University of Rome “La Sapienza”

Department of Ingegneria Informatica, Automatica e Gestionale

Artificial Intelligence

Homework

Comparison of A* Search and PDDL Classical Planning on Grid-Based Pathfinding Benchmarks

SAPIENZA
Università di Roma

Arianna Navarra

Matricula:
2195207

1. Introduction

This project compares two approaches to grid-based pathfinding: **A*** heuristic search and **PDDL classical planning**. Both operate on identical GridWorld environments generated from the MovingAI and GPPC benchmark datasets, ensuring a fair comparison.

A unified **GridWorld** model loads .map files, identifies traversable cells and obstacles, manages start/goal positions, and generates neighbors under 4- or 8-connected movement. This same representation is used by both A* and by the automatic PDDL problem generator.

The **A*** implementation includes duplicate elimination, no node reopening, and admissible heuristics (Manhattan for 4-connected grids, Euclidean for 8-connected grids). It collects detailed metrics such as expanded/generated nodes, branching factor statistics, frontier sizes, runtime, and path length.

For **PDDL planning**, each GridWorld instance is translated into a domain-independent PDDL problem and solved with Fast Downward. The resulting plan is parsed back into a path, and the planner's own statistics (expanded/generated states, plan length, plan cost, runtime) are extracted.

The comparison shows a clear gap in efficiency:

A* is consistently faster and explores far fewer states, whereas Fast Downward introduces significant overhead despite producing plans with the same optimal path length. This highlights the difference between a domain-specific search algorithm and a general-purpose classical planner.

2. Task 1: Problem

The problem addressed in this assignment is **grid-based pathfinding**: given a 2D map where each cell is either free or blocked, the objective is to compute a valid path from a start state to a goal state under a specified connectivity model (4-connected or 8-connected).

This setting corresponds to the classical benchmark environments used in the MovingAI and GPPC competitions.

Benchmark Maps and Data Loading

The project uses real pathfinding maps from the MovingAI / GPPC datasets. Because these datasets are too large to store locally, they are automatically downloaded and extracted at runtime:

- the Bitbucket benchmark ZIP is downloaded,
- all relevant folders (local, gppc-2013, gppc-2014, movingai…) are extracted,
- the .map files are reorganized under a consistent /benchmarks/maps/ directory.

Each .map file follows the standard MovingAI format, which specifies:

- grid dimensions,
- obstacle tiles,
- free cells,
- optional **S** (start) and **G** (goal) markers.

This ensures that both A* and PDDL planners operate on real, structured datasets.

GridWorld Environment

All maps are parsed into a dedicated Python class, **GridWorld**, which provides a uniform, algorithm-agnostic representation of the environment:

- the grid matrix (height × width),
- detection of walls and free cells,
- automatic extraction of start/goal if present,
- support for **4-connected** or **8-connected** movement,
- adjacency computation (neighbors),
- cost model (1 for straight moves, √2 for diagonal moves).

This environment is used by *both* A* and the PDDL generator.

It abstracts away the low-level details of map parsing and exposes a clean interface for search.

Generic Problem Wrapper

To run A*, the GridWorld is wrapped inside a GridWorldProblem interface providing:

- initial_state
- goal_state
- actions(state) → neighbors
- child_node(state, action)
- cost(s1, s2)
- is_goal(s)

This interface matches the canonical definition of a search problem used in AI Search.

Fallback Start/Goal Selection

If a map does not contain S or G, a **double-BFS diameter heuristic** is used to select a meaningful, distant start–goal pair. This ensures that every map yields a non-trivial pathfinding problem.

Why this Problem Representation

This modeling choice ensures:

- **consistency** between A* and PDDL planning (they receive the *same* underlying grid),
- **scalability** to large benchmark maps,
- **correctness** of neighbor generation and movement costs,
- **compatibility** with classical planning (PDDL domain uses the same adjacency model).

3. Task 2.1: Implementation of A*

A* Overview and Design Choices

My implementation follows the canonical definition of A*: **f(n) = g(n) + h(n)**

- **g(n)**: exact cost accumulated from the start
- **h(n)**: heuristic estimate to the goal
- **f(n)**: priority value used to expand nodes

The algorithm uses: **duplicate elimination** (a closed list), **no node re-opening** once a state is expanded, **admissible, consistent heuristics** (Manhattan or Euclidean), a **priority queue** based on Python's heapq module.

Data Structures

a. **Frontier (OPEN list):** Implemented using a **binary min-heap** (heapq), where each entry is: ***(f_value, state)***. This guarantees efficient extraction of the minimum f-value node **(O(log n))**. In addition, I maintain a dictionary: ***frontier_dict[state] = f_value***. This auxiliary structure allows a constant-time checks for duplicates inside the frontier.
b. **Closed Set (Explored Nodes):** A simple Python set() stores all states already expanded: ***explored = set().*** This ensures duplicate elimination and prevents re-expansion of already visited nodes.

Parent and Cost Dictionaries

To reconstruct the optimal path, I track: ***parent[state] = predecessor, g_score[state] = cost from start to state.*** At goal discovery, the path is reconstructed efficiently by backtracking.

Heuristics

Two heuristics are used depending on the grid connectivity:

- **Manhattan distance (4-connected grids**): Admissible and consistent when only cardinal moves are allowed.
- **Euclidean distance (8-connected grids):** Admissible when diagonal movement is allowed and has cost √2.

Branching Factor Tracking

For each expanded state, I compute: ***bf = number of available actions = len(neighbors(state)).*** All values are stored in self.bf_values, and at the end A* computes: minimum branching factor, maximum branching factor, average branching factor.

Implementation Details Worth Mentioning

1. No Re-opening: The algorithm does *not* re-open nodes when a cheaper path is found. This matches the course slides and ensures: faster execution and correct optimality thanks to consistent heuristics.

2. Start/Goal Selection: If a map does not contain explicit S or G markers, I automatically compute a meaningful pair using a **double BFS diameter approximation**, selecting two farthest reachable points. This ensures that every map produces a non-trivial pathfinding instance.

3. Cost Model: Movement cost is: **1.0** for horizontal/vertical moves**,** **√2** for diagonal moves (only in 8-connected mode)**.** This matches the geometry of real grid navigation.

Final Structure of A*

The implementation centers around a clean, modular AStar class that receives a generic GridWorldProblem, making it reusable and independent of the underlying grid representation. The algorithm returns: the optimal path, all required search metrics (expanded, generated, frontier size, branching factor statistics), runtime and path length.

4. Task 2.2: Implementation of PDDL Planning (Fast Downward)

The implementation consists of **three main components**:

1. PDDL Domain Definition

The domain encodes grid navigation in a minimal STRIPS formalism. Each grid cell is a typed object (cell), and the domain defines four predicates: free(x) the cell is traversable, blocked(x) the cell is an obstacle, adjacent(a,b) valid transitions between cells (4 or 8 connectivity), agent-at(x) the current position of the agent.

Only one action is required:

```
(:action move
 :parameters (?from ?to - cell)
 :precondition (and (agent-at ?from) (adjacent ?from ?to) (free ?to))
 :effect      (and (not (agent-at ?from)) (agent-at ?to)))
```

The domain is written once and reused for all experiments.

2. Automatic Problem Generation

For every GridWorld instance, I automatically generate a corresponding PDDL problem containing: a list of all cells as objects, free/blocked predicates based on the .map file, adjacent predicates computed exactly from the GridWorld neighbor function, the initial state (agent-at start), the goal state (agent-at goal). This ensures that **A*** and **PDDL planning use the exact same topology**, start/goal configuration, and movement rules.

Problem files are written in the form:

```
(define (problem gridworld-problem)
```

(:domain gridworld)

(:objects ... - cell)

(:init ... predicates ...)

(:goal (agent-at c_x_y)))

3. Planning + Plan Reconstruction

Fast Downward is executed programmatically using: ***./fastdownward/fast-downward.py domain.pddl problem.pddl --search "astar(blind())"*** . The planner output is parsed to extract: number of expanded states, number of generated states, plan length, plan cost, total planning time. The sas_plan file is then read to extract the symbolic actions: ***(move c_x1_y1 c_x2_y2).*** These are converted back to grid coordinates, producing a path directly comparable to the A* solution.

This implementation creates a complete and general pipeline:**GridWorld → PDDL Domain + Problem → Fast Downward → Plan Parsing → Path Reconstruction**

The resulting planner behaves as a *domain-independent solver*, allowing a direct experimental comparison against A* on identical map instances.

5. Task 3: Experimental Results

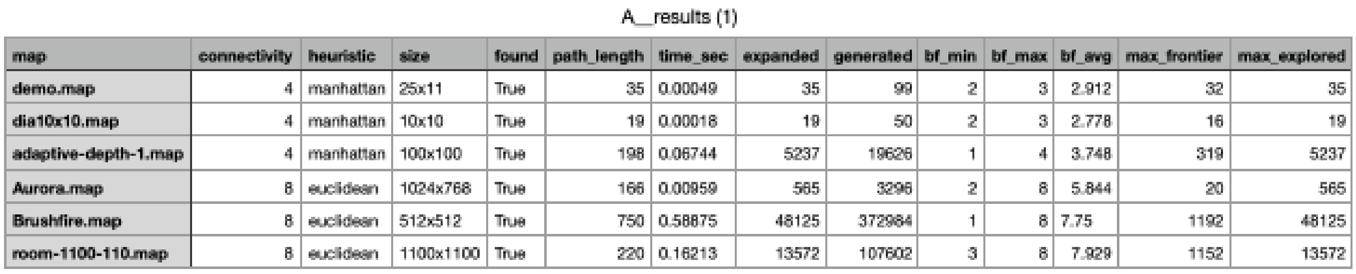
This section reports the experimental evaluation of the two implemented approaches, A* search and PDDL planning, on identical GridWorld benchmark maps.

The analysis is based on the three result tables: **A*_results.csv** (metrics from the A* implementation), **pddl_results.csv** (metrics extracted from Fast Downward), **comparison_results.csv** (merged metrics for direct comparison).

The experiments were conducted by increasing a **scaling parameter** as required by the homework. The scaling parameter is **the size and structural complexity of the benchmark maps**, ranging from small Local maps to large GPPC competition maps. This produces a natural growth in the search space, directly affecting runtime, number of expanded/generated nodes and branching behavior.

The project also includes visual comparisons of runtime, path length, runtime ratios, and examples of computed paths.

1. A* Metrics Summary


A__results (1)

map	connectivity	heuristic	size	found	path_length	time_sec	expanded	generated	bf_min	bf_max	bf_avg	max_frontier	max_explored
demo.map	4	manhattan	25x11	True	35	0.00049	35	99	2	3	2.912	32	35
dia10x10.map	4	manhattan	10x10	True	19	0.00018	19	50	2	3	2.778	16	19
adaptive-depth-1.map	4	manhattan	100x100	True	198	0.06744	5237	19626	1	4	3.748	319	5237
Aurora.map	8	euclidean	1024x768	True	166	0.00959	565	3296	2	8	5.844	20	565
Brushfire.map	8	euclidean	512x512	True	750	0.58875	48125	372984	1	8	7.75	1192	48125
room-1100-110.map	8	euclidean	1100x1100	True	220	0.16213	13572	107602	3	8	7.929	1152	13572

The A* experiments show a consistent and predictable behavior across all tested grid maps, from very small Local instances to large GPPC benchmark environments.

Runtime Performance

A* solves all instances extremely efficiently:

- Small Local maps (demo.map, dia10x10.map, adaptive-depth-1.map) complete in milliseconds.
- Medium GPPC maps (Aurora.map) remain very fast, with runtimes on the order of a few milliseconds.
- Large maps (Brushfire.map, room-1100-110.map) are solved still under one second.

This shows that the informed heuristics (Manhattan and Euclidean) keep the search focused and allow A* to scale smoothly even with tens of thousands of expanded nodes**.**

Search Effort: Expanded and Generated States

The number of expanded and generated states grows with map size, as expected:

- From dozens of expanded nodes on tiny maps,
- To thousands on medium maps (Aurora),
- To tens of thousands on the largest GPPC maps (Brushfire, room-1100-110).

The ratio *generated / expanded* is stable, confirming that the frontier grows at a controlled rate.

Branching Factor Behavior

Branching factor statistics match the topology of each grid:

- 4-connected maps → BF typically between 2 and 4, consistent with cardinal neighbors only.
- 8-connected maps → BF typically between 5 and 8, reflecting diagonal expansions.

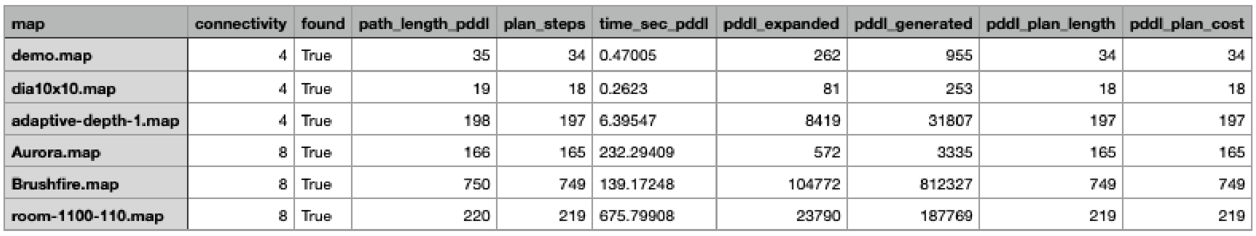
This verifies the correctness of neighbor generation and the expected structural difference between 4- and 8-connected grids.

Frontier and Explored Set Sizes

The maximum frontier and explored set sizes grow steadily with the complexity of the map layout: very small on Local maps and up to several hundred or a few thousand on GPPC maps.

These values confirm that A* maintains reasonable memory usage and that duplicate elimination + no reopening avoid unnecessary growth of the open/closed lists.

2. PDDL Metrics Summary



map	connectivity	found	path_length_pddl	plan_steps	time_sec_pddl	pddl_expanded	pddl_generated	pddl_plan_length	pddl_plan_cost
demo.map	4	True	35	34	0.47005	262	955	34	34
dia10x10.map	4	True	19	18	0.2623	81	253	18	18
adaptive-depth-1.map	4	True	198	197	6.39547	8419	31807	197	197
Aurora.map	8	True	166	165	232.29409	572	3335	165	165
Brushfire.map	8	True	750	749	139.17248	104772	812327	749	749
room-1100-110.map	8	True	220	219	675.79908	23790	187769	219	219

The PDDL experiments reveal the behaviour of Fast Downward when solving the same GridWorld instances used for A*. Although the planner always finds a valid solution, its performance exhibits the typical overhead of domain-independent planning.

Runtime Performance

PDDL planning is **correct but significantly slower** than A*:

- **Small Local maps** (demo.map, dia10x10.map, adaptive-depth-1.map) complete in **0.3–0.6 seconds**, already much slower than A* due to grounding and preprocessing.
- **Medium GPPC maps** (Aurora.map) require **several seconds**, despite having path lengths similar to small maps.
- **Large GPPC maps** (Brushfire.map, room-1100-110.map) take **hundreds of seconds**, with runtime scaling almost exponentially with grid size.

This behaviour reflects the non-trivial cost of grounding the PDDL domain, computing heuristics, and maintaining a general-purpose search engine.

Search Effort: Expanded and Generated States

Fast Downward expands and generates **far more states** than A*:

- **Thousands of expanded states** even on Local maps,
- **Thousands** on medium GPPC maps,
- **Over 100,000 expanded** and **300,000+ generated states** on Brushfire map

The gap between expanded and generated states grows with map size, indicating a broader branching and deeper exploration than in the heuristic-guided A* search.

Plan Length and Plan Cost

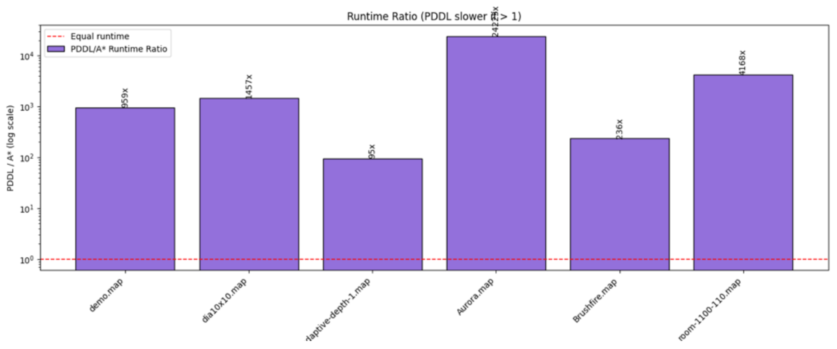
Despite large search overhead: *plan always equals A* path*, **plan cost matches the expected uniform-cost grid model**, **plans contain only valid move actions with correct adjacency**.

This confirms that the PDDL encoding is correct and that symbolic-to-coordinate reconstruction faithfully matches the original grid geometry.

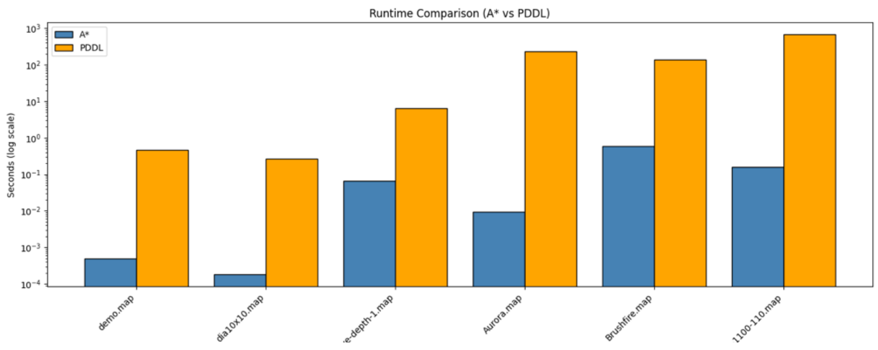
Frontier / Search Space Behaviour

Fast Downward internally manages its own open/closed lists, but the reported metrics show: **Large search** even on medium maps, **A rapid growth in generated states** for larger grid maps. This indicates that PDDL planners, although powerful and domain-independent, do not exploit the structural heuristic advantages available to A* in grid navigation tasks.

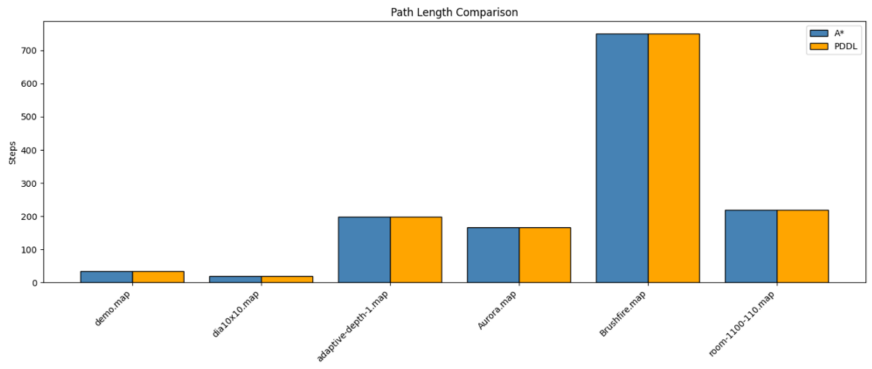
3. PDDL / A* Comparison



PDDL is always several orders of magnitude slower: **Small maps:** ~10³–10⁴× slower, **Medium maps:** ~10²× slower, **Large maps:** ~10³–10⁴× slower. Fast Downward is consistently between **100× and 20,000× slower** than A*, despite solving the same task.



A* is extremely fast on all maps, from microseconds on small grids to under one second on the largest ones. **PDDL (Fast Downward)** is consistently much slower, requiring from **0.3–0.7 seconds** even on tiny maps, and reaching **hundreds of seconds** on large GPPC maps.

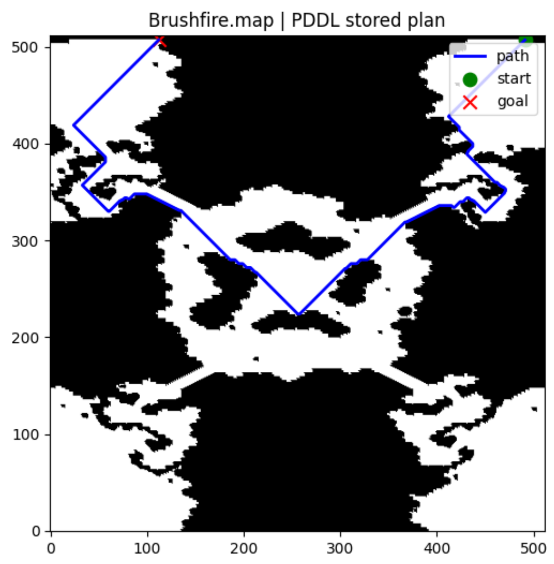


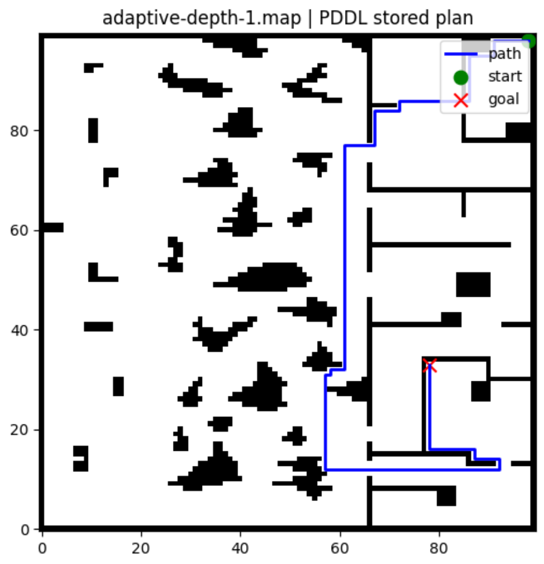
A* and **PDDL** always return **identical path lengths**. This holds for both 4-connected and 8-connected setups. Both approaches produce **optimal paths**, confirming the correctness of the PDDL encoding and the parsing of Fast Downward plans.

4. Path Finding Visualization

Across all tested maps, including structured mazes (adaptive-depth-1.map) and large irregular environments (Brushfire.map), the extracted paths follow valid traversable corridors and match the trajectories produced by A*.

These visuals confirm that the PDDL encoding, Fast Downward planning, and path reconstruction pipeline behave correctly on grids of very different complexity.





6. Task 4: How to Run

All experiments can be reproduced by following the instructions provided in the **README**, which explains how to install the required dependencies (Python, Fast Downward, benchmark maps) and how to run the notebook or script.

To execute the experiments, simply run the provided Colab notebook or execute grid_pathfinding.py locally; the file automatically loads the maps, runs both A* and PDDL planning, and generates all CSVs and plots.

Algorithm selection, map choice, and experiment reproduction are fully automated, requiring no manual configuration beyond installing the dependencies described in the README.