Artificial Intelligence Homework

Solving Ricochet Robots with A* and Automated Planning

Andrea Liviero
Student ID: 2212838
liviero2212838@student.uniroma1.it

January 2026

Abstract

In This report we will presents a comparative analysis of two Artificial Intelligence techniques which were applied to the *Ricochet Robots* puzzle. We implemented a custom environment in Python and solved it using (1) the A* search algorithm with a Manhattan distance heuristic, another A* with a heuristic bfs approach and (2) a reduction to Classical Planning (PDDL) solved via the Fast Downward planner. Experimental results on grid sizes ranging from 5×5 to 10×10 demonstrate that while A* is effective for trivial instances (both the A*), the PDDL planner (using the Landmark-Cut heuristic) offers significantly superior scalability and robustness for this specific domain logic.

Contents

1 Introduction

In this project, we address the *Ricochet Robots* problem, a puzzle game that poses challenges for pathfinding algorithms due to non-local movement mechanics. Unlike standard grid pathfinding where an agent moves to an adjacent cell, a robot in this domain must "slide" continuously until it hits a wall or another robot or a board boundary. They need to reach the goal position, but since they can't stop.

We model this problem as a deterministic, fully observable environment. To solve it, we implement two distinct approaches:

1. **Procedural Search:** A custom implementation of the A* algorithm[cite: 47].
2. **Declarative Planning:** A PDDL domain modeling the "sliding physics," solved using the Fast Downward planner.

The report details the problem modeling, the algorithmic implementations, and an experimental comparison focusing on runtime and node expansion metrics, up to the conclusions.

2 Task 1: The Ricochet Robots Problem

2.1 Problem Description

The game consists of an $N \times N$ grid containing k robots and a set of walls (obstacles) which can be seen as "bouncing" objects that make the robot changing direction. One robot is designated as the "Target" which must reach a specific goal coordinate. The big characteristic is mechanic of the movement: each robots do not move step-by-step, instead it begins moving in a cardinal direction (North, South, East, West), it cannot stop until it is blocked by:

- A static wall.
- Another robot.
- The boundary of the board.

And therefore continues to slide in the field, until it gets to the goal. This is different from other games, such as 15-puzzle, since even if two states are "near" in the search graph, can be opposite in the spatial realm. For example, we may have (0,0) -¿ (0, 10) in a single move of the robot, which is a single state change, but a complete overhaul of the robot position on the game field. Also, since in the field 2 robots are present at any given time, we can say that the field itself is dynamic, since each robot acts as a moving wall to the other. Also, we need to move the other robot in a way that "helps" the other to get to goal.

2.2 State Representation

In SP problems the state representation shall be minimal (containing only necessary dynamic information) but must be complete (distinguishing every unique configuration) We represent the state S as a tuple of coordinates for all robots:

$$S = ((x_0, y_0), (x_1, y_1), \ldots, (x_k, y_k))$$

where index 0 represents the target robot. The walls are stored statically in the environment definition. This representation is hashable, allowing efficient storage in the 'visited' set for search algorithms. We should analyse better why the ricochet problem state is given by all robots position at any given time: since each robot is wall of the other, we need to have the position of those at any given time, while normal obstacles (like wall) are not stored, since they're not needed in the computation. We chosed to use tuple inside the python code due to its immutable and hashable nature. This allows states to be added efficiently to the hash table in the A^* algorithm, enabling $O(1)$ duplicate detection.

2.3 Transition Function

The transition function determines how the system evolves from state S to state S' given an action a. Unlike the 15-Puzzle in which tiles swap or Grid-Pathfinding having an agent stepping, the transition function here is non-local and therefore a single conceptual "move" involves a loop that simulates physics. This is very important due to the nature of the problem, in which the transition function creates a discontinuous search space, thus a robot in position 0, 0 and having goal position 0, 1, may have to go on the other side of the board, hit a wall, or just wander around hoping to bounce on robot B at the right spot. This is why (and we will arrive at this) heuristic solver don't work well! Technically speaking, the transition function 'get_neighbors(state)' generates all reachable states by simulating the slide for every robot in every direction. This requires an iterative check (a "while loop") to determine the final resting position of a sliding robot.

3 Task 2.1: Implementation of A*

3.1 Algorithm Design

We implemented the A* algorithm manually in Python, strictly adhering to the requirement of duplicate elimination and no reopening (Graph Search)[cite: 58].

- **Open List:** A min-priority queue ('heapq') storing nodes ordered by $f(n) = g(n) + h(n)$.
- **Closed Set:** A Python 'set' storing the hash of visited states to prevent cycles and redundant expansions.

Later, we implement a second heuristic function, which you'll find in the code (as of now is commented) having less boundaries (no more a manhattan distance therefore) as a way to demonstrate that this approach is not useful (or at least not As useful) as other. Both are in the code, and shall be uncommented / activated as liked. It is a simplified version, but still shows and support the finding found before.

3.2 Heuristic Function

For the heuristic $h(n)$, we employed the Manhattan Distance of the target robot to the goal:

$$h(n) = |x_{target} - x_{goal}| + |y_{target} - y_{goal}|$$

This heuristic is admissible because a robot must travel at least the Manhattan distance to reach the goal. However, it is a "weak" heuristic for this domain because it ignores the sliding constraints (e.g., a robot might be geometrically close to the goal but forced to slide away from it due to a lack of walls). We later decided to implement a different approach, although it changes the problem partially. Since the manhattan distance doesn't care about walls or other things, and treats the board like a piece of paper, it just thinks that can go to the goal based on the distance, than if you can actually go in that direction is less of a problem to it. The BFS heuristic is much more realistic and eliminates thousands of "fake" paths that A* would otherwise explore, because it knows that being geometrically close doesn't mean you can actually stop there. In the conclusion section, we will discuss the results and reasons beside.

4 Task 2.2: Planning with PDDL

We chose **Classical Planning** as the second technique. We modeled the domain in PDDL and generated problem files dynamically using a Python script. PDDL works differently from

standard python, where you write "how" to solve a problem, instead you describe the problem and give it to a solver (in our case fast downward) which figures out how to solve it. It's made of the domain, where the physics of the universe (the walls stop the robot) are described, and the problem with the specific snapshot.

4.1 Domain Modeling Strategy

Standard PDDL actions are discrete and cannot natively represent a "while loop" (slide until blocked). To overcome this, we modeled the movement as a state machine with micro-steps therefore a robot can be in two modes: 'idle' or 'sliding' (but can't turn or stop voluntarly, since it's locked by the physic). The domain created some problems at the beginning, since it was deemed "too relaxed" from the solver, which created errors. It was later fixed with a much stricter domain, which allowed at the game to work properly.

The key Actions were:

- `start-slide`: Transitions a robot from 'idle' to 'sliding' in a direction.
- `move-slide`: Moves a sliding robot one cell forward if the next cell is free.
- `stop-slide-wall/border/robot`: Transitions a robot from 'sliding' to 'idle' if the next cell is blocked.

4.2 Handling Boundaries

A significant challenge encountered was handling grid boundaries, which were the afromentioned problems before in the domain. Trying to simply add walls at the edges wasn't working, and the planner failed because "falling off the map" was not a valid state. We resolved this by explicitly marking grid edges with a '(boundary ?cell ?direction)' predicate and adding a dedicated stop action:

```
(:action stop-slide-border
  :parameters (?r - robot ?current - cell ?d - direction)
  :precondition (and
      (sliding ?r ?d)
      (at ?r ?current)
      (boundary ?current ?d) ; Robot is at the edge
  )
  :effect (and
      (not (sliding ?r ?d))
      (idle ?r)
  )
)
```

Listing 1: PDDL Action for stopping at boundaries

4.3 The Solver

We used the Fast Downward planner, deemed to be the easiest to configure in our environment and machine (An M2 Macbook) It was downloaded diretcly in the folder via homebrew and shall be done by whoever uses it. Specifically, we used the 'seq-opt-lmcut' configuration (Landmark-Cut heuristic), which guarantees optimality and provides strong guidance for pathfinding tasks. The PDDL works differently from A*, since it tries to subdivide the problems and for example already knows it will have to move an helper robot in a given configuration to get the main one to objective. Differently from A* and other simple solver which work in a "dumb" way, looking only for distance, PDDL actively look for subproblems and fixes them to get the bigger one done. Therefore in our case it won't only use the helper robot, but will try to put it in a way to get it to help the other in goal position. The visualiser can help understanding this:

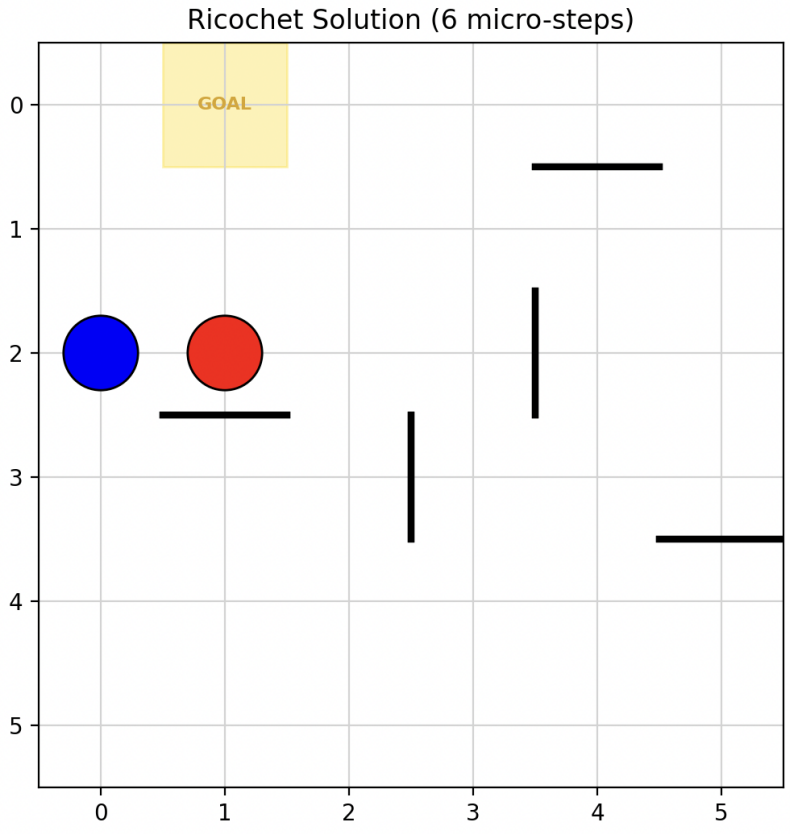

Figure 1: Screenshot of the visuliser (the animation can be run diretcly from running visualiser.py, with a random map.

5 Task 3: Experimental Results

We compared both algorithms on random grid instances of increasing size ($N = 5$ to $N = 10$). We measured execution time and the number of expanded nodes. We decided to create also a csv file for later creating graphs, and tried to create a visuliser with python. This created a lot of problems, since in python we can't easily create animation from csv and it was decided to create a fake map, with a fake solution, which does not change, but its useful only for understanding the problem. It would probably be doable a visuliser like that having more time and using a different language, and may be explored in the future. *All graphs are from the same execution.* We start with time analysis:

5.1 Runtime Comparison

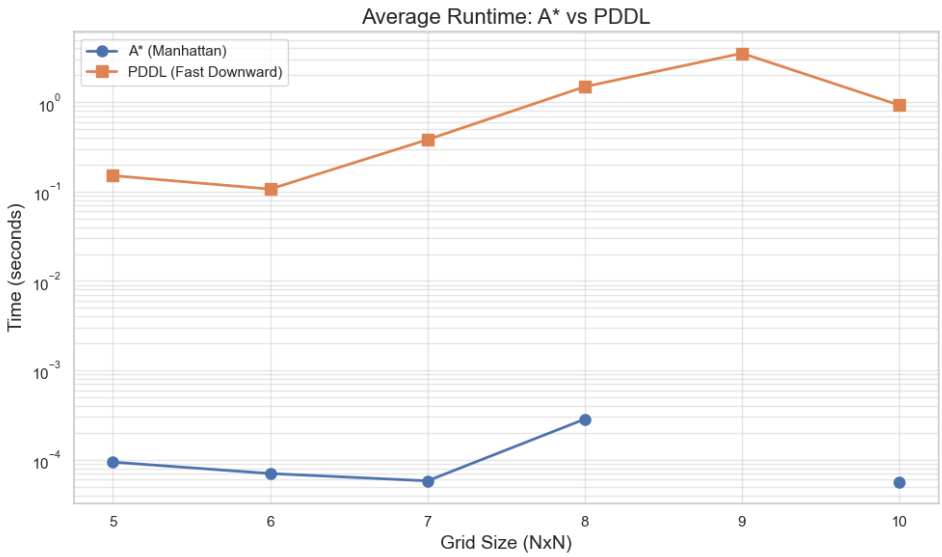

Figure 2: Runtime comparison: A* (Manhattan) vs PDDL

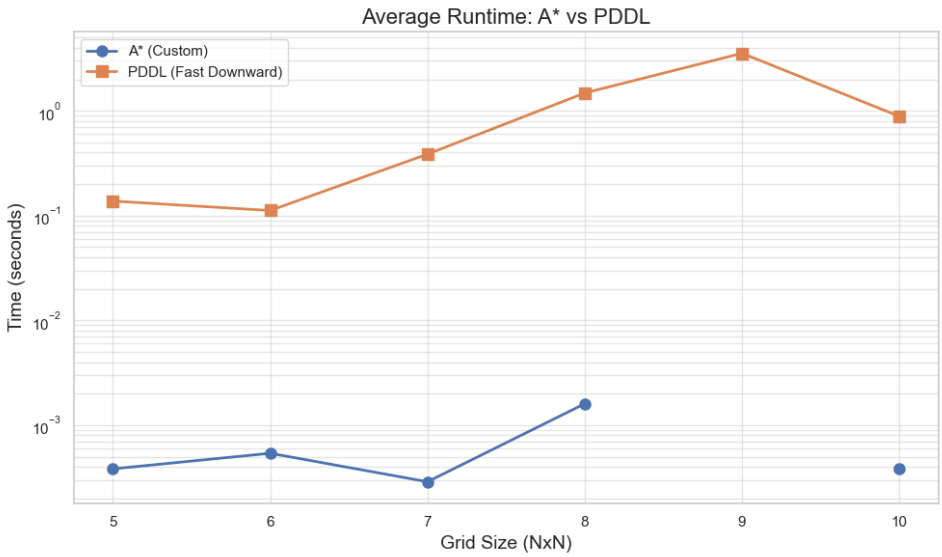

Figure 3: Runtime comparison: A*(BFS) vs PDDL

The experiments revealed a distinct performance gap.

- **A* Performance:** On small or trivial grids (5×5), A* was instantaneous ($< 0.001s$). However, as complexity increased, A* frequently timed out. This is due to the weakness of the Manhattan heuristic, causing the algorithm to degenerate into a blind search. Also the second heuristic doesn't show big improvements, almost being the same, although on average it would perform *slightly* better on time comparison (we remember that we're speaking of miniscule differences, not visible on the graph).
- **PDDL Performance:** The planner was remarkably consistent, solving complex 10×10 instances in under 2 seconds. The overhead of starting the planner makes it slower for trivial tasks, but its derived heuristics ('lmcut') make it much more robust for hard

instances. We can observe the small increase at 9*9 which then decreases again at 10*10. This can be explained by the fact that the random generator creates 3 different problems, of which one can be trivial (which means can be solved with a single slide). This is casual, and can be supported by the fact that the A* was able to solve the problem but timed out in the other 2 problems (not trivial), supporting this thesis. This usually won't happen, but we decided to include this specific graph to explain this phenomena.

5.2 Node Expansions

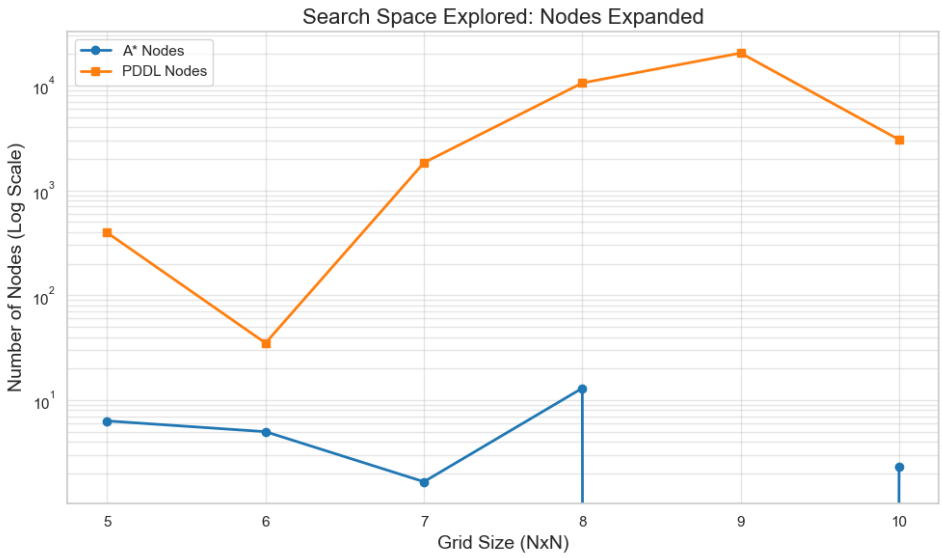

Figure 4: Nodes Expanded: A* (Manhattan) vs PDDL

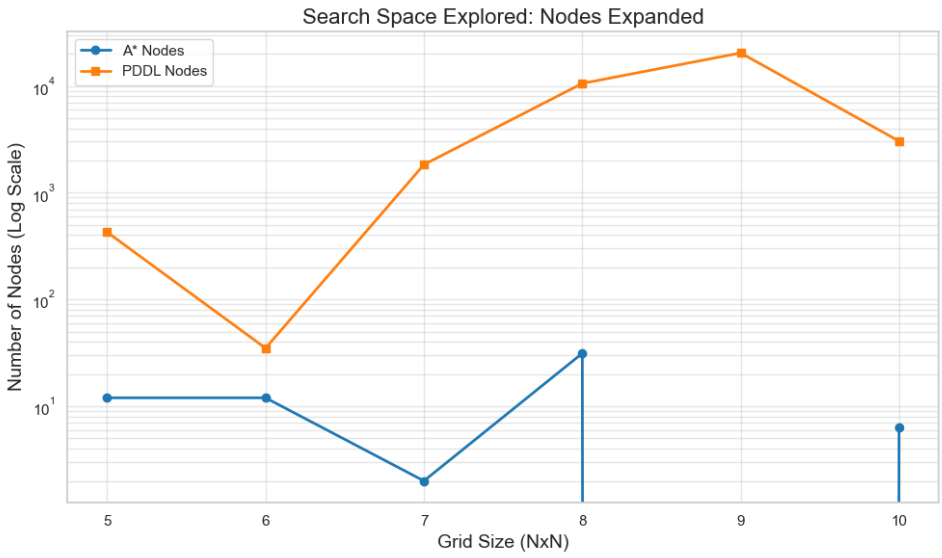

Figure 5: Nodes Expanded: A* (BFS) vs PDDL

The node expansion metrics confirm the heuristic disparity. A* expanded orders of magnitude more nodes than PDDL on non-trivial problems and as mazes get harder, the A* gets less and less effective, until timing out. The PDDL planner's ability to automatically infer landmarks (sub-goals) allowed it to prune the search space effectively, whereas A* wasted effort exploring

dead-ends. No meaningful differences can be observed from Manhattan and BFS, and the dot at maze 10 still indicate the presence of a trivial problem, which can be seen by the minuscule amount of nodes.

6 Conclusion

This project highlighted the trade-off between custom procedural implementations and generic solvers. While our custom A* was easy to implement, it struggled without a domain-specific heuristic, and even when a second BFS based one was implemented, even on a simpler problem, still failed to bring meaningful results on maze not trivial or bigger. In contrast, the generic PDDL planner, despite the overhead of file generation and parsing, proved to be a superior tool for this problem due to its advanced internal heuristics, demonstrating its strong logic and capabilities on complex tasks. It is therefore obvious which solver was the best, but the reason is not because A* is not good, it usually work good enough on small pathfinding. But it assumes that the best path is always the one with the smallest distance, while in this case as we explained before, being close spatially is not a synonymous of being close to the solution. Having a better Heuristic (like PDDL) in ricochet allows you to play with the second robot and with the elemtents in the maze, allowing you faster solution or even one at all, while A* would get to one (at some point) but in much more time statistically. And PDDL is not the best possible solution, a custom one would probably work better, but it's better than SAT/CSP since they struggle on sequential plans of unknown length and would perform really bad. PDDl is built for sequential decision making, so it can do handle better this kind of problems. Obviously a C++ solver with pattern database could run better, also mainly for not having to do the parse text files ecc.. but it would be much more complex.

7 Bibliography

This code was made thanks to the use of generative AI (for part of the code). For the theory and report, Samuel Masseport, Tom Davot : Ricochet Robots with Infinite Horizontal Board is Turing-complete was used, available at (*https* : *//www.researchgate.net/publication/*372388944)