Artificial Intelligence Fundamentals

Lab 1: Implementation of Search Algorithms

Master in Artificial Intelligence

2025–2026

Authors:

David Carballo Rodríguez

Antonio Vila Leis

Santiago Delgado Ferreiro

October 2025

Contents

1 Methods **3**
- 1.1 Formal Characterization of the Problem . . . 3
  - 1.1.1 State Representation . . . 3
  - 1.1.2 Set of Operators . . . 3
  - 1.1.3 Transition Model . . . 3
  - 1.1.4 Goal Test . . . 3
  - 1.1.5 Cost Function . . . 3
- 1.2 Analysis of Blind Search Methods . . . 3
  - 1.2.1 Breadth-First Search (BFS) . . . 3
  - 1.2.2 Depth-First Search (DFS) . . . 4
- 1.3 Heuristic for A* . . . 4
  - 1.3.1 Proposed Heuristic . . . 4
  - 1.3.2 Rationale and Justification . . . 4
  - 1.3.3 Admissibility Analysis . . . 5
  - 1.3.4 Monotonicity (Consistency) . . . 5

2 Results **5**
- 2.1 Execution Traces on `exampleMap.txt` . . . 5
  - 2.1.1 BFS Trace . . . 6
  - 2.1.2 DFS Trace . . . 6
  - 2.1.3 A* Trace . . . 6
- 2.2 Experimental Performance Analysis . . . 7
  - 2.2.1 Experimental Setup . . . 7
  - 2.2.2 Performance Metrics . . . 7
  - 2.2.3 Comparative Results . . . 7

3 Discussion **8**
- 3.1 Comparative Analysis of Implemented Methods . . . 8
  - 3.1.1 Breadth-First Search (BFS) . . . 8
  - 3.1.2 Depth-First Search (DFS) . . . 8
  - 3.1.3 A* Search . . . 9
- 3.2 Heuristic Performance Analysis . . . 9
  - 3.2.1 Choice of Euclidean Distance . . . 9
  - 3.2.2 Performance and Behavior . . . 9
  - 3.2.3 Possible Improvements . . . 10
- 3.3 Extension: Impassable Rocks . . . 10
  - 3.3.1 Problem Representation . . . 10
  - 3.3.2 Impact on Problem Characteristics . . . 10
  - 3.3.3 Heuristic Validity . . . 10
  - 3.3.4 Algorithm Suitability . . . 11

A Source Code Documentation **12**
- A.1 Overview of Implementation . . . 12
- A.2 State Representation (`state.py`) . . . 12
- A.3 Search Node (`node.py`) . . . 12
- A.4 Problem Specification (`problem.py`) . . . 12
- A.5 Breadth-First Search (`search.py`) . . . 13
- A.6 Depth-First Search (`search.py`) . . . 13
- A.7 A* Search (`search.py`) . . . 13
- A.8 Main Program (`main.py`) . . . 13

A.9 Experiment Automation (`run_experiments.py`) 14

1 Methods

1.1 Formal Characterization of the Problem

1.1.1 State Representation

A state is defined as $S = (x, y, o)$ where $x \in \{0, \ldots, N-1\}$ is the row coordinate, $y \in \{0, \ldots, M-1\}$ is the column coordinate, and $o \in \{0, 1, 2, 3, 4, 5, 6, 7\}$ is the robot orientation (0=North, 1=Northeast, 2=East, 3=Southeast, 4=South, 5=Southwest, 6=West, 7=Northwest). The state space has $|\mathcal{S}| = N \times M \times 8$ states.

1.1.2 Set of Operators

Three operators are available, always generated in fixed order:

1. **rotate_right**: Effect $o' = (o + 1) \bmod 8$, Cost = 1
2. **move**: Effect $(x', y') = (x+\Delta x, y+\Delta y)$, Cost = hardness$(x', y')$. Precondition: destination within map bounds.
3. **rotate_left**: Effect $o' = (o - 1) \bmod 8$, Cost = 1

For each orientation, displacements are: 0:(-1,0), 1:(-1,1), 2:(0,1), 3:(1,1), 4:(1,0), 5:(1,-1), 6:(0,-1), 7:(-1,-1).

1.1.3 Transition Model

Function successors(s, M) generates valid successors in R→M→L order. DFS reverses this order on the stack so expansion follows the same R→M→L sequence.

1.1.4 Goal Test

$$\text{is\_goal}(s, s_{\text{goal}}) = \begin{cases} (x = x_{\text{goal}}) \wedge (y = y_{\text{goal}}) & \text{if } o_{\text{goal}} = 8 \\ s = s_{\text{goal}} & \text{otherwise} \end{cases}$$

1.1.5 Cost Function

Path cost: $g(n) = \sum_{i=1}^{d} c(\text{action}_i)$ where rotations cost 1 and moves cost the destination cell's hardness.

1.2 Analysis of Blind Search Methods

1.2.1 Breadth-First Search (BFS)

Algorithm Description. BFS explores the state space level by level, guaranteeing to find the solution with minimum depth. It uses a FIFO queue (`collections.deque`) to manage the frontier.

Implementation Details.

- **Data structures:** FIFO queue (deque), explored set, frontier_set for $O(1)$ duplicate checks
- **Successor order:** Uses shared `successors()` function (R→M→L order)
- **Goal detection:** Halts immediately when goal is generated

Theoretical Properties.

- **Completeness:** Yes (finite state space with duplicate detection)
- **Optimality:** Finds minimum depth, but NOT minimum cost (non-uniform costs)

Justification for this problem. BFS is preferable to DFS due to guaranteed completeness and shorter solutions, though it doesn't guarantee minimum cost given heterogeneous action costs.

1.2.2 Depth-First Search (DFS)

Algorithm Description. DFS explores each branch to maximum depth before backtracking. Uses a LIFO stack (Python list).

Implementation Details.

- **Data structures:** LIFO stack (list), explored set, frontier_set
- **Successor order:** Pushes in reverse (L→M→R) so pop() yields R→M→L
- **Goal detection:** Terminates on first solution found (may be deep/suboptimal)

Theoretical Properties.

- **Completeness:** Not complete without depth control; complete with duplicate detection on finite grids
- **Optimality:** NO — finds any solution, not necessarily the best

Justification for this problem. DFS unsuitable for this domain: lacks cost optimization, can find very suboptimal solutions (as confirmed experimentally), and explores deep unproductive branches without heuristic guidance.

1.3 Heuristic for A*

1.3.1 Proposed Heuristic

$$h(s) = d_{\text{Euclidean}}(s, s_{\text{goal}}) + c_{\text{rotation}}(s, s_{\text{goal}})$$

where:

$$d_{\text{Euclidean}} = \sqrt{(x_{\text{goal}} - x)^2 + (y_{\text{goal}} - y)^2}$$

$$c_{\text{rotation}} = \begin{cases} \min(|o_{\text{goal}} - o|, 8 - |o_{\text{goal}} - o|) & \text{if } o_{\text{goal}} \neq 8 \\ 0 & \text{if } o_{\text{goal}} = 8 \text{ (wildcard)} \end{cases}$$

1.3.2 Rationale and Justification

Euclidean distance component:

- Represents straight-line distance (shortest geometric path)
- Optimistic: assumes direct movement without considering terrain hardness
- Generally underestimates the true cost, but may slightly overestimate in diagonal movements where the real cost per step exceeds the geometric distance traveled

Rotation component:

- Computes minimum angular distance (circular metric modulo 8)
- Exact cost for reaching target orientation (each rotation = 1 unit)
- Returns 0 when target orientation is wildcard (code 8)

1.3.3 Admissibility Analysis

The heuristic $h(s)$ is **admissible in most practical cases**, as it generally does not overestimate the true remaining cost. However, it is not strictly admissible in all configurations, because:

1. The Euclidean component assumes a minimum cost per movement of 1, but the real terrain cost can be higher due to rock hardness. In diagonal moves, the geometric distance ($\sqrt{2}$) may exceed the actual cost of advancing if the terrain is soft, or underestimate it if it is hard.
2. The rotation component is exact and does not affect admissibility.
3. Therefore, while the heuristic typically provides an optimistic estimate, minor overestimations may occur in isolated cases (e.g., long diagonal movements across hard terrain).

In practice, $h(s)$ remains a reasonable and effective heuristic for A*, and in the experiments performed it did not lead to suboptimal solutions.

1.3.4 Monotonicity (Consistency)

A heuristic is **monotonic** if $h(n) \leq c(n, a, n') + h(n')$ for all successors n'.

Analysis:

- **Rotations:** $h(n) = d + r$ changes to $h(n') = d + (r \pm 1)$ with cost 1. Satisfies: $d + r \leq 1 + (d + (r - 1))$ ✓
- **Movements:** Euclidean distance decreases by at most the traveled distance, while real cost ≥ 1. The heuristic is mostly monotonic, though small inconsistencies may appear when moving diagonally across hard terrain.

Importance of monotonicity: Monotonicity ensures that the first expansion of each node yields the optimal path, avoiding re-expansions and improving efficiency. Although our heuristic is not strictly monotonic in all cases, it behaves consistently in practice and maintains A* performance and optimality in all tested maps.

2 Results

2.1 Execution Traces on `exampleMap.txt`

The example map is a 3×4 grid with starting position $(0, 3, 0)$ (row 0, column 3, facing North) and goal position $(1, 2, 8)$ where orientation 8 indicates any final orientation is acceptable. Below are the solution traces produced by each algorithm.

2.1.1 BFS Trace

```
Node(depth, cost, operator, state)
Node 0 (starting node)
  (0, 0, START, (0, 3, 0))
Operator 1
  rotate_left
Node 1
  (1, 1, rotate_left, (0, 3, 7))
Operator 2
  rotate_left
Node 2
  (2, 2, rotate_left, (0, 3, 6))
Operator 3
  rotate_left
Node 3
  (3, 3, rotate_left, (0, 3, 5))
Operator 4
  move
Node 4
  (4, 4, move, (1, 2, 5))
Total number of items in explored list: 8
Total number of items in frontier: 5
```

2.1.2 DFS Trace

```
Node(depth, cost, operator, state)
Node 0 (starting node)
  (0, 0, START, (0, 3, 0))
Operator 1
  rotate_right
Node 1
  (1, 1, rotate_right, (0, 3, 1))
Operator 2
  rotate_right
Node 2
  (2, 2, rotate_right, (0, 3, 2))
Operator 3
  rotate_right
Node 3
  (3, 3, rotate_right, (0, 3, 3))
Operator 4
  rotate_right
Node 4
  (4, 4, rotate_right, (0, 3, 4))
Operator 5
  rotate_right
Node 5
  (5, 5, rotate_right, (0, 3, 5))
Operator 6
  move
Node 6
  (6, 6, move, (1, 2, 5))
Total number of items in explored list: 6
Total number of items in frontier: 3
```

2.1.3 A* Trace

```
Node(depth, cost, operator, h, state)
Node 0 (starting node)
  (0, 0, START, 1.41, (0, 3, 0))
Operator 1
  rotate_left
Node 1
  (1, 1, rotate_left, 1.41, (0, 3, 7))
Operator 2
  rotate_left
Node 2
  (2, 2, rotate_left, 1.41, (0, 3, 6))
Operator 3
  rotate_left
Node 3
  (3, 3, rotate_left, 1.41, (0, 3, 5))
Operator 4
  move
Node 4
  (4, 4, move, 0.00, (1, 2, 5))
Total number of items in explored list: 8
Total number of items in frontier: 2
```

2.2 Experimental Performance Analysis

2.2.1 Experimental Setup

All experiments were performed on randomly generated maps of four different sizes: 3×3, 5×5, 7×7, and 9×9. For each map size, 10 independent random configurations were created, and the reported results correspond to the average values computed over the 10 trials for each dimension.

2.2.2 Performance Metrics

- **d**: Solution depth (number of actions in path)
- **g**: Solution cost (sum of action costs)
- **#E**: Number of nodes explored (expanded)
- **#F**: Final frontier size (nodes in queue when solution found including it)

2.2.3 Comparative Results

Table 1: Performance comparison for 3×3 maps

Algorithm	**d**	**g**	**#E**	**#F**
BFS	5.0	12.0	10.0	6.0
DFS	21.0	34.4	24.0	9.0
A*	5.4	11.8	29.4	8.2

Table 2: Performance comparison for 5×5 maps

Algorithm	**d**	**g**	**#E**	**#F**
BFS	7.0	24.0	30.0	26.0
DFS	79.0	128.0	119.0	16.0
A*	8.8	23.0	127.6	25.4

Table 3: Performance comparison for 7×7 maps

Algorithm	**d**	**g**	**#E**	**#F**
BFS	9.0	32.8	97.0	56.0
DFS	136.0	213.2	178.0	29.0
A*	10.2	30.8	262.6	45.8

Table 4: Performance comparison for 9×9 maps

Algorithm	**d**	**g**	**#E**	**#F**
BFS	11.0	43.2	225.0	88.0
DFS	210.0	328.8	258.0	43.0
A*	14.6	38.4	446.6	73.4

3 Discussion

3.1 Comparative Analysis of Implemented Methods

3.1.1 Breadth-First Search (BFS)

Observed Advantages:

- Always finds the solution with the minimum depth, confirming its completeness and optimality in terms of number of actions.
- Maintains predictable memory growth: although the frontier increases with map size, it does so in a controlled manner without abrupt spikes.
- Exhibits stable and easily interpretable behavior, as it follows a systematic level by level exploration.

Observed Disadvantages:

- Ignores terrain costs, so it may traverse highly resistant cells if they shorten the geometric distance.
- The total cost of its trajectories grows rapidly with map size: it finds short paths but with significantly higher costs than A*.
- Its memory usage grows exponentially with the depth of the search tree, which can become problematic for large maps.

Explanation: BFS prioritizes distance in terms of steps rather than accumulated cost, making it suitable for finding quick routes in terms of the number of actions, but not necessarily the cheapest ones in terms of energy or drilling time. It performs well on small or uniform maps but its efficiency degrades as terrain variability increases.

3.1.2 Depth-First Search (DFS)

Observed Advantages:

- Low memory consumption, keeping small frontiers even on large maps.
- Can reach the goal quickly if the initial exploration direction happens to be favorable.

Observed Disadvantages:

- Produces very poor quality solutions: paths are long and expensive, and performance worsens as map size increases.
- Explores deep branches without considering cost or proximity to the goal, generating redundant movements and unnecessary rotations.
- Despite using little memory, it can expand more nodes than BFS due to its unstructured exploration.
- Does not guarantee finding the optimal solution, not even in terms of depth.

Explanation: DFS is inefficient for this problem because it has no notion of direction or cost. Its tendency to rotate repeatedly and move blindly results in erratic and expensive trajectories. In our experiments, its solutions were up to an order of magnitude worse than those of A*, demonstrating that DFS is not suitable when efficiency or solution quality is required.

3.1.3 A* Search

Observed Advantages:

- Always obtains the lowest cost paths, outperforming BFS and DFS in efficiency.
- Exploits terrain information by avoiding high hardness cells and balancing distance and cost.
- Maintains path lengths similar to BFS but with lower costs, showing effective heuristic guidance.
- Exhibits stable and predictable behavior as map size increases.

Observed Disadvantages:

- Requires more memory and time than BFS, as it expands a larger number of nodes and keeps a bigger frontier.
- The Euclidean heuristic, while providing useful guidance, underestimates real costs by ignoring terrain hardness.
- Its computational cost increases noticeably on larger maps, although it still produces the most efficient routes.

Explanation: A* represents the best trade off between cost and exploration. Thanks to its heuristic, it directs the search toward promising regions and avoids paths with high accumulated cost. Although the heuristic is not perfectly informative, its approximation is sufficient to significantly improve performance compared to BFS. Overall, A* is the most balanced and effective method for this type of navigation problem in heterogeneous terrains.

3.2 Heuristic Performance Analysis

3.2.1 Choice of Euclidean Distance

The heuristic combines Euclidean distance with a rotation cost component. We chose Euclidean distance because the robot can move diagonally, making the straight-line distance a natural geometric estimate of proximity to the goal.

3.2.2 Performance and Behavior

Positive Aspects:

- **Good directional guidance:** Effectively steers the search toward the goal.
- **Rotation awareness:** Incorporates the orientation difference between the current state and the goal.
- **Fast to compute:** $O(1)$ per state, enabling efficient heuristic evaluation.
- **Monotonic behavior:** The heuristic generally satisfies consistency, ensuring that $f(n)$ values remain non decreasing along paths.

Limitations Observed:

- **Underestimates costs:** Assumes uniform terrain cost ($= 1$), while the real terrain varies from 1 to 9, leading to additional exploration.
- **Ignores terrain variation:** Two points at equal geometric distance may have very different traversal costs.

- **Ignores intermediate rotations:** Does not account for orientation changes that occur during movement.
- **Assumes straight lines:** Real paths are constrained to the grid and rarely follow perfect diagonals.
- **Minor overestimation on diagonals:** Euclidean distance ($\sqrt{2} \approx 1.41$) can slightly exceed the actual move cost (1), but this effect is rare and localized.

3.2.3 Possible Improvements

- **Terrain-aware:** Incorporate average terrain hardness along the direct path.
- **Weighted Euclidean:** Multiply the Euclidean distance by the minimum cell cost to maintain admissibility.
- **Precomputed terrain distances:** Use a 2D Dijkstra search from the goal backward to obtain exact terrain aware distances (computationally expensive but optimal).

Summary: Although A* expanded more nodes than BFS due to heuristic underestimation, it consistently produced cheaper paths. This demonstrates that a weak heuristic can increase exploration but still achieve better cost efficiency. In our problem, the Euclidean heuristic provided good guidance and, despite its simplifications, generally led to optimal solutions. It thus represents a simple yet effective choice for this navigation domain.

3.3 Extension: Impassable Rocks

3.3.1 Problem Representation

Map encoding: Use a special value (e.g., 0 or -1) to mark impassable cells, or equivalently assign an infinite cost (∞).

Operator modification: The `move` action requires an additional precondition:

$$\text{precondition(move)} = (0 \leq x' < N) \wedge (0 \leq y' < M) \wedge (\text{hardness}(x', y') \neq \text{IMPASSABLE})$$

3.3.2 Impact on Problem Characteristics

- **Increased complexity:** The robot must navigate around obstacles, increasing average path length.
- **Potential incompleteness:** The goal may become unreachable if completely surrounded by impassable cells.
- **Reduced local branching:** The effective branching factor b decreases near obstacles.
- **State-space modification:** The overall number of reachable states decreases, though local exploration density near barriers may rise.

3.3.3 Heuristic Validity

Euclidean heuristic:

- **Remains admissible:** The straight line estimate still never exceeds the true minimal cost.
- **Less informative:** May point directly through obstacles, misleading the search.
- **Increased exploration:** A* must explore more nodes around blocked regions to find valid paths.

Improvement — Obstacle-aware heuristic: Precompute true terrain distances (ignoring orientation) using a backward Dijkstra search from the goal. This approach is more computationally expensive but significantly more informative.

3.3.4 Algorithm Suitability

- **BFS:** Remains complete but still not cost-optimal.
- **DFS:** Requires a depth limit (IDDFS) to guarantee completeness; still highly inefficient.
- **A*:** *Best choice:* preserves completeness and cost optimality, even though efficiency decreases due to increased branching complexity.

Conclusion: When impassable obstacles are introduced, A* becomes clearly the most appropriate algorithm, as it is the only one that guarantees both completeness and cost optimality in this extended problem setting.

A Source Code Documentation

A.1 Overview of Implementation

The implementation consists of 5 main Python modules:

- `state.py`: State representation
- `node.py`: Search tree node with path reconstruction
- `problem.py`: Problem specification (successors, goal check, map loading)
- `search.py`: Implementation of BFS, DFS, and A* algorithms
- `main.py`: Command-line interface for running experiments

Additionally, `run_experiments.py` automates performance testing on random maps.

A.2 State Representation (`state.py`)

Purpose: Defines the state space representation using an immutable dataclass.

Key design decisions:

- Compact `__str__`: Formatted output for debugging/tracing

A.3 Search Node (`node.py`)

Purpose: Represents nodes in the search tree with parent pointers for path reconstruction.

Key attributes:

- `state`: Current state (position + orientation)
- `parent`: Link to parent node for backtracking
- `op`: Action taken to reach this node
- `g`: Accumulated path cost from root
- `depth`: Number of actions from root

Methods:

- `path()`: Traces parent pointers to root, returns solution path
- `__lt__`: Dummy comparison for heap tie-breaking

A.4 Problem Specification (`problem.py`)

Purpose: Centralizes problem definition: successor generation, goal testing, and map I/O.

Key components:

- `MOVES`: Global array mapping orientations to $(\Delta x, \Delta y)$ displacements
- `load_map()`: Parses map file (dimensions, terrain matrix, start/goal states)
- `successors()`: Generates valid successors in fixed R→M→L order
- `is_goal()`: Checks goal satisfaction (handles orientation wildcard)

Design rationale: Centralizing `successors()` ensures all algorithms (BFS, DFS, A*) explore in consistent order, enabling fair comparison.

A.5 Breadth-First Search (`search.py`)

Implementation highlights:

- Uses `collections.deque` for efficient $O(1)$ append/popleft
- `frontier_set` provides $O(1)$ duplicate checking (avoids linear search)
- `explored` set prevents state re-expansion
- Goal check is performed on generation for early termination
- Verbose mode provides step-by-step execution trace

A.6 Depth-First Search (`search.py`)

Implementation highlights:

- Uses a Python `list` as a stack for efficient LIFO node expansion
- `frontier_set` enables $O(1)$ duplicate checking
- `explored` set prevents state re-expansion
- Successors are stacked in reverse order to ensure exploration order: rotate_right $\rightarrow$ move $\rightarrow$ rotate_left
- Goal check is performed on generation for early termination
- Verbose mode provides step-by-step execution trace

A.7 A* Search (`search.py`)

Implementation highlights:

- Uses `heapq` for priority queue (min-heap by $f(n) = g(n) + h(n)$)
- `itertools.count()` for tie-breaking (FIFO when f values equal)
- `frontier_best` dict tracks best g per state (lazy deletion)
- Stale entry detection: skips outdated heap entries
- `explored` dict maps states to best-known g values

Heuristic function: Euclidean distance + minimum rotation cost.

A.8 Main Program (`main.py`)

Purpose: Command-line interface for running search algorithms on map files.

Features:

- Accepts algorithm choice (bfs, dfs, astar) via command line
- Optional verbose mode for detailed execution traces
- Outputs solution path with node details
- Reports explored and frontier statistics

A.9 Experiment Automation (`run_experiments.py`)

Purpose: Automates performance testing on randomly generated maps.

Features:

- Generates random maps of configurable sizes (3×3, 5×5, 7×7, 9×9)
- Runs all three algorithms (BFS, DFS, A*) on each map
- Collects statistics: depth, cost, nodes explored, frontier size
- Averages results over multiple trials (default: 5 per size)
- Outputs formatted LaTeX tables for report integration