Artificial Intelligence Fundamentals

Lab 1: *Implementation of search algorithms*

Máster in Artificial Intelligence 2024-25

Author: Paula Biderman Mato

Date: 23 June 2025

Index

1. Methods

1.1 Formal Characterization of the Problem

1.1.1 Representation of the State

Each state is represented by the robot’s position on the grid and its orientation. The position is given by coordinates (x,y) and the orientation can take one of eight directions: *North, NE, East, SE, South, SW, West, or NW* — typically encoded as integers from 0 to 7, with an optional value 8 representing "idle" (no specific direction). Thus, a full state can be described as: $State = \big((x, y), orientation\big)$.

For instance, in the *exampleMap.txt* file, the initial state is at position (0,3) facing North (code 0), and the goal is at (1,2) with no required orientation (code 8).

1.1.2 Specification of the available set of Operators (Actions of Robot)

The robot can perform three types of actions in any state:

- *Turn Left:* Rotate 45° counter-clockwise to the next orientation. This changes the robot’s orientation to the left neighbor in the orientation list (*e.g. from North to North-West)* without changing its position.
- *Turn Right:* Rotate 45° clockwise to the next orientation. This changes the orientation to the right neighbor (e.*g. North to North-East)* with no position change.
- *Advance:* Move forward one cell in the direction of the robot’s current orientation. This action changes the robot’s position in the grid by adding the orientation’s movement vector (e.*g. if oriented North-West, the position changes by x=-1, y=-1).* The orientation remains the same during an advance.

The robot can move in all eight compass directions (including diagonals). There is no 'idle' move action; the code 8 for orientation represents an idle goal state, not an actual movement.

1.1.3 Transition Model (Preconditions and Results of each action)

Applying an action leads to a deterministic transition to a new state as follows:

- *Turn Left/Right:* Results in a new state `(x, y, orientation)` at the same position. The new orientation is the current orientation rotated 45° left or right. For example, from orientation W (west), turning right yields NW, while turning left yields SW. These actions are always applicable (no precondition, since rotation in place is always possible).

- *Advance*: Results in a new state `(x', y', orientation)` where `(x',y')` is the adjacent cell in the direction of the current orientation. For example, if the robot is at (x,y) facing NE, an advance yields `(x+(-1), y+1)=(x-1, y+1).` The precondition for advancing is that the resulting cell `(x',y')` lies within the grid bounds (and is not an impassable obstacle). In the given implementation, the grid is composed of numeric cost values for each cell and there are no impassable cells (all moves within bounds are allowed). If the move would go outside the map boundaries, that action is simply not generated as a possible successor.

Each action has an associated cost (defined below), which contributes to the path cost. The transition model thus defines a state-space graph where nodes are `(x,y,orientation)` and edges correspond to these actions leading to neighboring states.

1.1.4 Goal Test

A state is considered a goal state if the robot's position matches the goal coordinates. If a specific orientation is required (e.g., North, South), the orientation must also match. However, in most scenarios of this project, the goal orientation is set to "I" (Idle), meaning it is irrelevant. In such cases, reaching the goal position is sufficient to satisfy the goal test, regardless of orientation.

1.1.5 Cost Function

Each action incurs a step cost, and the total path cost `g(n)` is the sum of costs of all actions taken from the start up to the current state. The cost function is defined as follows:

- *Turn Left or Turn Right*: cost = 1. Rotating in place by 45° has a uniform unit cost.
- *Advance* (move forward one cell): cost = the terrain cost value of the destination cell. Each cell in the grid carries an integer cost value (e.g. in the example map, cells have costs ranging from 1 to 4). When the robot moves into a new cell, it pays the cost associated with that cell. All cost values are positive, and in the generated maps they range from 1 to 10 (since random terrain generation uses 1–10 costs).

The total path cost is the sum of 1 unit for each turn and the terrain cost for each forward move. As a result, paths through higher-cost cells are more expensive, and algorithms aim to minimize the total accumulated cost **g**. Since all actions have non-negative costs, optimality is preserved for applicable search algorithms.

1.2 Analysis of Blind Search Methods

1.2.1 Analysis of Breadth-First Search (BFS)

Breadth-First Search is a blind search algorithm that explores the state space by expanding the shallowest nodes first. It uses a FIFO queue to manage the frontier, ensuring that nodes at lower depths (shorter action sequences) are expanded before deeper ones.

Completeness: BFS is complete in finite state spaces, such as our grid with a fixed number of cells and orientations. It guarantees finding a solution if one exists.

Optimality: BFS finds the path with the fewest actions, but not necessarily the lowest cost, since it treats all actions equally. For instance, it might return a 5-step path with cost 15, even if a 6-step path with cost 12 exists.

Time and Space Complexity: From a computational perspective, it is worth noting that both BFS and DFS have a worst-case time complexity of $O(b^d)$, where b is the branching factor (in our case, up to 3 possible actions per state: move forward, turn left, or turn right), and d is the depth of the shallowest solution. However, their space complexity differs: BFS requires $O(b^d)$ memory to store all nodes in the frontier at each depth level, which can grow rapidly and become impractical for large maps. In contrast, DFS uses only $O(bd)$ space, since it stores only the current path being explored. This explains the empirical results shown in *Section 2.2*, where BFS exhibits the largest frontier size among the evaluated algorithms.

Implementation: In our code, BFS uses a deque for the frontier and a visited set to avoid revisiting states. States are marked as explored when dequeued. The algorithm stops once the goal is reached, returning the path, total cost, depth, and metrics such as explored nodes and frontier size. While BFS finds the shortest path in terms of steps, it does not account for different action costs, which may lead to suboptimal total cost.

Note: Depth-First Search (DFS) is also implemented as a blind search method. It explores one path as deep as possible before backtracking. DFS is less memory-intensive than BFS but does not guarantee optimality and may produce longer paths.

1.3 Heuristics for A* Algorithm

1.3.1 Heuristic Function Definition (mathematical form, rationale, monotonicity)

In informed search, a heuristic function h(n) estimates the cost of reaching the goal from a given node n. For the A* search algorithm the chosen heuristic is the Euclidean distance to the goal, which assumes the robot could travel directly toward the target in a straight line. This is appropriate in a grid that allows diagonal moves, as it reflects the shortest possible path ignoring obstacles or movement constraints. Thus it is computed as the straight-line distance between the current position (x, y) and the goal (x_g, y_g). It is calculated using the standard formula: $h(n) = \sqrt{(x - x_g)^2 + (y - y_g)^2}$. For example, if the robot is at (0,3) and goal is at (1,2), $h(n) = \sqrt{(0-1)^2 + (3-2)^2}$ *= 1.414 aprox.*

Although it underestimates the real path cost — since it does not account for terrain variations or turns — it never overestimates it, making it an admissible heuristic. For instance, even when the goal is nearby, the real path may require turns or detours, increasing the true cost beyond the straight-line estimate.

Additionally, the heuristic satisfies the monotonicity property (also known as consistency): any move from a node to its neighbor reduces the heuristic value by no more than the cost of that move. This guarantees that the estimated cost along a path never decreases, allowing A* to avoid revisiting nodes and ensuring that the first time a node is expanded, it has already been reached optimally. This property is important not only to guarantee optimality but also to improve efficiency, as it allows A* to behave like Dijkstra's algorithm with added guidance from the heuristic — reducing redundant work and avoiding backtracking.

In summary, the Euclidean distance is a suitable and effective heuristic for this problem: it is admissible, consistent, aligned with the robot's movement rules, and helps A* find optimal paths efficiently.

2 Results

2.1 Execution Trace Example

To illustrate the behavior of each algorithm, we present an execution trace on a specific example map. We use the map defined in exampleMap.txt, which is a 3x4 grid with the start at (0,3) facing North and the goal at (1,2) (orientation not required). The terrain costs in this map are as follows:

```
3 4     (grid dimensions: 3 rows × 4 columns)
3 2 4 1 (row 0 costs)
2 3 1 2 (row 1 costs)
1 4 2 3 (row 2 costs)
0 3 0   (start: position (0,3), orientation 0 = North)
1 2 8   (goal: position (1,2), orientation 8 = "I")
```

Note: *See additional execution details in the Appendix (section 5.2).*

Below, it is shown the sequence of actions and states for a solution found by each algorithm. Each 'Node `i`' entry shows the depth d (number of actions from start), cumulative cost g(n) up to that state, the action (op) that led to that state, the state's coordinates and orientation S=`(x, y)`, Orientation, and if applicable the heuristic value `h(n)` (for A*). Explored nodes refers to the total number of nodes expanded (visited) during the search, and frontier refers to the number of nodes left in the frontier at the end (when the solution was found).

2.1.1 Breadth-First Search (BFS)

Using BFS on this map, one optimal sequence in terms of steps is found (BFS will find the shortest path in number of moves). The trace of the solution path is:

- `operator(Start)`
- `Node 0: (d=0, g(n)=0, op=Start, S=(0, 3), Orientation=N)`
- `operator(Turn left)`
- `Node 1: (d=1, g(n)=1, op=Turn left, S=(0, 3), Orientation=NW)`
- `operator(Turn left)`
- `Node 2: (d=2, g(n)=2, op=Turn left, S=(0, 3), Orientation=W)`
- `operator(Turn left)`
- `Node 3: (d=3, g(n)=3, op=Turn left, S=(0, 3), Orientation=SW)`
- `operator(Advance)`
- `Node 4: (d=4, g(n)=4, op=Advance, S=(1, 2), Orientation=SW)`

Note: *See additional execution details in Appendix (section 5.3)*

Summary: BFS reached the goal in 4 actions (depth d=4) with total cost g=4. Total number of nodes explored: 10 and Total number of nodes in frontier: 13.

BFS explores a broad set of nodes, but it finds this solution which involves three left turns to face South-West, then an advance into the goal. The path cost of 4 comes from three turns = 3 and entering the goal cell of cost 1.

2.1.2 Depth-First Search (DFS)

Using DFS on the same map, a valid solution is found, though it is not optimal. DFS dives deep into one branch before exploring alternatives, so it often finds a long path to the goal. One such DFS solution path is:

- *`operator(Start)`*
- *`Node 0: (d=0, g(n)=0, op=Start, S=(0, 3), Orientation=N)`*
- *`operator(Turn Left)`*
- *`Node 1: (d=1, g(n)=1, op=Turn Left, S=(0, 3), Orientation=NW)`*
- *`operator(Turn Left)`*
- *`Node 2: (d=2, g(n)=2, op=Turn Left, S=(0, 3), Orientation=W)`*
- *`operator(Advance)`*
- *`Node 3: (d=3, g(n)=6, op=Advance, S=(0, 2), Orientation=W)`*
- *`operator(Advance)`*
- *`Node 4: (d=4, g(n)=8, op=Advance, S=(0, 1), Orientation=W)`*
- *`operator(Advance)`*
- *`Node 5: (d=5, g(n)=11, op=Advance, S=(0, 0), Orientation=W)`*
- *`operator(Turn Left)`*
- *`Node 6: (d=6, g(n)=12, op=Turn Left, S=(0, 0), Orientation=SW)`*
- *`operator(Turn Left)`*
- *`Node 7: (d=7, g(n)=13, op=Turn Left, S=(0, 0), Orientation=S)`*
- *`operator(Advance)`*
- *`Node 8: (d=8, g(n)=15, op=Advance, S=(1, 0), Orientation=S)`*
- *`operator(Advance)`*
- *`Node 9: (d=9, g(n)=16, op=Advance, S=(2, 0), Orientation=S)`*
- *`operator(Turn Left)`*
- *`Node 10: (d=10, g(n)=17, op=Turn Left, S=(2, 0), Orientation=SE)`*

- *operator(Turn left)*
- *Node 11: (d=11, g(n)=18, op=Turn left, S=(2, 0), Orientation=E)*
- *operator(Advance)*
- *Node 12: (d=12, g(n)=22, op=Advance, S=(2, 1), Orientation=E)*
- *operator(Advance)*
- *Node 13: (d=13, g(n)=24, op=Advance, S=(2, 2), Orientation=E)*
- *operator(Advance)*
- *Node 14: (d=14, g(n)=27, op=Advance, S=(2, 3), Orientation=E)*
- *operator(Turn left)*
- *Node 15: (d=15, g(n)=28, op=Turn left, S=(2, 3), Orientation=NE)*
- *operator(Turn left)*
- *Node 16: (d=16, g(n)=29, op=Turn left, S=(2, 3), Orientation=N)*
- *operator(Advance)*
- *Node 17: (d=17, g(n)=31, op=Advance, S=(1, 3), Orientation=N)*
- *operator(Turn left)*
- *Node 18: (d=18, g(n)=32, op=Turn left, S=(1, 3), Orientation=NW)*
- *operator(Turn left)*
- *Node 19: (d=19, g(n)=33, op=Turn left, S=(1, 3), Orientation=W)*
- *operator(Advance)*
- *Node 20: (d=20, g(n)=34, op=Advance, S=(1, 2), Orientation=W)*

***Note:** See additional execution details in the Appendix (section 5.3)*

Summary: DFS reached the goal in 20 actions (depth d=20) with total cost g=34. Total number of nodes explored: 21 and Total number of nodes in frontier: 40.

This DFS path is much longer – it essentially went in a loop around the grid (all the way to the top-left corner, then down and across to the goal). DFS did not find the shorter 4-step path because it pursued one branch exhaustively before others. The high frontier count (40) indicates many alternative nodes were still stacked to explore later, but the search stopped once a goal was found.

2.1.3 A* search (heuristic = Euclidean Distance)

Using the A* algorithm with the Euclidean distance heuristic on the same example, it finds an optimal cost path. In this case, the heuristic guides it to effectively the same short path that BFS found (since that path also happened to be optimal in cost). The trace of the A* solution is:

- operator(Start)
- Node 0: (d=0, g(n)=0, op=Start, S=(0, 3), Orientation=N, h(n)=1.414)
- operator(Turn left)
- Node 1: (d=1, g(n)=1, op=Turn left, S=(0, 3), Orientation=NW, h(n)=1.414)
- operator(Turn left)
- Node 2: (d=2, g(n)=2, op=Turn left, S=(0, 3), Orientation=W, h(n)=1.414)
- operator(Turn left)
- Node 3: (d=3, g(n)=3, op=Turn left, S=(0, 3), Orientation=SW, h(n)=1.414)
- operator(Advance)
- Node 4: (d=4, g(n)=4, op=Advance, S=(1, 2), Orientation=SW, h(n)=0.0)

***Note:** See additional execution details in the Appendix x (section 5.3)*

Summary: A* reached the goal in 4 actions (depth d=4) with total cost g=4 (which is the optimal cost path). Total number of nodes explored: 8 and Total number of nodes in frontier: 9.

A* expands fewer nodes than BFS (8 vs 10) thanks to the heuristic guiding the search toward the goal. Although the solution path matches that of BFS, A* is more efficient. The heuristic values decrease consistently as the robot approaches the goal, reaching h = 0 at the end. This reflects the heuristic's consistency, where h(n) + cost never increases across steps.

2.2 Comparison of the performance of the different implemented methods

The performance of BFS, DFS, and A* (with Euclidean heuristic) was evaluated on randomly generated 3×3, 5×5, 7×7, and 9×9 maps. For each size, 10 maps were generated with fixed start (top-left) and goal (bottom-right) positions. Each algorithm was run once per map, and the following metrics were recorded: solution depth (d), path cost (g), number of nodes expanded (#E), and final frontier size (#F). Tables 1–4 report the average values per algorithm and map size.

Table 1: Comparative performance (average results over 10 runs) for map size 3×3.

Algorithm	d (steps)	g (cost)	#E (explored)	#F (frontier)
Breadth-first	5.0	10.1	29.0	12.0
Depth-first	10.0	19.9	19.0	8.0
A* (Euclidean)	5.2	9.9	32.5	10.3

Table 2: Comparative performance for map size 5×5

Algorithm	d (steps)	g (cost)	#E (explored)	#F (frontier)
Breadth-first	7.0	15.6	102.0	40.0
Depth-first	14.0	33.8	31.0	16.0
A* (Euclidean)	7.4	15.5	102.9	27.5

Table 3: Comparative performance for map size 7×7

Algorithm	d (steps)	g (cost)	#E (explored)	#F (frontier)
Breadth-first	9.0	23.7	238.0	72.0
Depth-first	18.0	46.2	43.0	24.0
A* (Euclidean)	11.2	22.0	239.7	68.0

Table 4: Comparative performance for map size 9×9

Algorithm	d (steps)	g (cost)	#E (explored)	#F (frontier)
Breadth-first	11.0	32.9	438.0	104.0
Depth-first	22.0	62.3	55.0	32.0
A* (Euclidean)	15.0	29.5	459.6	95.3

Across all map sizes, BFS and A* consistently produced shorter and lower-cost paths than DFS, with A* achieving the best average cost overall. DFS expanded far fewer nodes due to early solutions, but its paths were significantly longer and costlier. In larger maps, A* and BFS explored similar portions of the state space, with A* showing only a modest advantage in frontier size. These trends will be further analyzed in the next section.

Note: *See additional execution details in Appendix (section 5.4)*

3. Discussion

3.1 Advantages and Disadvantages of Each Search Method (based on Section 2.2 results)

Based on the results in section 2.2, I analyzed the strengths, weaknesses, and potential risks of each search method—DFS, BFS, and A*—when applied to randomly generated maps of increasing size.

Depth-First Search explores as deep as possible along one path before backtracking. Its main advantages are low memory usage—as it only stores a single path and a few unexplored siblings—and fast discovery of some solution, especially in small or sparse spaces. In my experiments, DFS had the smallest average frontier size (e.g., 32 in 9×9 maps) and expanded far fewer nodes than BFS or A*. However, the disadvantages are significant: DFS often produces very suboptimal solutions, with long paths and high total costs (e.g., 22 steps and cost 62.3 in 9×9 vs. BFS's 11 steps and 32.9 cost). A critical risk is that DFS may get trapped in deep paths, especially in large spaces. Although cycle checking ensured completeness in my implementation, the algorithm still ignores cost and path quality, leading to inefficient solutions.

Breadth-First Search explores nodes uniformly by depth and guarantees to find the shortest path in terms of number of steps. This makes it complete and optimal under uniform step costs. In my results, BFS consistently achieved the minimal step count, such as ~11 steps in 9×9 maps. However, its disadvantages include high memory and processing requirements. The frontier size grows exponentially with depth, reaching over 100 nodes in larger maps, and the number of expanded nodes is also high. BFS also does not consider path cost, meaning it can return suboptimal-cost paths when step costs vary. A risk is that BFS becomes computationally infeasible in very large or complex spaces due to its memory consumption and exhaustive exploration.

*A** balances DFS and BFS by combining path cost with a heuristic. Using an admissible heuristic, it guarantees optimal-cost paths and often requires fewer node expansions than BFS. In my experiments, A* consistently found the lowest-cost solutions, and on smaller maps it had smaller frontiers and expanded fewer nodes than BFS. However, in larger maps with uniform terrain, the heuristic lost selectivity, so A* sometimes expanded as many or more nodes than BFS. The main advantages of A* are optimality, guided exploration, and often better efficiency than uninformed methods. Still, drawbacks include high memory usage (similar to BFS), and strong dependence on heuristic quality. A poor heuristic can make A* behave like BFS or lose optimality altogether. Therefore, a key risk is that in complex spaces with poor heuristics, A* may become inefficient or ineffective.

Thus DFS is memory-efficient and sometimes fast, but risks very poor solutions. BFS finds shortest-step paths and is reliable but highly resource-intensive. A* generally offers the best cost-optimal results with greater efficiency, but its success hinges on the heuristic and it shares BFS's memory risks.

4. Structure of the Program

4.1 Prerequisites

The implementation is developed in Python 3.x and requires the following:

- numpy, for generating random maps.
- pandas, for running batch experiments and exporting CSV results.

Notably, the core search algorithms (Breadth-First Search, Depth-First Search, and A*) use only the Python standard library, so additional libraries are only needed for experimentation, not for basic execution on predefined maps.

All program files must reside in the same working directory. These include:

```
BreadthFirstSearch.py, DepthFirstSearch.py, AStarSearch.py, Heuristicas.py,
PlanificadorBase.py, Mapa.py, Navegation.py, Nodo.py, Reportero.py, Metricas.py,
RunExperiments.py, exampleMap.txt, mainMapa.py, main_2.py.
```

Once the environment is properly configured, the program can be executed using the scripts described in the following subsection, depending on the desired goal.

4.2 Running the Program

There are three main Python scripts in this project, each supporting different execution modes depending on the desired goal:

4.2.1. Solving a Specific Map with Algorithm Execution

To observe the detailed step-by-step execution of the implemented search algorithms (BFS, DFS, and A*), the main script to use is:

```
python main_2.py
```

This will automatically load the predefined exampleMap.txt, display the map, the start and goal positions, and then prompt the user to choose one of the following algorithms:

- `bfs – Breadth-First Search`
- `dfs – Depth-First Search`
- `astar – A* Search using Euclidean Distance as the heuristic`

Once an algorithm is selected, the program will perform the search and print a complete trace that includes:

- `The nodes explored during the search`
- `The resulting path from start to goal (if found), printed in the required format`
- `The number of explored nodes (#E)`
- `The size of the frontier at the end (#F)`
- `Execution time`
- `Depth (d) and cost (g) of the solution`

For A*, the trace also includes the evaluation of h(n) (heuristic value) at each step, following the specific format required by the project instructions.

4.2.2. Running Batch Experiments and Collecting Performance Metrics

To evaluate and compare the algorithms statistically, the script to run is:

```
python RunExperiments.py
```

This script automatically generates 10 random maps for each of the following sizes: 3x3, 5x5, 7x7, and 9x9, always using the same fixed start position (0,0) and goal (N-1, N-1). It runs BFS, DFS, and A* on each generated map, collects the performance metrics for each run, and computes the averages. The maps generated and results are save.

4.2.3 Exploring or Generating Custom Maps

To manually inspect or generate a map without performing any algorithmic search, use:

```
python mainMapa.py
```

The program will prompt whether to:

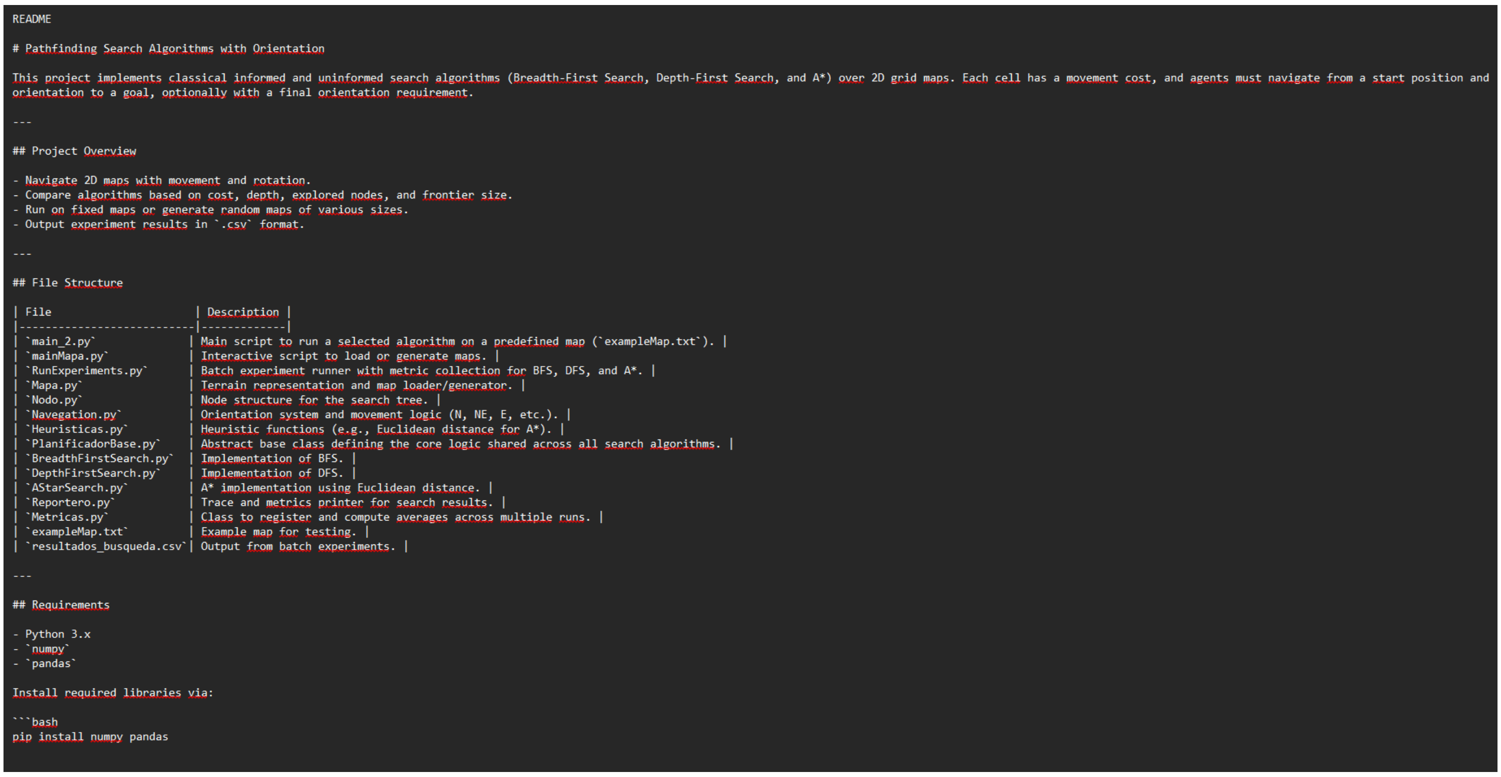
- `Load a map from file (e.g., exampleMap.txt)`
- `Generate a new random map of a specified dimension (e.g., 6x6)`

In either case, the map details (dimensions, number of obstacles, etc.) will be displayed for inspection. This utility is helpful for testing specific configurations or previewing randomly generated environments before running search algorithms on them.

5. Appendix

5.1 README.txt

In the folder there is a readme file specifying the project overview, structure and requirements.

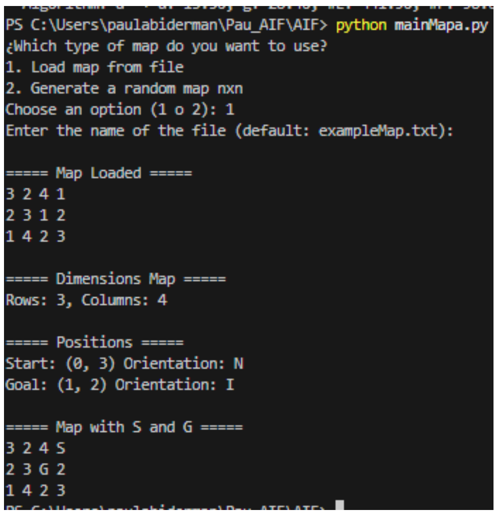


~~~
README

# Pathfinding Search Algorithms with Orientation

This project implements classical informed and uninformed search algorithms (Breadth-First Search, Depth-First Search, and A*) over 2D grid maps. Each cell has a movement cost, and agents must navigate from a start position and
orientation to a goal, optionally with a final orientation requirement.

---

## Project Overview

- Navigate 2D maps with movement and rotation.
- Compare algorithms based on cost, depth, explored nodes, and frontier size.
- Run on fixed maps or generate random maps of various sizes.
- Output experiment results in `.csv` format.

---

## File Structure

| File                       | Description |
|--------------------------|-------------|
| `main_2.py`               | Main script to run a selected algorithm on a predefined map (`exampleMap.txt`). |
| `mainMapa.py`             | Interactive script to load or generate maps. |
| `RunExperiments.py`       | Batch experiment runner with metric collection for BFS, DFS, and A*. |
| `Mapa.py`                 | Terrain representation and map loader/generator. |
| `Nodo.py`                 | Node structure for the search tree. |
| `Navegation.py`           | Orientation system and movement logic (N, NE, E, etc.). |
| `Heuristicas.py`          | Heuristic functions (e.g., Euclidean distance for A*). |
| `PlanificadorBase.py`     | Abstract base class defining the core logic shared across all search algorithms. |
| `BreadthFirstSearch.py`   | Implementation of BFS. |
| `DepthFirstSearch.py`     | Implementation of DFS. |
| `AStarSearch.py`          | A* implementation using Euclidean distance. |
| `Reportero.py`            | Trace and metrics printer for search results. |
| `Metricas.py`             | Class to register and compute averages across multiple runs. |
| `exampleMap.txt`          | Example map for testing. |
| `resultados_busqueda.csv`| Output from batch experiments. |

---

## Requirements

- Python 3.x
- `numpy`
- `pandas`

Install required libraries via:

```bash
pip install numpy pandas
~~~

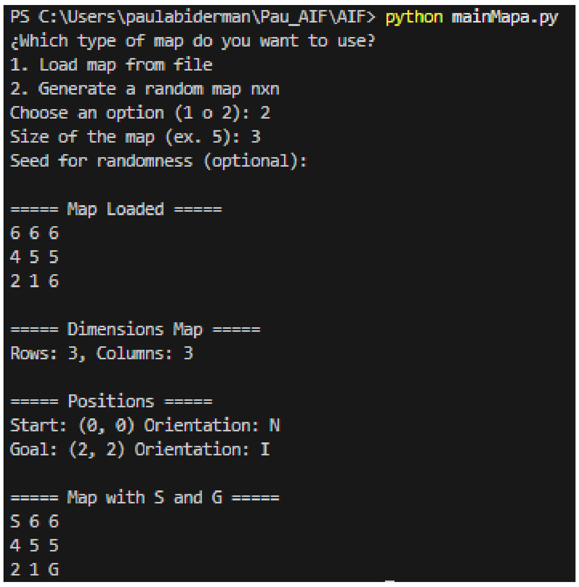
5.2 Execution of MainMapa.py

This script allows the user to interactively choose between two options: (1) loading a predefined map from a file (e.g., exampleMap.txt), or (2) generating a random square map of custom size. Once the terrain is created, the map is visualized using the mostrar_mapa_y_posiciones function. This execution is intended for verifying the correct loading or generation of terrain data and inspecting start and goal positions before running search algorithms.

Below is shown the execution of the two possibles path taken.



```
PS C:\Users\paulabiderman\Pau_AIF\AIF> python mainMapa.py
¿Which type of map do you want to use?
1. Load map from file
2. Generate a random map nxn
Choose an option (1 o 2): 1
Enter the name of the file (default: exampleMap.txt):

===== Map Loaded =====
3 2 4 1
2 3 1 2
1 4 2 3

===== Dimensions Map =====
Rows: 3, Columns: 4

===== Positions =====
Start: (0, 3) Orientation: N
Goal: (1, 2) Orientation: I

===== Map with S and G =====
3 2 4 S
2 3 G 2
1 4 2 3
PS C:\Users\paulabiderman\Pau_AIF\AIF>
```



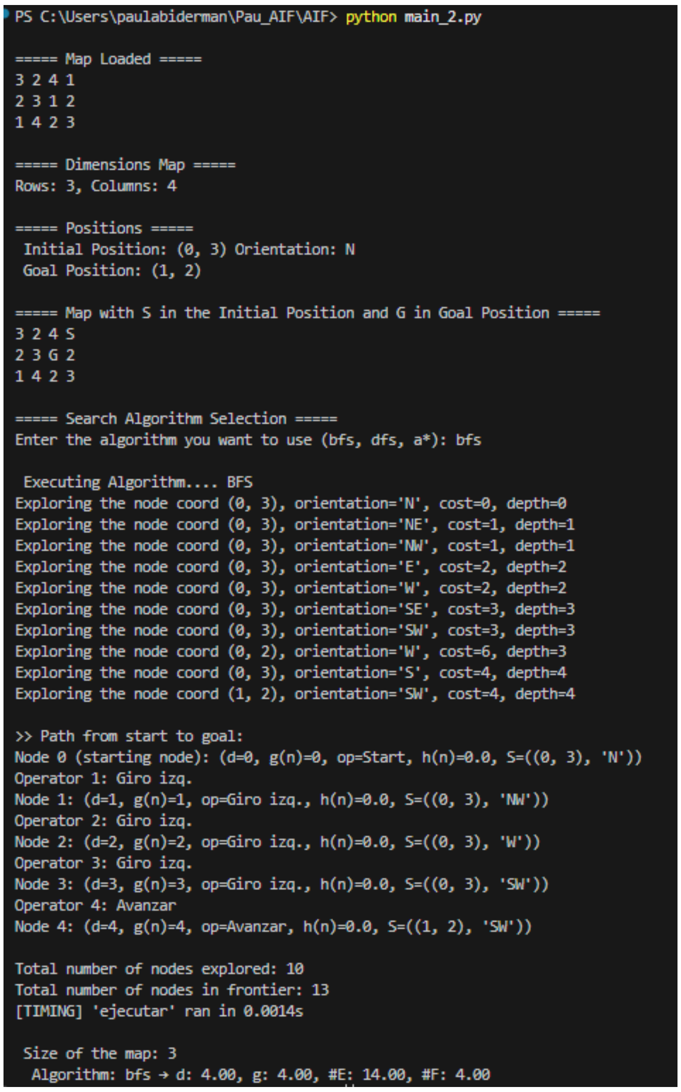
```
PS C:\Users\paulabiderman\Pau_AIF\AIF> python mainMapa.py
¿Which type of map do you want to use?
1. Load map from file
2. Generate a random map nxn
Choose an option (1 o 2): 2
Size of the map (ex. 5): 3
Seed for randomness (optional):

===== Map Loaded =====
6 6 6
4 5 5
2 1 6

===== Dimensions Map =====
Rows: 3, Columns: 3

===== Positions =====
Start: (0, 0) Orientation: N
Goal: (2, 2) Orientation: I

===== Map with S and G =====
S 6 6
4 5 5
2 1 G
```

5.3 Execution of Main_2.py

This script loads a predefined terrain from a file (exampleMap.txt), displays the map with the initial (S) and goal (G) positions, and allows the user to select a search algorithm (bfs, dfs, or a*). The selected planner executes the search and reports the key metrics: depth (d), cost (g), number of explored nodes (E), and frontier size (F). This script is mainly used for manually testing and comparing the behavior of each algorithm on a fixed map.

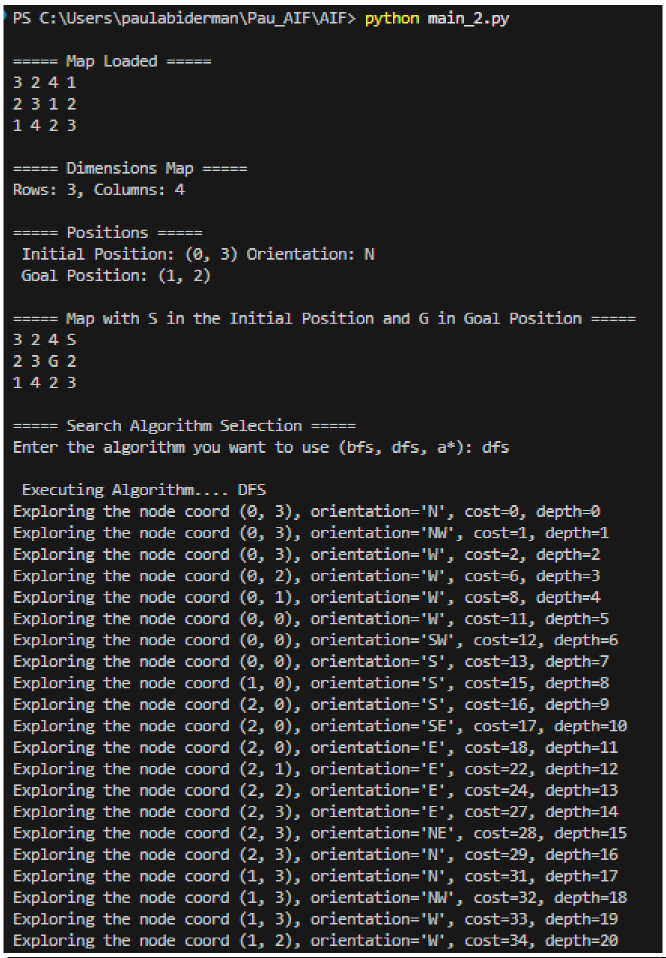
Below is shown the exeuction of the three possibles paths taken, one for each algorithm.

For BFS

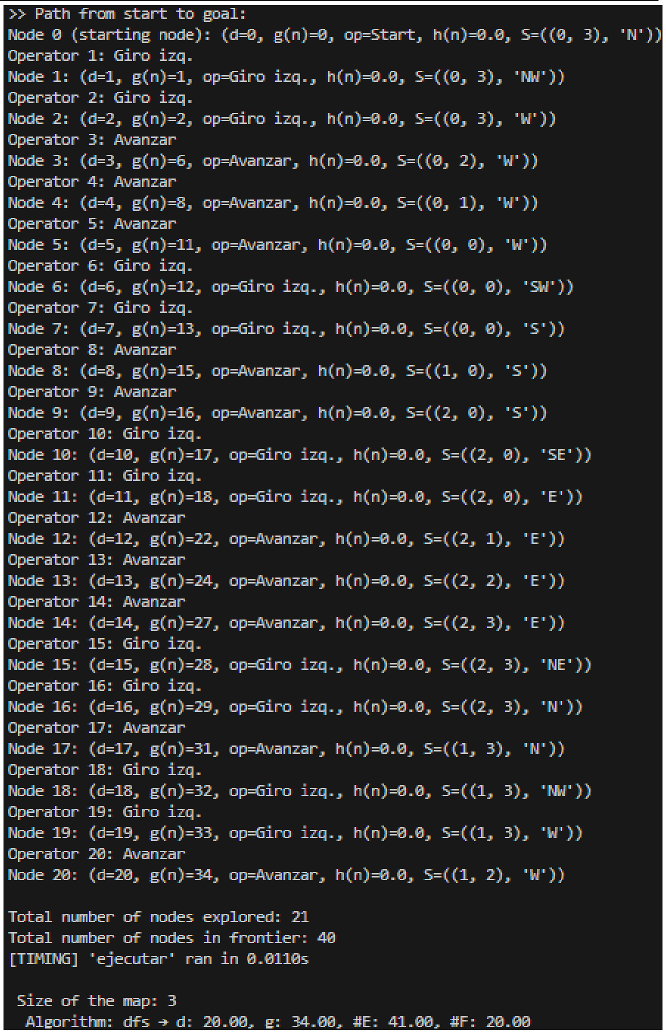


```
PS C:\Users\paulabiderman\Pau_AIF\AIF> python main_2.py

===== Map Loaded =====
3 2 4 1
2 3 1 2
1 4 2 3

===== Dimensions Map =====
Rows: 3, Columns: 4

===== Positions =====
 Initial Position: (0, 3) Orientation: N
 Goal Position: (1, 2)

===== Map with S in the Initial Position and G in Goal Position =====
3 2 4 S
2 3 G 2
1 4 2 3

===== Search Algorithm Selection =====
Enter the algorithm you want to use (bfs, dfs, a*): bfs

 Executing Algorithm.... BFS
Exploring the node coord (0, 3), orientation='N', cost=0, depth=0
Exploring the node coord (0, 3), orientation='NE', cost=1, depth=1
Exploring the node coord (0, 3), orientation='NW', cost=1, depth=1
Exploring the node coord (0, 3), orientation='E', cost=2, depth=2
Exploring the node coord (0, 3), orientation='W', cost=2, depth=2
Exploring the node coord (0, 3), orientation='SE', cost=3, depth=3
Exploring the node coord (0, 3), orientation='SW', cost=3, depth=3
Exploring the node coord (0, 2), orientation='W', cost=6, depth=3
Exploring the node coord (0, 3), orientation='S', cost=4, depth=4
Exploring the node coord (1, 2), orientation='SW', cost=4, depth=4

>> Path from start to goal:
Node 0 (starting node): (d=0, g(n)=0, op=Start, h(n)=0.0, S=((0, 3), 'N'))
Operator 1: Giro izq.
Node 1: (d=1, g(n)=1, op=Giro izq., h(n)=0.0, S=((0, 3), 'NW'))
Operator 2: Giro izq.
Node 2: (d=2, g(n)=2, op=Giro izq., h(n)=0.0, S=((0, 3), 'W'))
Operator 3: Giro izq.
Node 3: (d=3, g(n)=3, op=Giro izq., h(n)=0.0, S=((0, 3), 'SW'))
Operator 4: Avanzar
Node 4: (d=4, g(n)=4, op=Avanzar, h(n)=0.0, S=((1, 2), 'SW'))

Total number of nodes explored: 10
Total number of nodes in frontier: 13
[TIMING] 'ejecutar' ran in 0.0014s

 Size of the map: 3
  Algorithm: bfs → d: 4.00, g: 4.00, #E: 14.00, #F: 4.00
```

For DFS

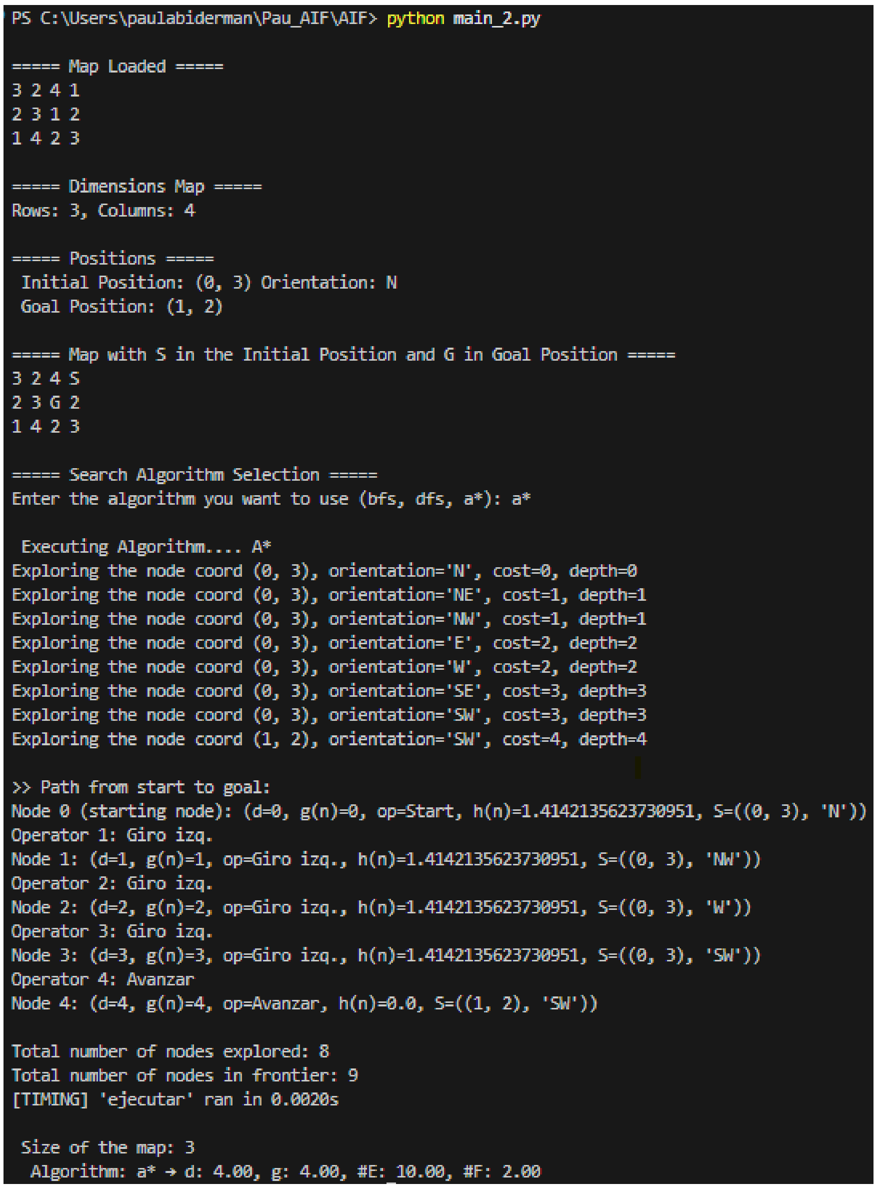


```
PS C:\Users\paulabiderman\Pau_AIF\AIF> python main_2.py

===== Map Loaded =====
3 2 4 1
2 3 1 2
1 4 2 3

===== Dimensions Map =====
Rows: 3, Columns: 4

===== Positions =====
 Initial Position: (0, 3) Orientation: N
 Goal Position: (1, 2)

===== Map with S in the Initial Position and G in Goal Position =====
3 2 4 S
2 3 G 2
1 4 2 3

===== Search Algorithm Selection =====
Enter the algorithm you want to use (bfs, dfs, a*): dfs

 Executing Algorithm.... DFS
Exploring the node coord (0, 3), orientation='N', cost=0, depth=0
Exploring the node coord (0, 3), orientation='NW', cost=1, depth=1
Exploring the node coord (0, 3), orientation='W', cost=2, depth=2
Exploring the node coord (0, 2), orientation='W', cost=6, depth=3
Exploring the node coord (0, 1), orientation='W', cost=8, depth=4
Exploring the node coord (0, 0), orientation='W', cost=11, depth=5
Exploring the node coord (0, 0), orientation='SW', cost=12, depth=6
Exploring the node coord (0, 0), orientation='S', cost=13, depth=7
Exploring the node coord (1, 0), orientation='S', cost=15, depth=8
Exploring the node coord (2, 0), orientation='S', cost=16, depth=9
Exploring the node coord (2, 0), orientation='SE', cost=17, depth=10
Exploring the node coord (2, 0), orientation='E', cost=18, depth=11
Exploring the node coord (2, 1), orientation='E', cost=22, depth=12
Exploring the node coord (2, 2), orientation='E', cost=24, depth=13
Exploring the node coord (2, 3), orientation='E', cost=27, depth=14
Exploring the node coord (2, 3), orientation='NE', cost=28, depth=15
Exploring the node coord (2, 3), orientation='N', cost=29, depth=16
Exploring the node coord (1, 3), orientation='N', cost=31, depth=17
Exploring the node coord (1, 3), orientation='NW', cost=32, depth=18
Exploring the node coord (1, 3), orientation='W', cost=33, depth=19
Exploring the node coord (1, 2), orientation='W', cost=34, depth=20
```



```
>> Path from start to goal:
Node 0 (starting node): (d=0, g(n)=0, op=Start, h(n)=0.0, S=((0, 3), 'N'))
Operator 1: Giro izq.
Node 1: (d=1, g(n)=1, op=Giro izq., h(n)=0.0, S=((0, 3), 'NW'))
Operator 2: Giro izq.
Node 2: (d=2, g(n)=2, op=Giro izq., h(n)=0.0, S=((0, 3), 'W'))
Operator 3: Avanzar
Node 3: (d=3, g(n)=6, op=Avanzar, h(n)=0.0, S=((0, 2), 'W'))
Operator 4: Avanzar
Node 4: (d=4, g(n)=8, op=Avanzar, h(n)=0.0, S=((0, 1), 'W'))
Operator 5: Avanzar
Node 5: (d=5, g(n)=11, op=Avanzar, h(n)=0.0, S=((0, 0), 'W'))
Operator 6: Giro izq.
Node 6: (d=6, g(n)=12, op=Giro izq., h(n)=0.0, S=((0, 0), 'SW'))
Operator 7: Giro izq.
Node 7: (d=7, g(n)=13, op=Giro izq., h(n)=0.0, S=((0, 0), 'S'))
Operator 8: Avanzar
Node 8: (d=8, g(n)=15, op=Avanzar, h(n)=0.0, S=((1, 0), 'S'))
Operator 9: Avanzar
Node 9: (d=9, g(n)=16, op=Avanzar, h(n)=0.0, S=((2, 0), 'S'))
Operator 10: Giro izq.
Node 10: (d=10, g(n)=17, op=Giro izq., h(n)=0.0, S=((2, 0), 'SE'))
Operator 11: Giro izq.
Node 11: (d=11, g(n)=18, op=Giro izq., h(n)=0.0, S=((2, 0), 'E'))
Operator 12: Avanzar
Node 12: (d=12, g(n)=22, op=Avanzar, h(n)=0.0, S=((2, 1), 'E'))
Operator 13: Avanzar
Node 13: (d=13, g(n)=24, op=Avanzar, h(n)=0.0, S=((2, 2), 'E'))
Operator 14: Avanzar
Node 14: (d=14, g(n)=27, op=Avanzar, h(n)=0.0, S=((2, 3), 'E'))
Operator 15: Giro izq.
Node 15: (d=15, g(n)=28, op=Giro izq., h(n)=0.0, S=((2, 3), 'NE'))
Operator 16: Giro izq.
Node 16: (d=16, g(n)=29, op=Giro izq., h(n)=0.0, S=((2, 3), 'N'))
Operator 17: Avanzar
Node 17: (d=17, g(n)=31, op=Avanzar, h(n)=0.0, S=((1, 3), 'N'))
Operator 18: Giro izq.
Node 18: (d=18, g(n)=32, op=Giro izq., h(n)=0.0, S=((1, 3), 'NW'))
Operator 19: Giro izq.
Node 19: (d=19, g(n)=33, op=Giro izq., h(n)=0.0, S=((1, 3), 'W'))
Operator 20: Avanzar
Node 20: (d=20, g(n)=34, op=Avanzar, h(n)=0.0, S=((1, 2), 'W'))

Total number of nodes explored: 21
Total number of nodes in frontier: 40
[TIMING] 'ejecutar' ran in 0.0110s

 Size of the map: 3
  Algorithm: dfs → d: 20.00, g: 34.00, #E: 41.00, #F: 20.00
```

For A*



```
PS C:\Users\paulabiderman\Pau_AIF\AIF> python main_2.py

===== Map Loaded =====
3 2 4 1
2 3 1 2
1 4 2 3

===== Dimensions Map =====
Rows: 3, Columns: 4

===== Positions =====
 Initial Position: (0, 3) Orientation: N
 Goal Position: (1, 2)

===== Map with S in the Initial Position and G in Goal Position =====
3 2 4 S
2 3 G 2
1 4 2 3

===== Search Algorithm Selection =====
Enter the algorithm you want to use (bfs, dfs, a*): a*

 Executing Algorithm.... A*
Exploring the node coord (0, 3), orientation='N', cost=0, depth=0
Exploring the node coord (0, 3), orientation='NE', cost=1, depth=1
Exploring the node coord (0, 3), orientation='NW', cost=1, depth=1
Exploring the node coord (0, 3), orientation='E', cost=2, depth=2
Exploring the node coord (0, 3), orientation='W', cost=2, depth=2
Exploring the node coord (0, 3), orientation='SE', cost=3, depth=3
Exploring the node coord (0, 3), orientation='SW', cost=3, depth=3
Exploring the node coord (1, 2), orientation='SW', cost=4, depth=4

>> Path from start to goal:
Node 0 (starting node): (d=0, g(n)=0, op=Start, h(n)=1.4142135623730951, S=((0, 3), 'N'))
Operator 1: Giro izq.
Node 1: (d=1, g(n)=1, op=Giro izq., h(n)=1.4142135623730951, S=((0, 3), 'NW'))
Operator 2: Giro izq.
Node 2: (d=2, g(n)=2, op=Giro izq., h(n)=1.4142135623730951, S=((0, 3), 'W'))
Operator 3: Giro izq.
Node 3: (d=3, g(n)=3, op=Giro izq., h(n)=1.4142135623730951, S=((0, 3), 'SW'))
Operator 4: Avanzar
Node 4: (d=4, g(n)=4, op=Avanzar, h(n)=0.0, S=((1, 2), 'SW'))

Total number of nodes explored: 8
Total number of nodes in frontier: 9
[TIMING] 'ejecutar' ran in 0.0020s

 Size of the map: 3
  Algorithm: a* → d: 4.00, g: 4.00, #E: 10.00, #F: 2.00
```

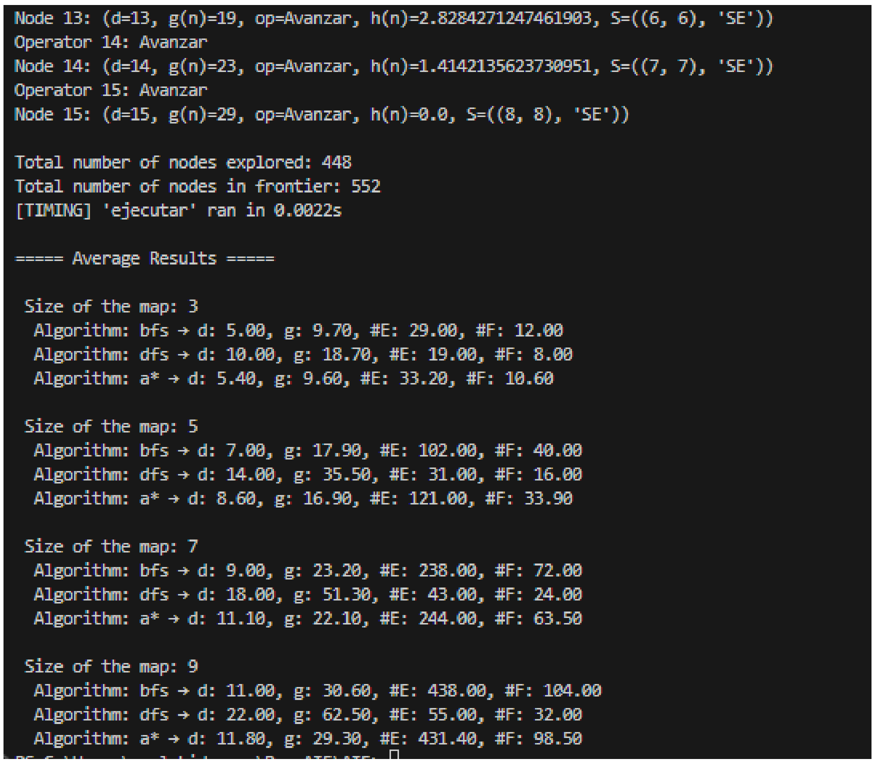
5.4 Execution of RunExperiments.py

This script automates the execution of BFS, DFS, and A* on randomly generated square maps of various sizes (3x3 to 9x9), running each configuration 10 times. It records performance metrics — solution depth, path cost, expanded nodes, and frontier size — and saves the results to a CSV file for statistical analysis and comparison.

Below it is shown the final portion of the execution as it is display in the terminal all the explore nodes and traces for each of the algorithms and each iteration for the different maps dimensions.

The final output of the script displays the average results of running BFS, DFS, and A* over 10 random maps for each size (3×3, 5×5, 7×7, and 9×9). For each configuration, it reports the average solution depth (d), total cost (g), number of explored nodes (#E), and frontier size (#F).

As expected, BFS guarantees the shortest path in steps but often explores more nodes and may yield higher costs. DFS explores fewer nodes but produces deeper and more costly paths. A*, guided by the Euclidean heuristic, balances cost and efficiency, achieving lower g and moderate #E across all sizes. These results support the theoretical expectations regarding blind vs informed search strategies.



```
Node 13: (d=13, g(n)=19, op=Avanzar, h(n)=2.8284271247461903, S=((6, 6), 'SE'))
Operator 14: Avanzar
Node 14: (d=14, g(n)=23, op=Avanzar, h(n)=1.4142135623730951, S=((7, 7), 'SE'))
Operator 15: Avanzar
Node 15: (d=15, g(n)=29, op=Avanzar, h(n)=0.0, S=((8, 8), 'SE'))

Total number of nodes explored: 448
Total number of nodes in frontier: 552
[TIMING] 'ejecutar' ran in 0.0022s

===== Average Results =====

 Size of the map: 3
  Algorithm: bfs → d: 5.00, g: 9.70, #E: 29.00, #F: 12.00
  Algorithm: dfs → d: 10.00, g: 18.70, #E: 19.00, #F: 8.00
  Algorithm: a* → d: 5.40, g: 9.60, #E: 33.20, #F: 10.60

 Size of the map: 5
  Algorithm: bfs → d: 7.00, g: 17.90, #E: 102.00, #F: 40.00
  Algorithm: dfs → d: 14.00, g: 35.50, #E: 31.00, #F: 16.00
  Algorithm: a* → d: 8.60, g: 16.90, #E: 121.00, #F: 33.90

 Size of the map: 7
  Algorithm: bfs → d: 9.00, g: 23.20, #E: 238.00, #F: 72.00
  Algorithm: dfs → d: 18.00, g: 51.30, #E: 43.00, #F: 24.00
  Algorithm: a* → d: 11.10, g: 22.10, #E: 244.00, #F: 63.50

 Size of the map: 9
  Algorithm: bfs → d: 11.00, g: 30.60, #E: 438.00, #F: 104.00
  Algorithm: dfs → d: 22.00, g: 62.50, #E: 55.00, #F: 32.00
  Algorithm: a* → d: 11.80, g: 29.30, #E: 431.40, #F: 98.50
```