.

## 2. PROJECT STRUCTURE (DIRECTORY TREE)

```
import TreeGenerator.py"
Donuts/
├── assets/
│   ├── audio/
│   │   └── simpsong.mp3
│   ├── fonts/
│   │   ├── Montserrat-Bold.ttf
│   │   └── Montserrat-Regular.ttf
│   └── images/
│       ├── donuts.png
│       ├── evil.png
│       └── homer.png
├── requirements.txt
├── run.pyw
└── src/
    ├── animations.py
    ├── constants.py
    ├── generate.py
    ├── main.py
    ├── maze.py
    ├── pathfinder/
    │   ├── main.py
    │   └── models/
    │       ├── frontier.py
    │       ├── grid.py
    │       ├── node.py
    │       ├── search_types.py
    │       └── solution.py
    ├── search/
    │   ├── astar.py
    │   ├── bfs.py
    │   ├── dfs.py
    │   ├── dijkstras.py
    │   └── gbfs.py
    ├── state.py
    └── widgets.py
```

## 3. PROJECT IMPLEMENTATION DETAILS

let's break down the project implementation details based on the structure:

1. **assets directory:** This directory contains all the static files needed for the game.
  - **audio:** Contains the audio file (simpson.mp3) used in the game.
  - **fonts:** Contains the font files (Montserrat-Bold.ttf, Montserrat-Regular.ttf) used in the game.
  - **images:** Contains the images (donuts.png, evil.png, homer.png) used in the game.
2. **src directory:** This is the main source code directory. It contains the following Python files and subdirectories:
  - **animations.py:** This file contains the code for game animations.
  - **constants.py:** This file defines the constants used across the game.
  - **generate.py:** This file contains the code responsible for generating game elements.
  - **main.py:** This is the main entry point of the game. It likely contains the main game loop and controls the game flow.
  - **maze.py:** This file contains the code that generates and manages the game maze.
  - **state.py:** This file manages the game states.
  - **widgets.py:** This file contains the code for creating and managing game widgets (buttons, bars, etc.)

3. **pathfinder directory:** This directory contains the implementation of the agent's pathfinding logic.
  - **main.py:** This is the main entry point for the pathfinding logic.
  - **models directory:** Contains the classes that represent the conceptual models of the game.
    - **frontier.py:** This file manages the frontier of the search space in the pathfinding algorithm.
    - **grid.py:** This file represents the game grid.
    - **node.py:** This file represents a node in the search space.
    - **search\_types.py:** This file defines the types of search algorithms used.
    - **solution.py:** This file represents a solution to the pathfinding problem.
  - **search directory:** Contains the implementation of the different search algorithms used.
    - **astar.py:** Contains the implementation of the A\* search algorithm.
    - **bfs.py:** Contains the implementation of the Breadth-First Search algorithm.
    - **dfs.py:** Contains the implementation of the Depth-First Search algorithm.
    - **dijkstras.py:** Contains the implementation of Dijkstra's algorithm.
    - **gbfs.py:** Contains the implementation of the Greedy Best-First Search algorithm.
4. **requirements.txt:** This file lists all the Python libraries that your project depends on. It is used to ensure that all the necessary dependencies can be installed.
5. **run.pyw:** This is the script that starts the game. It calls a function from main.py in the src directory.

## 4. Heuristic & Search Algorithms Implementation

1. *A Search (astar.py)\*\*:* The A Search algorithm uses both the heuristic function and the actual cost from the start node to the current node to find the shortest path. In your code, the `search` method is responsible for implementing this algorithm. It maintains a priority queue `frontier` where nodes are prioritized based on the sum of their actual cost from the start node (`g_score`) and the heuristic cost from the current node to the goal (`f_score`). The heuristic function `heuristic` is used to estimate the cost from the current state to the goal state, which in this case is calculated as the Manhattan distance.

```
frontier.add(node, priority=AStarSearch.heuristic(grid.start, grid.end))  
...  
f_score[state] = cost + AStarSearch.heuristic(state, grid.end)
```





2. Breadth-First Search (bfs.py): In your code, the BFS algorithm uses a Queue `frontier` to explore all the neighboring nodes at the current depth before moving on to nodes at the next depth level. It ensures that all the closest nodes are explored before moving to the next level nodes. This is evident from the code snippet where new nodes are added to the frontier:

```
frontier.add(node=new)
```



3. Depth-First Search (dfs.py): The DFS algorithm in your code uses a Stack `frontier` to explore as far as possible along each branch before backtracking. This is evident from the code snippet where new nodes are added to the frontier:

```
frontier.add(node=new)
```





4. Dijkstra's Algorithm (dijkstras.py): Dijkstra's algorithm finds the shortest path in a graph with non-negative edge weights. It uses a priority queue **frontier** where nodes are prioritized based on their cost from the start node. The algorithm continues until the target node has been processed. This is evident from the code where new nodes are added to the frontier with their cost as the priority:

```
frontier.add(node=n, priority=cost)
```



5. Greedy Best-First Search (gbfs.py): This algorithm uses a heuristic function to estimate the cost from the current node to the target node and expands the node that appears to be closest to the target. Unlike A\*, it doesn't consider the cost to reach the current node from the start node. This is evident from the code where new nodes are added to the frontier with their heuristic cost as the priority:

```
frontier.add(  
    node=n,  
    priority=GreedyBestFirstSearch.heuristic(  
        state, grid.end),  
)
```



Each of these algorithms is implemented as a static method in their respective classes, and they all follow a similar structure. They create a node for the source cell, instantiate a frontier, and then enter a loop where they continuously explore nodes until they find a solution or exhaust all possible paths. Upon finding a solution, they generate a path and return a Solution

### 3. PERFORMANCE ANALYSIS

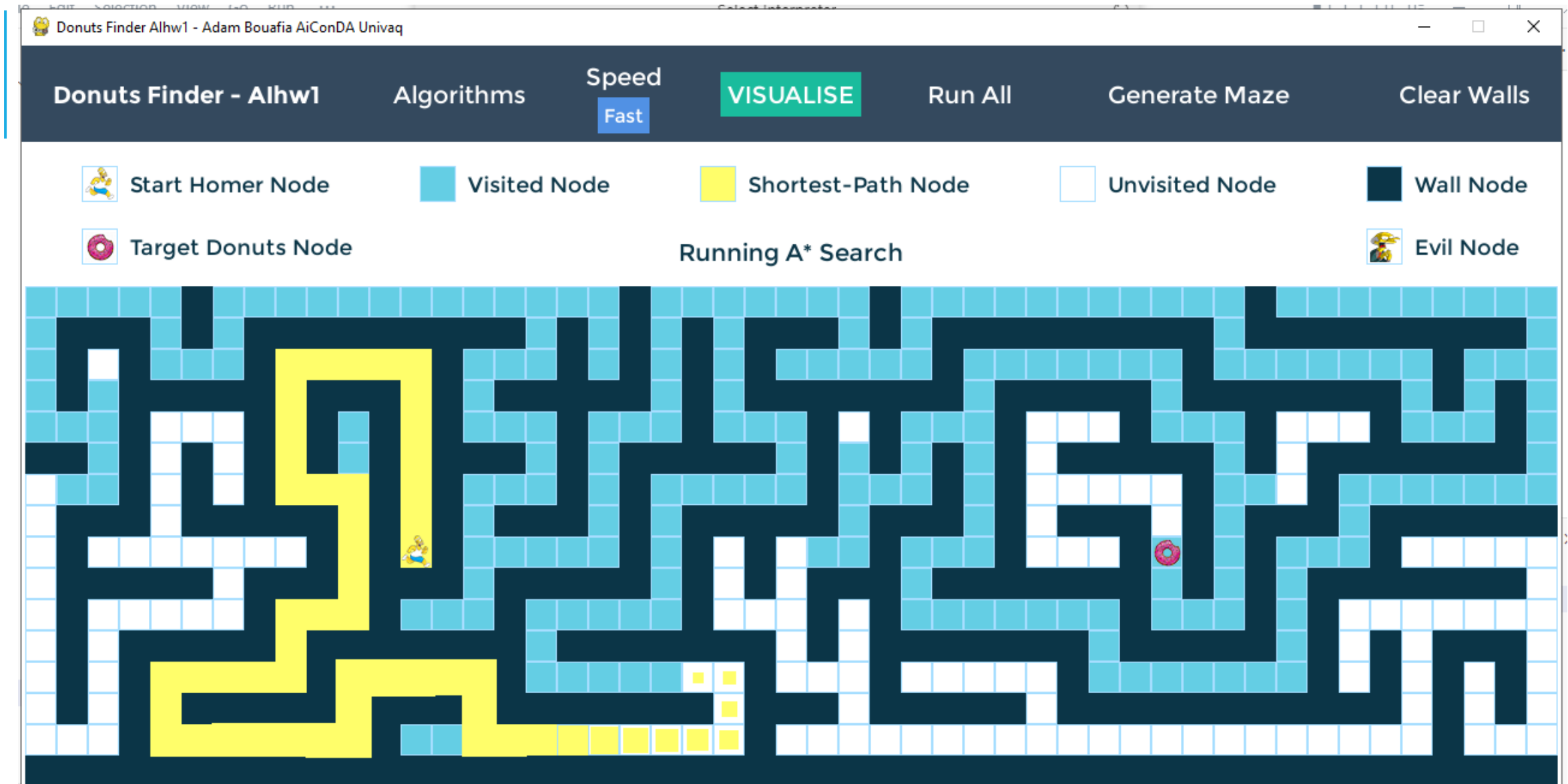
When evaluating the performance of these search algorithms, there are a few key factors to consider: the speed of finding a solution, the quality of the solution (i.e., is it the shortest path?), and the computational resources required.

1. *A Search\**: A\* Search is generally quite efficient and is guaranteed to find the shortest path if one exists. It uses a heuristic to guide its search, which can greatly speed up the process by prioritizing nodes that are estimated to be closer to the goal. However, its performance can degrade in complex environments with many obstacles, as it needs to maintain and update information about all visited nodes.
2. **Breadth-First Search (BFS)**: BFS is an optimal algorithm that always finds the shortest path, but it can be slow because it explores all neighboring nodes at the current depth before moving on to nodes at the next depth level. BFS could be inefficient in scenarios where the goal node is far from the start node or in larger state spaces as it keeps every generated node in memory.

3. **Depth-First Search (DFS):** DFS is not optimal and can be slow in the worst case, as it explores as far as possible along each branch before backtracking. It might end up exploring much more of the state space than necessary. However, DFS uses less memory than BFS because it needs to store only a single path from the root to a leaf node, along with the remaining unexplored adjacent nodes for each node on the path.
4. **Dijkstra's Algorithm:** Dijkstra's algorithm is very efficient for computing shortest paths in a graph with non-negative edge weights. It provides a very accurate and optimal solution. However, it can be slow on graphs with a large number of nodes and edges because it examines all edges to find the shortest path.
5. **Greedy Best-First Search (GBFS):** GBFS is typically faster than A\*, BFS, and Dijkstra's algorithm but it does not guarantee the shortest path. It uses a heuristic to estimate the cost to the goal and prioritizes nodes based on this estimated cost. However, this can lead GBFS to get stuck in loops or choose longer paths.

Given these characteristics, if the goal is to find the shortest path, A\* Search and Dijkstra's algorithm would be the best options. If memory is a concern and the shortest path is not required, DFS could be a good option. If speed is a priority and finding the exact shortest path is not necessary, GBFS might be the best choice.

# RECURSIVE DIVISION MAZE



Donuts Finder - Alhw1

Algorithms

Speed  
Fast

VISUALISE

Run All

Generate Maze

Clear Walls



Start Homer Node



Visited Node



Shortest-Path Node



Unvisited Node



Wall Node



Target Donut



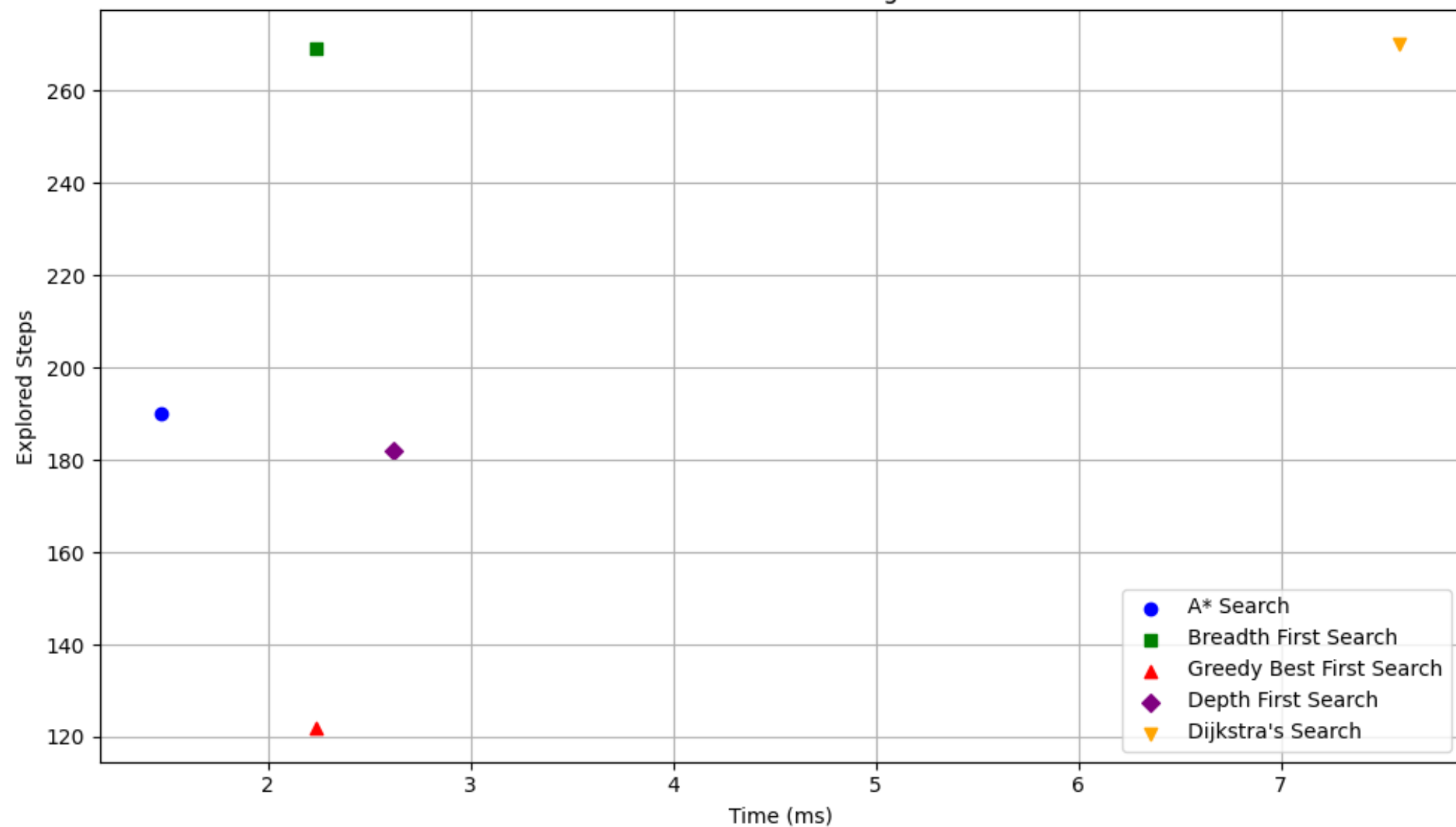
Evil Node

## COMPARISON RESULTS


Algorithm	Steps Explored	Path Length	Path Cost	Time Taken
1. A* Search	276	181	180	3.03ms
2. Dijkstra's Search	338	181	180	2.66ms
3. Greedy Best First Search	186	181	180	1.11ms
4. Breadth First Search	338	181	180	1.08ms
5. Depth First Search	265	181	180	0.85ms



Detailed Results for All Algorithms



# Randomised DFS MAZE

 Donuts Finder Alhw1 - Adam Bouafia AiConDA Univaq

Donuts Finder - Alhw1

Algorithms


Speed  
Fast


VISUALISE


Run All


Generate Maze


Clear Walls


 Start Homer Node


 Visited Node

 Shortest-Path Node

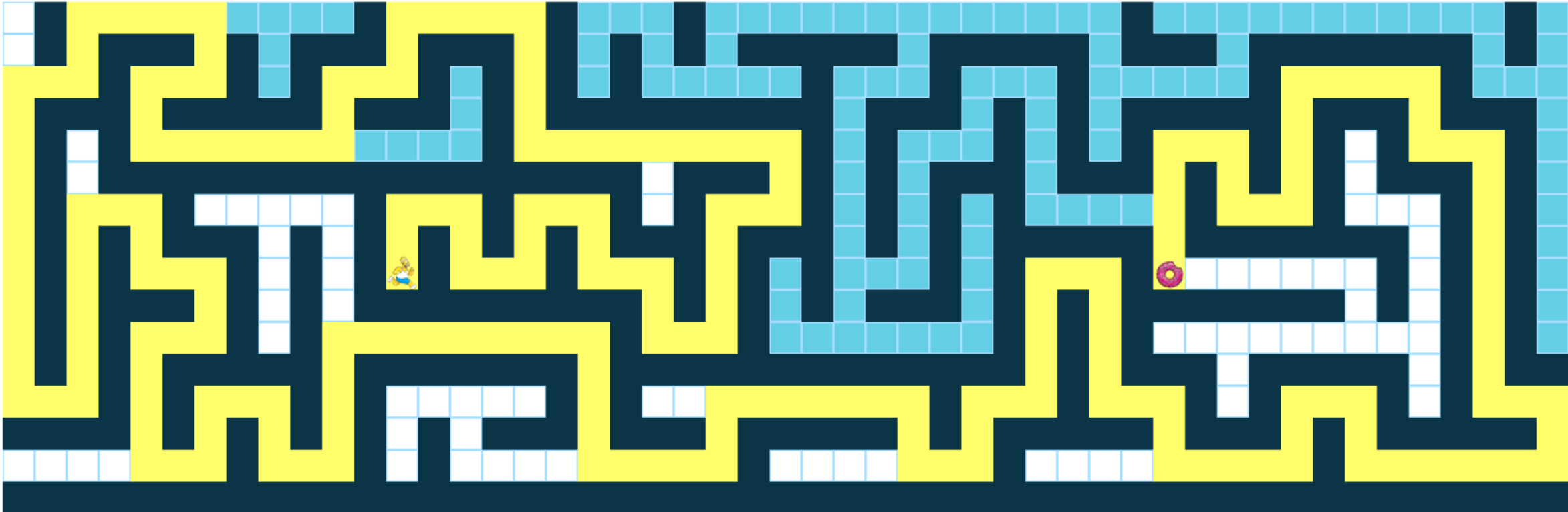
 Unvisited Node

 Wall Node

 Target Donuts Node

 Evil Node

Depth First Search



Donuts Finder - Alhw1

Algorithms

Speed

Fast

VISUALISE

Run All

Generate Maze

Clear Walls



Start Homer Node



Visited Node



Shortest-Path Node



Unvisited Node



Wall Node



Target Donut

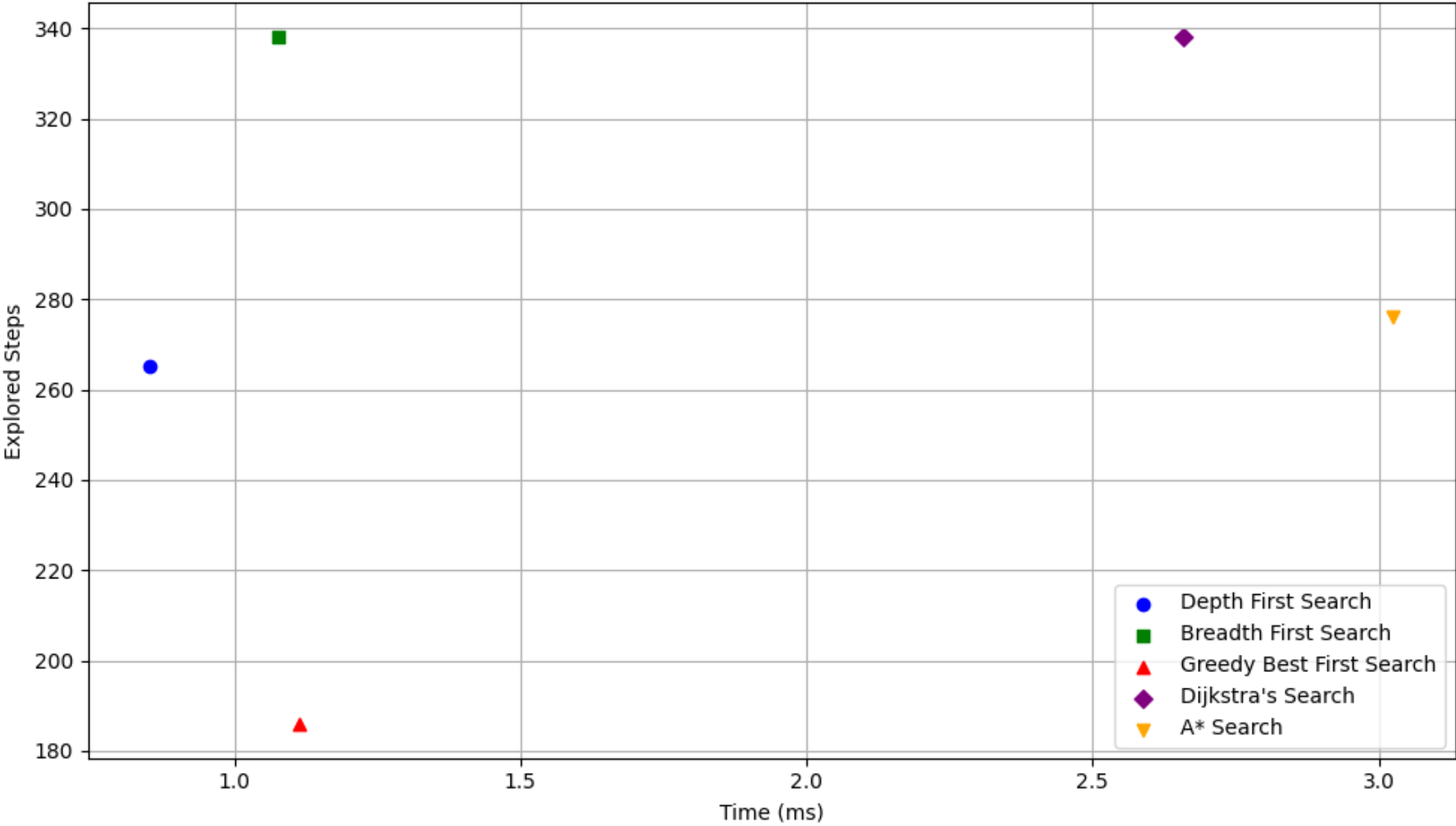


Evil Node

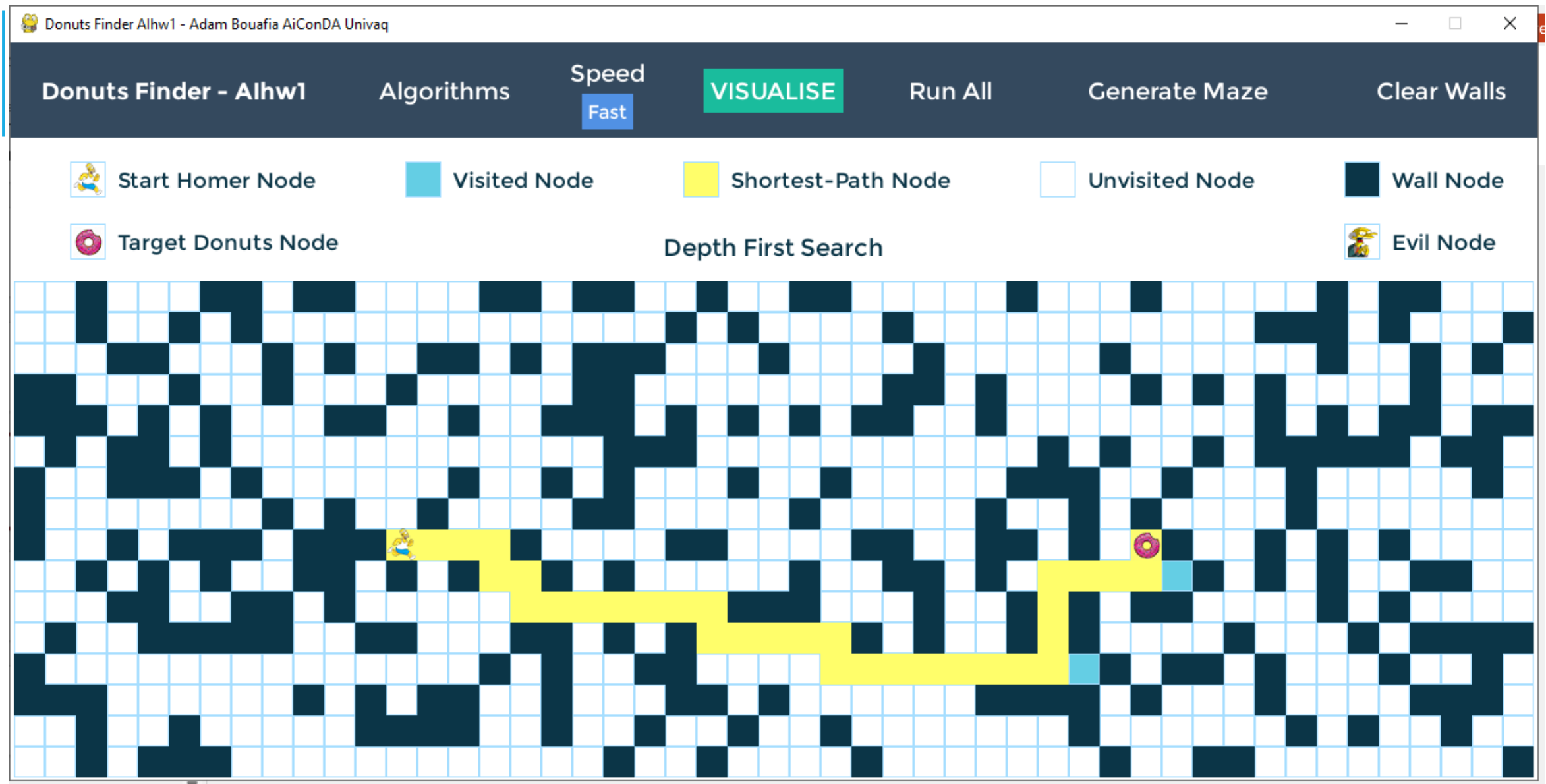
### COMPARISON RESULTS

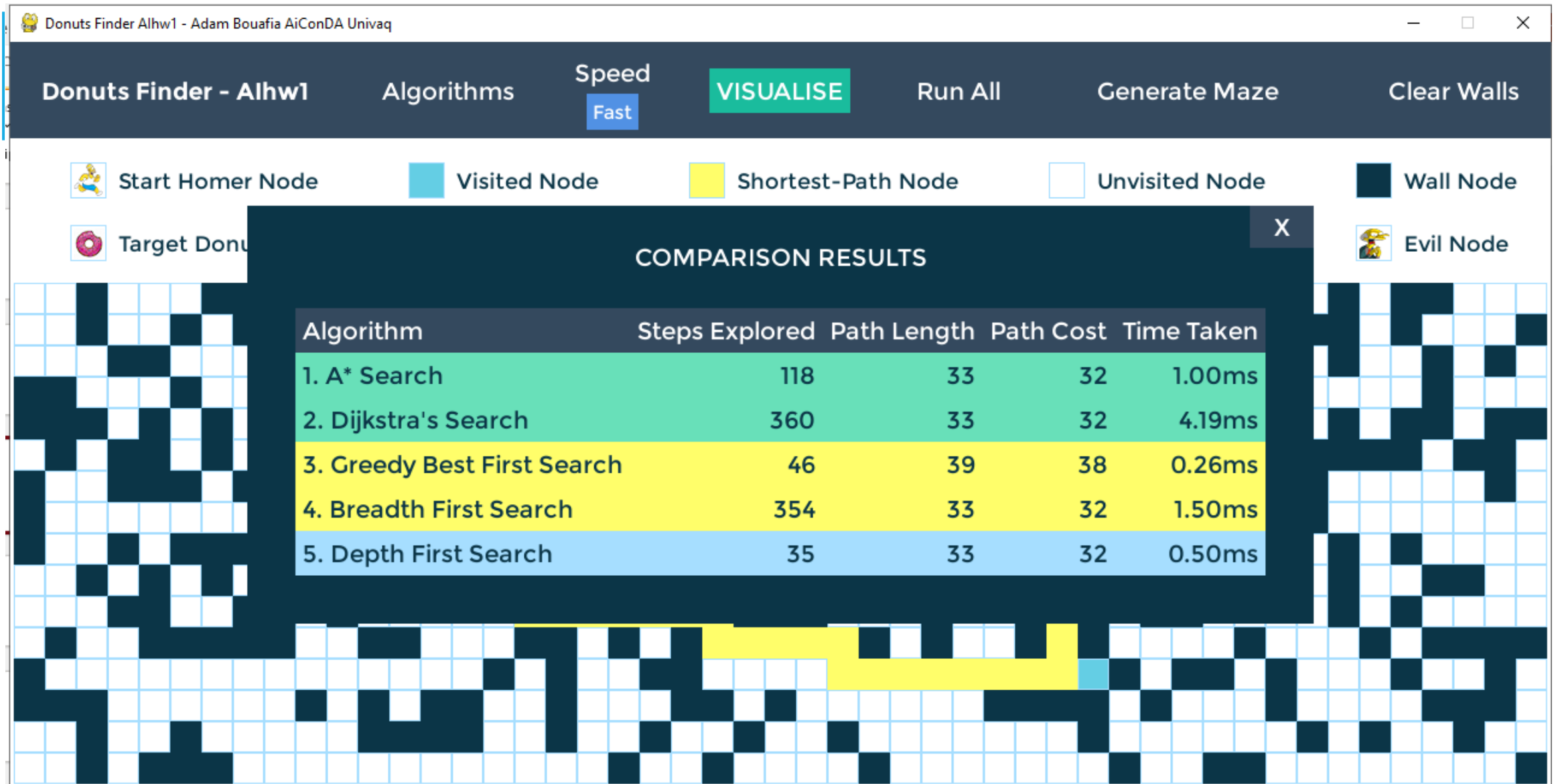
Algorithm	Steps Explored	Path Length	Path Cost	Time Taken
1. A* Search	269	213	212	5.14ms
2. Dijkstra's Search	271	213	212	3.82ms
3. Greedy Best First Search	239	213	212	3.47ms
4. Breadth First Search	270	213	212	0.83ms
5. Depth First Search	327	213	212	1.14ms

Detailed Results for All Algorithms



# BASIC RANDOM MAZE





Detailed Results for All Algorithms

