Automated Planning Project Report

This project focuses on the design and implementation of an automated planning approach for a warehouse domain. A robot (named Soko) operates on a grid-based warehouse containing walls, boxes, keys, doors and designated drop zones. The objective is to generate a valid plan which makes Soko move each box to its corresponding drop zones while respecting certain constraints such as doors being locked and a limited carrying capacity. The main objectives of this project were:

1. Design an effective state representation and successor function.
2. Implement multiple classical and heuristic-based search algorithms.
3. Design and evaluate heuristics suitable for this domain.
4. Encode the same domain inside PDDL and compare performance.

Search algorithms such as Breadth-First Search, Greedy Best-First Search, A* Search and Enforced Hill Climbing were used as search strategies to help Soko navigate the warehouse. For each search algorithm (except for breadth-first search), two heuristics were applied, namely the Manhattan and Euclidean distances. After the algorithms and heuristics were implemented, a domain-independent planner was also evaluated using PDDL.

Implementation

Parsing and Storing

We first started the project by retrieving our mazes and parsing each one of them. Each maze is encoded as a CSV file, where each cell represents an object or empty space in the grid. Most of these objects are static with Soko being the only dynamic object being able to move, however it only moves based on the search algorithm and heuristic to it. The maze is represented using a “Maze” data structure containing:

- The grid dimensions
- A set of wall coordinates
- A mapping of box identifiers to drop zone positions
- A mapping of door identifiers to their positions

This representation is static and does not change during search. All dynamic information is stored exclusively in the state representation. Separating static and dynamic components simplifies reasoning, reduces memory overhead, and avoids duplication across states.

State Representation

We then considered state representation, which is one of the most critical components of the planning system. Each state captures a complete snapshot of all relevant dynamic information required to determine action applicability and goal satisfaction. In this implementation, each state contains:

- The robot’s current position on the grid.
- The identifier of the box currently being carried, if any.
- The positions of all boxes that are not being carried.
- The set of keys currently owned by the robot.
- The positions of keys still present on the floor.
- The path cost, represented by the number of actions taken to reach the state.

States are implemented as immutable objects. This immutability is essential for safe reuse across search algorithms and enables reliable hashing and equality checking. To support duplicate detection, a state key is derived from all logically relevant components of the state. Two states are considered identical if and only if all these components match exactly. This representation ensures correctness and efficiency across all search strategies while remaining expressive enough to model the full complexity of the domain.

Successor Function

The successor function was then implemented and is responsible for generating all valid successor states from a given state. Each successor corresponds to the application of a single action from the action set defined as part of the domain model. These actions include movement actions, key pickup actions, door traversal actions, box lifting actions, and box dropping actions. Using the successor function, Soko is able to perform the necessary tasks to achieve the goal state.

Each action is associated with important preconditions that must be satisfied in the current state. For example, movement actions require that the target cell is within the maze bounds and is not blocked by a wall or a locked door. Door traversal actions additionally require that the robot owns the corresponding key. Box lifting actions require that the robot is in the same cell as a box and is not already carrying another box. Boxes can only be placed in their respective drop zones, otherwise the goal state’s prerequisites are not satisfied.

When an action is applicable, its effects are applied to generate a new state. This includes updating the robot’s position, modifying box or key locations, and incrementing the path cost. Each successor state is returned along with a symbolic action label, which is later used to reconstruct the final plan and extract key action sequences. With every new search algorithm or heuristic function used, the

successor function is always updated to align with each search algorithm and heuristic applied.

Goal Test

We then implemented the goal test which checks whether a given state satisfies the goal state's requirements. In this domain, the goal state is achieved when all boxes are placed in their corresponding drop zones and the robot is not carrying any box.

The implementation checks each box's position against the predefined mapping of drop zones stored in the maze representation. This approach ensures that partial or incorrect placements are rejected. By enforcing that the robot is not carrying a box in the final state, the goal test ensures that the final state fully satisfies the planning objective.

Search Algorithms

To explore different strategies of solving the maze problems, several search algorithms were implemented and compared. These algorithms differ in how they explore the maze and their efficiency in finding optimal or near-optimal paths. Some methods systematically explore all possible paths, while others use guidance from heuristics to focus on more promising directions. Breadth-First Search serves as an uninformed search baseline, as it always finds the shortest solution, however, it can be slow for larger mazes. Greedy Best-First Search, A* Search, and Enforced Hill Climbing aim to improve performance by using heuristic information. Together, these approaches allow for a clear comparison of solution quality and efficiency across different maze scenarios.

1. Breadth-First Search

Breadth-First Search is an uninformed search strategy that explores the state space in increasing order of depth. It uses a queue as its frontier and guarantees that the first solution found is optimal with respect to the number of actions. Despite its limitations, Breadth-First Search later proves to be an important baseline for comparison with other search strategies.

2. **Greedy Best-First Search**

Greedy Best-First Search prioritises states based solely on a heuristic estimate of their distance to the goal. The frontier is implemented as a priority queue ordered by the heuristic value. This strategy is generally faster than Breadth-First Search because it focuses exploration on promising regions of the state space. However, it does not guarantee optimality and may fail to find a solution in some cases if the heuristic misleads the search into unproductive regions of the state space.

3. **A* Search**

A* Search combines the strengths of Breadth-First Search and heuristic-guided search by using an evaluation function that sums the path cost and heuristic estimate. This approach balances exploration and exploitation and guarantees optimal solutions when the heuristic is admissible. When paired with a strong heuristic, A* usually provides significant performance benefits compared to others of its style.

4. **Enforced Hill Climbing (EHC)**

Enforced Hill Climbing is a local search technique that iteratively improves the heuristic value of the current state. When no immediate improvement is available, EHC performs a breadth-first search from the current state until it finds a successor with a strictly better heuristic value. Overall, EHC is highly efficient and conceptually straightforward, but is sensitive to heuristic plateaus and local minima.

Heuristic Functions

In addition to the choice of search strategy, the performance of the planner is strongly influenced by the heuristic used to guide the search. Heuristics provide an estimate of how close a given state is to the goal, enabling the search algorithms to prioritize goal-directed paths. In this project, two distance-based heuristics were explored — Manhattan and Euclidean distances. Both heuristics measure how far boxes are from

their target drop zones, but they differ in how accurately they reflect movement within a grid environment. Putting the box in its corresponding zone is the priority when soko is holding a box, else the smallest distance to a box is found and that cost is added to the total cost. For each key that is on the floor a small penalty is added to encourage picking up keys. The differences between these heuristics are the distance functions. By comparing these heuristics, this comparison highlights how heuristic choice impacts search efficiency, solution quality, and overall performance across the different algorithms.

1. Manhattan Distance Heuristic

The Manhattan distance heuristic computes the sum of the grid-based distances between each box and its respective drop zone. This heuristic is informative because it provides a meaningful lower-bound estimate of the effort required to place each box correctly, therefore guiding the search algorithm toward states where boxes are closer to their target locations. It captures the important structure of the problem without additionally considering irrelevant details, such as Soko's exact path or the order in which the boxes are moved. The heuristic is also always admissible because it ignores obstacles, locked doors, box carrying constraints and key dependencies, all of which contribute to the true cost of reaching the goal. Since the Manhattan distance assumes the most direct path in a grid with no obstructions, it will never overestimate the actual number of actions required, this ensures the maximum outcome when used with certain search algorithms such as A*.

2. Euclidean Distance Heuristic

The Euclidean distance heuristic computes straight-line distances between boxes and their drop zones. This heuristic is also informative, as it provides smooth and continuous guidance toward the main goal and often prioritises states that are closer to the goal state. In practice, this can significantly reduce the number of expanded states in greedy or local search strategies. However, the Euclidean distance heuristic is not always admissible in this domain. Since the environment is discrete and movement is restricted to only horizontal or vertical steps, more direct distances in a

diagonal straight-line distances can overestimate the true cost of reaching the goal, especially in the presence of obstacles, walls, or locked doors.. As a result, the heuristic may assign a value that is lower than the minimum number of actions required, not complying with the admissibility condition and potentially leading to suboptimal solutions when used with search algorithms such as Hill Climb.

PDDL Encoding

The domain was encoded in PDDL to allow comparison with a domain independent planner. The PDDL domain defines object types for agents, tiles interactables at a high level, then interactable is further defined for types box, key, zone and door. Furthermore, predicates were used to define relationships between these objects, namely:

- agent and interactable positions
- paths between tiles
- matching interactables between each other (keys to doors, boxes to zones)
- doors being locked or not (for ease of use by the actions)

Then all the actions that Soko can perform were implemented with their corresponding preconditions and effects. These include move, move through a door, pick up key, lift box, and drop box actions. This complete domain file served as the basis that the problem generation was built on top of and provided a clear structure for problem syntax and predicate usage.

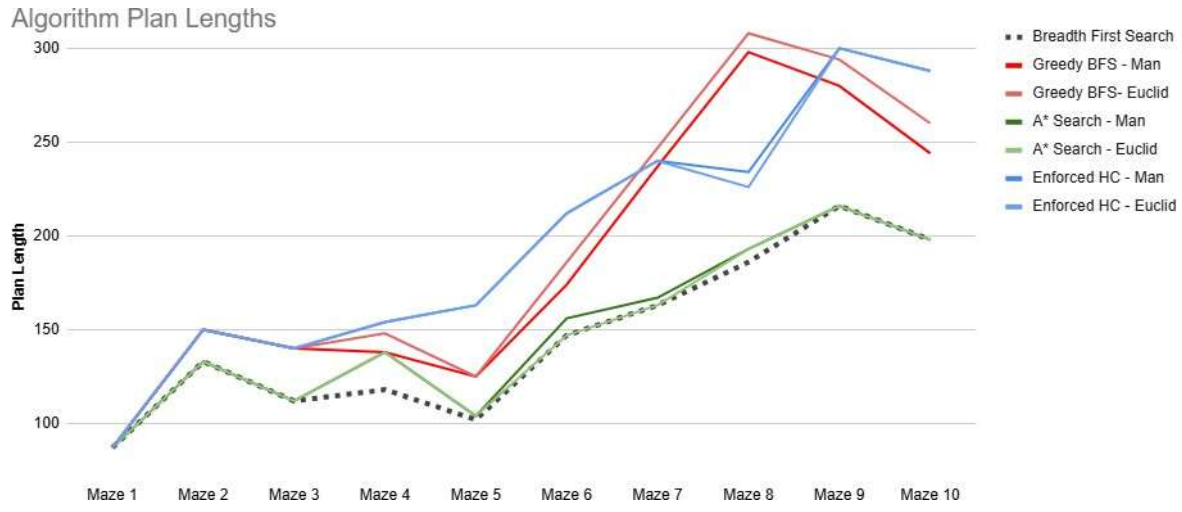
After modelling our domain in a domain.pddl file, we proceeded to implement a set of functions which encode the problem into PDDL syntax. We used the same parser used for the python solvers, and then encoded each object (grid cells, doors, zones, boxes, and keys), one by one into the problem file. The problem file also initialises relationships such as paths between grid cells and the relationship between a door and its respective key. Finally, it defines the goals which must be met to reach the solution, in this case being that the boxes are placed on their respective zones.

Evaluation

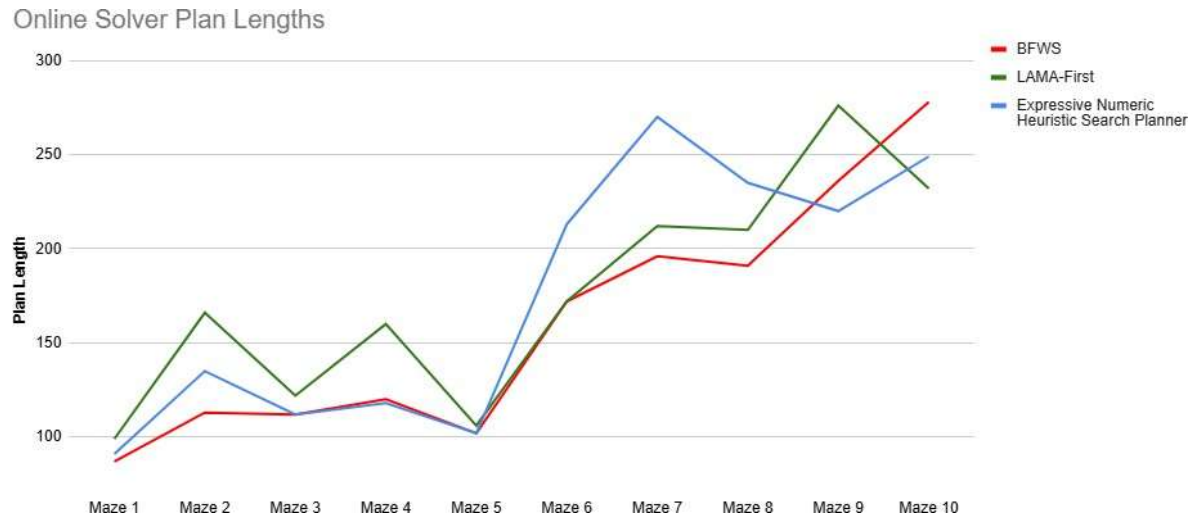
Search Algorithms

Across all experiments, the results highlight many trade-offs between solution quality, computational cost, and reliance on heuristic guidance among the different search algorithms. On one hand, optimal algorithms such as Breadth-First Search and A* Search, generally required more computation than purely heuristic-driven approaches, but they also provided more reliable and consistent outcomes. On the other hand, strategies that prioritized speed through aggressive heuristic use significantly reduced the number of expanded states, yet usually sacrificed optimality in more complex environments.

Breadth-First Search served as a useful baseline for comparison, consistently finding optimal solutions by exhaustively exploring the search space level by level. However, its lack of heuristic guidance resulted in increased plan length in explored states compared to other algorithms as maze complexity increased. This made it impractical for larger or more intricate mazes, emphasizing that while optimality is guaranteed, scalability is a major limitation. Similarly, BFWS (Best-First Width Search) and A* Search maintained a similar pattern — optimising plan complexity with the cost of an uneven time distribution.

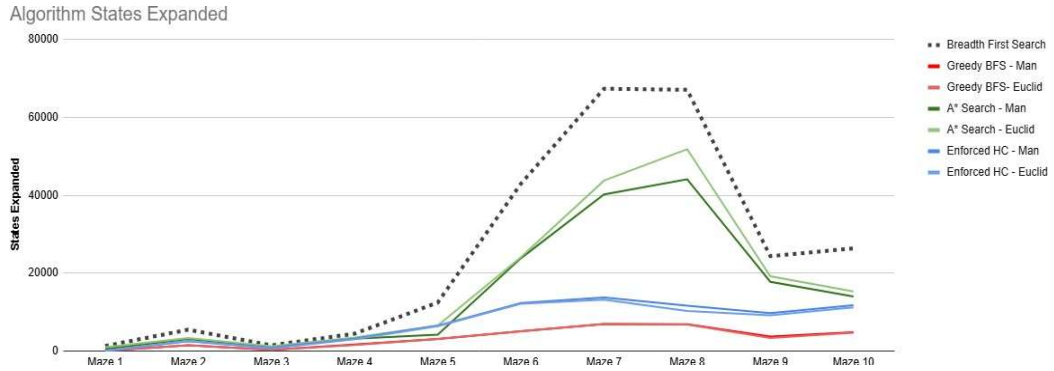


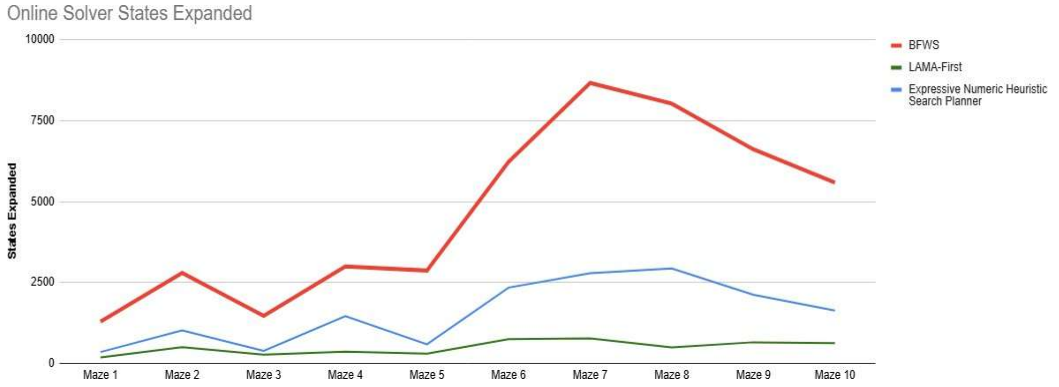
Comparing the *online* algorithms by plan length proved to be very different. Even though LAMA-First and Expressive Numeric Heuristic Search Planner are uneven at certain points, it is far too difficult to pinpoint extremities, exhibit variability. However, one *can* say that since BFWS (almost) consistently created the shortest plans, it would be the optimal planner in this set of experiments.



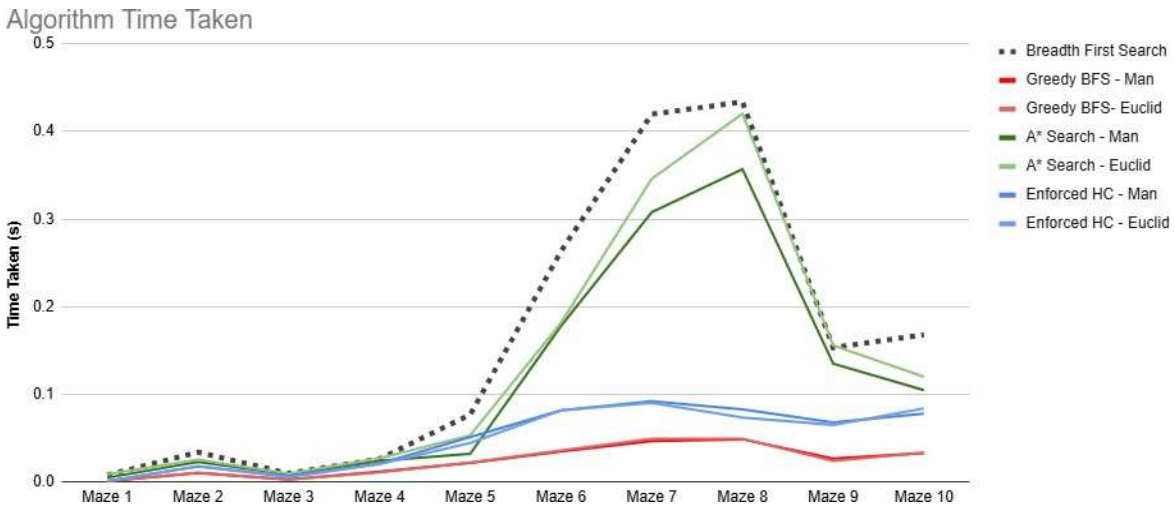
Heuristic-driven approaches demonstrated clear efficiency gains over uninformed search. Greedy Best-First Search and Enforced Hill Climbing significantly reduced the number of expanded states by aggressively following heuristic estimates toward the goal, while LAMA-First likewise outperformed its online counterparts in this regard. This trend also held consistently regardless of the chosen heuristic. In simpler mazes, this led to very fast solutions, often outperforming more exhaustive methods.

However, this efficiency came at the cost of reliability. Both locally-run algorithms often produced unnecessarily longer plans, and occasionally even failed to generate a valid plan altogether in EHC's case. Interestingly, LAMA-First managed to maintain a far more linear number of states expanded between mazes, while ENHSP and especially BFWS present very significant overhead in terms of states expanded around mazes 7 and 8. These results show that heavy reliance on heuristics can improve speed but can consequently introduce significant risk as problem complexity rises.

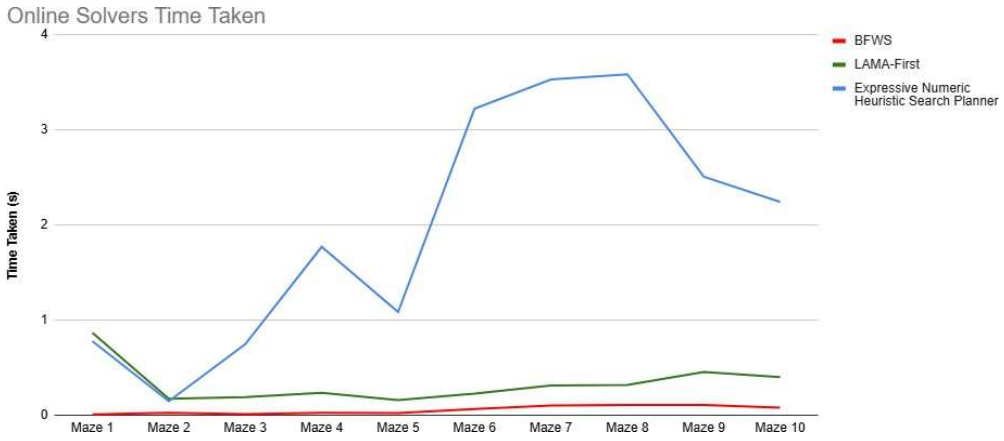




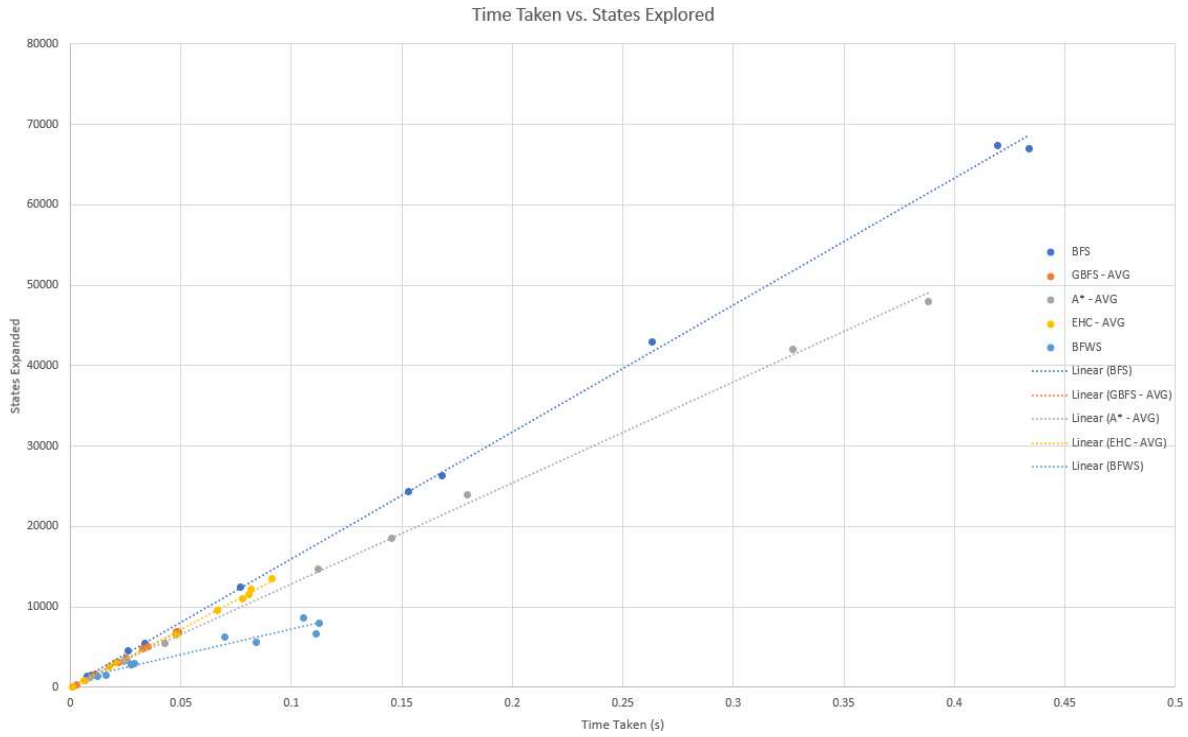
When it comes to the local search algorithms, we can see a strikingly identical pattern in terms of time complexity, where Greedy Best-First Search and Enforced Hill Climbing maintain a very linear and stable pattern regardless of problem complexity. This implies that for rapid local solutions, Greedy Best-First Search would be the obvious and preferred choice. This is because it is reasonably swift and was far more reliable than EHC during testing.



As comparing execution time across different hardware is unreliable, online planners are visualised separately. In doing so, we can compare the online planners with each other while maintaining clear separability. BFWS produced short plans quickly, although it consistently expanded more states. Additionally observe that while ENHSP generally generates as many states as LAMA-First, it has a highly unstable time complexity. If short on time, ENHSP must be avoided altogether.



The following graph illustrates the linear relationship between the time taken to generate a plan and the number of states expanded when generating such a plan. LAMA-First and ENHSP are omitted, as including them obscures trends in the other algorithms. However they still follow the linear trend visible in the other algorithms.

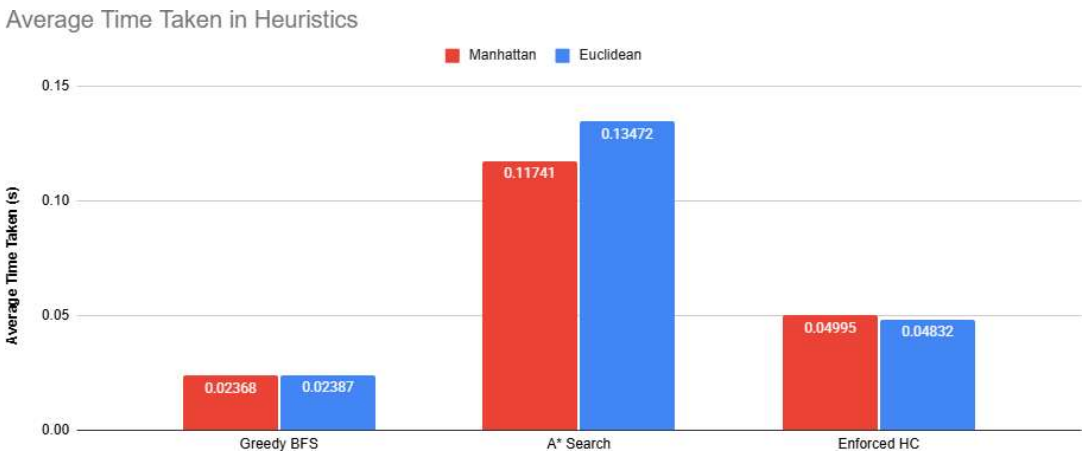


A* Search generally presented an effective solution balancing these competing concerns. By combining path cost with heuristic estimates, it achieved strong efficiency gains over Breadth-First Search while still maintaining optimality when using the Manhattan heuristic. Although the Euclidean heuristic still presents a viable solution, it should clearly be used as an alternative in lower-complexity scenarios. At higher complexities it introduced occasional suboptimal solutions, effectively hindering holistic performance. Overall, A* Search offers a robust compromise between computational cost, optimality, reliability and scalability.

Taken together, no single strategy is universally superior; instead, each excels in different contexts. Breadth-First Search offers reliability at high cost, other heuristic-driven methods offer speed with variable amounts of risk, and hybrid approaches like A* Search provides a practical middle ground. These findings highlight the importance of choosing appropriate strategies for complex planning problems. Furthermore, the observed results align with our expected behavior.

Heuristics (Local)

Unlike the choice of search algorithms, choosing the heuristic is a much more trivial task. In almost every experiment involving A* Search, the Manhattan heuristic was just better, being on average 15% faster than the Euclidean heuristic. Collected data from each instance (of A* Search only) shows that the Manhattan heuristic also consistently generated and expanded fewer states. *Outside* A* Search, the difference in time is negligible, however, the Euclidean heuristic tended to generate and expand less states across instances. This means that unless using A* Search, the Euclidean heuristic, despite occasional inadmissibility, may be preferable in non-A* scenarios.



Conclusion

This project demonstrates the effectiveness of classical planning techniques when combined with carefully designed state representations and heuristics tailored to the given domain. The domain-specific solvers outperformed the generic online planners in most cases, especially when using A* Search with the Manhattan heuristic, which balanced efficiency, optimality, reliability and scalability, effectively.

The project also shows challenges and comparisons of search algorithms and the importance of heuristic quality. Overall, the project highlights that careful alignment between algorithm and heuristic is crucial, and that no single approach is universally superior. These findings provide practical insights for designing efficient and reliable automated planning systems in complex domains.

Appendix

Below are citations and generative AI prompts/conversations which we used throughout this project and the responses we got from each one of them. All of these prompts were on chatGPT and Claude sonnet 4.5 via GitHub copilot.

Citations

[1] GeeksforGeeks, “Breadth First Search (BFS) for Artificial Intelligence,” Available: https://www.geeksforgeeks.org/artificial-intelligence/breadth-first-search-bfs-for-artificial-intelligence/.

[2] GeeksforGeeks, “Greedy Best First Search Algorithm,” Available: https://www.geeksforgeeks.org/dsa/greedy-best-first-search-algorithm/.

[3] GeeksforGeeks, “A* Search Algorithm,” Available: https://www.geeksforgeeks.org/dsa/a-search-algorithm/.

[4] GeeksforGeeks, “Hill Climbing in Artificial Intelligence,” Available: https://www.geeksforgeeks.org/artificial-intelligence/introduction-hill-climbing-artificial-intelligence/.

[5] GeeksforGeeks, “Manhattan Distance,” Available: https://www.geeksforgeeks.org/data-science/manhattan-distance/#:~:text=Manhattan%20Distance%2C%20also%20known%20as,absolute%20differences%20of%20their%20coordinates.

[6] GeeksforGeeks, “Euclidean Distance: Formula, Derivation & Solved Examples,” Available: https://www.geeksforgeeks.org/maths/euclidean-distance/.

GenerativeAI Prompts:

Here you can find the links to the chatGPT conversations we had when building this project:

Conversation 1: https://chatgpt.com/share/693c1314-6c68-8002-924f-b05b610d2378

Conversation 2:
https://chatgpt.com/share/69529386-be44-8003-a1a5-4089a9804342

In the first conversation, generative AI was used to get help on how to properly structure the data and state representation for the project, so that the search algorithms can efficiently operate on the maze and robot actions.

For the second conversation, we needed help improving parts of the report to make the language more precise, consistent, and academically strong. The AI reviewed each paragraph, identified issues (grammatical and factual), and suggested targeted wording improvements without rewriting entire paragraphs.

PDDL Solvers:

Link: https://editor.planning.domains/