Sokoban Solver Analysis Notes

April 27, 2025

1 Beam Search

- I started by implementing the Beam Search algorithm because my first thought was that I should start to build my intuition about the dynamic of the Sokoban game and implicitly the way I should design my heuristics around an offline search algorithm.

1.1 Euclidean Distance Heuristic

- I started off by designing the simplest heuristic one could think of for a 2D game: the euclidean distance. One main difference from this semester's labwork, especially the A* search lab (the labyrinth part) was that, for a given box, its final position is not fixed, that is, we have multiple target states for a box. This means that we would have a final state domain of size equal to the cardinality of the cartesian product of the set of boxes and the set of target positions i.e. N^2, where N is the number of boxes.
- The first critical thinking point was to understand how I should interpret this final state domain. One idea would be to assign a given box to its closest target position where the euclidean distance is the smallest, but, by following this line of thought I would have ended up with a non-admissible heuristic, considering the following scenario which I observed after glancing at the maps: `super_hard_map1` and `hard_map1`.:
  - For a given box, I would have assigned it to its closest target position, but, this may not always be the best, for example, in `hard_map1` this would have led to a deadlock where I would encounter 2 boxes on a border column in which there is only one target position. As pull moves are not desirable, this will be an irreversible move.
- This lead me to the conclusion I need to add some kind of penalty for moves that lead to deadlocks. I thought it would be a good idea if this penalty would just give zero-tolerance to those moves, thus just removing them from the search space i.e. an infinite cost.
- After momentarily taking care of those assignments that lead to deadlocks, the main problem still persisted: How to manage the final state domain? After researching a bit, I found that I could consider the final state domain as a bipartite graph where the boxes are the left nodes and the target positions are the right nodes. The edges of this graph are the euclidean distances between the boxes and the target positions. The goal is to find the minimum cost perfect matching of this bipartite graph. This is a well known problem in combinatorial optimization known as the assignment problem. I used the `scipy` library that uses an $O(N^3)$ algorithm which is pretty insignificant in computational terms for the sizes of the maps I was dealing with (a maximum of 5 boxes) therefore I didn't bother to optimize this further.
- The final heuristic value is the sum of the weights of the edges in the minimum cost perfect matching of the bipartite graph.
- I was pretty optimistic about this method of min-weight perfect matching so I introduced 0 tolerance for pull moves, disabling them completely from the search space.

- After running the algorithm with this heuristic, and analysing the runtimes and states explored for each kind of map, the results were as expected:
  - For the easy maps, where the search space is small, the number of states explored and the runtime was proportional to the map size.
  - For the medium maps, with a slightly larger search space, the number of states explored still remained proportional to the map size as we can see an average **2.5x increase factor** but the runtime remained similar to the easy maps.
  - For the hard maps and the super hard map, the runtime still remained similar to the previous maps i.e. the 0.4s range, and surprisingly so did the number of states explored. This was a bit unexpected as I thought that the number of states explored would be larger since these maps require more complex tactics to solve them.
  - The last maps which I was most curious about, the *large* maps showcased this heuristic's true downfall. While for the first large map which is similar in size to one of the medium maps and has a total of 2 boxes the algo reaches a solution with a number of explored states and runtime similar to the medium maps, the second large map which has a significantly larger search space and a higher number of boxes (3) which are also placed in a more complex manner (the closest target position is not always the best one), the algorithm fails to find a solution.

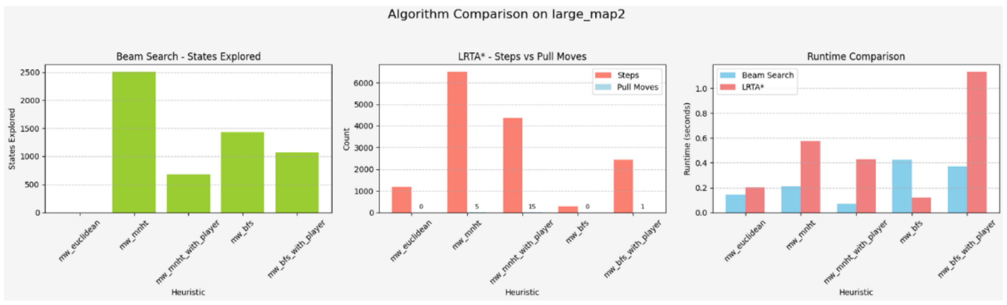

Figure 1: **Beam Search Results for large_map_2**

- The best explanation I could come up with for this failure was the heuristic's quality. Because Beam Search is a greedy algorithm, it will always try to expand the most promising nodes first. This means that if the heuristic is not good enough, it will get stuck in a local minimum and will not be able to find the optimal solution as was the case for `large_map2`.

- Next step was finding a better heuristic for calculating the distances between the boxes and the targets while maintaining the min_weight perfect matching.
- Next candidate would be the Manhattan distance.

1.2 Manhattan Distance Heuristic

- The Manhattan distance is calculated as the sum of the absolute differences between the x and y coordinates of two points i.e. the total number of moves needed to reach a target position from a given box position.

After analysing the results, these were the main conclusions:

- As expected, the Manhattan distance heuristic is a better heuristic than the Euclidean distance heuristic as it takes into account the fact that the boxes can only move in 4 directions (up, down, left, right) and not diagonally.
- While the number of states explored remained similar to the Euclidean distance, just a bit lower, we can see a decrease in the runtime for all maps, averaging around a **0.5x factor decrease** to the Euclidean distance heuristic.
- Furthermore, now, the algorithm was able to find a solution for the `large_map2` which was previously unsolvable, indicating that the Manhattan distance avoids local minima better. Again, one conclusion is that this heuristic is better suited for the Sokoban game as it doesn't take into account diagonal moves.

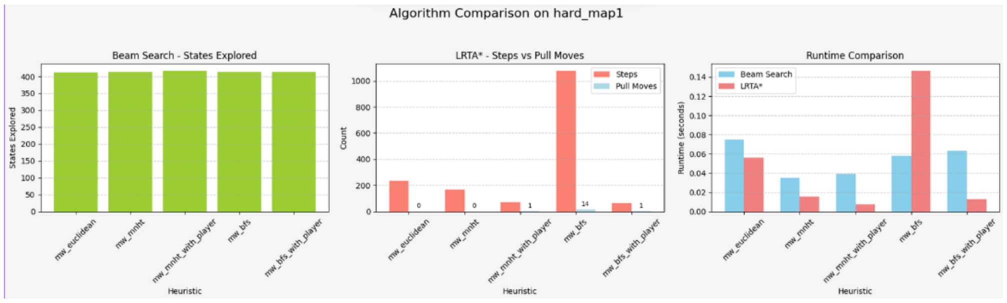

Figure 2: **hard_map_1**

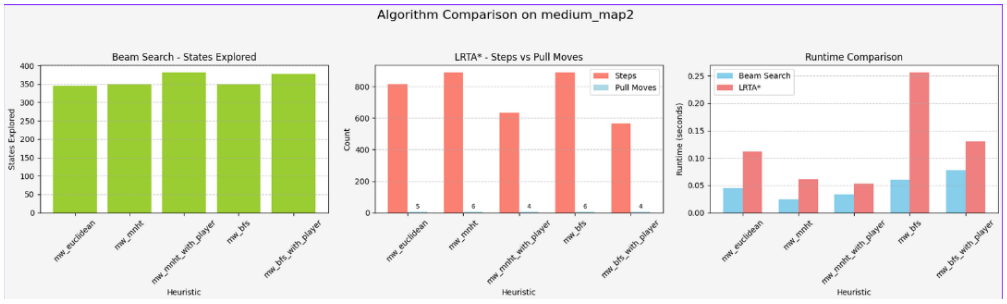

Figure 3: **medium_map_2**

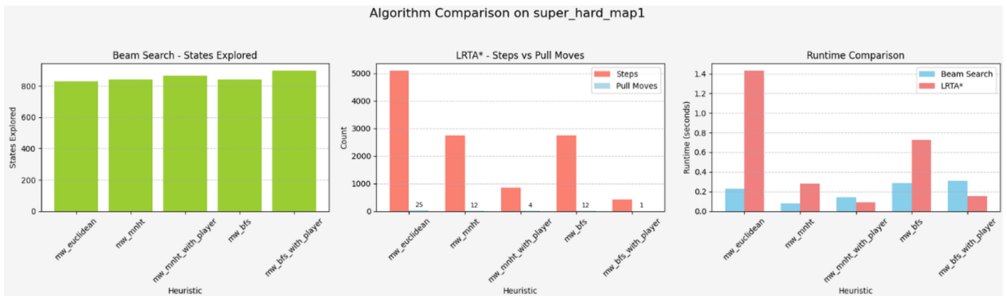

Figure 4: **super_hard_map**

1.3 BFS Distance Heuristic

The next step was following the same line of thought as before which is to try and optimize the method of calculating the distances between the boxes and the targets while maintaining the min_weight perfect matching. First, we removed diagonal moves from the Euclidean distance since they were impossible, leading to the Manhattan distance which indeed showed better results. The obvious thing to do next is to calculate the distances while ignoring unpassable obstacles i.e. walls and boxes. The best solution is using Breadth-First-Search.

Here's a few things I expected to happen since BFS only considers passable trajectories, the distances will be more accurate and thus the heuristic will perform better:

- The number of states explored will be lower since the heuristic will be more accurate and thus the algorithm will not get stuck in local minima as often.
- The runtime will be lower since it will converge faster to the optimal paths.
- We'll see these benefits especially for the larger maps and maps in which we have plentiful walls (hards, super_hard and medium2).

Observed behaviour:

- While for the easy and medium maps and *even the hard / super hard maps*, we can see a *very slight* decrease in the number of states explored and the runtime, as expected, the main benefits were observed for larger maps.
- For the larger maps, we can see a **0.5x factor decrease** in the total number of states explored.
- For the runtime, we can see a slight increase in the runtime for the larger maps, explained by the fact that BFS has a much higher time complexity than the Manhattan and Euclidean distances which are O(1) calculations. Therefore, my assumption for the runtime was wrong.

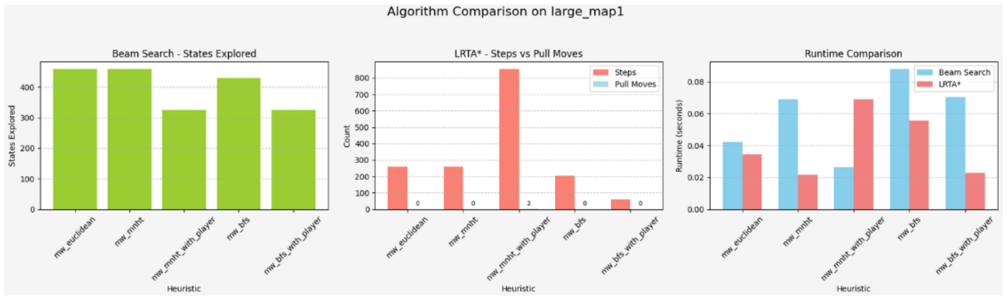

Figure 5: **large_1**

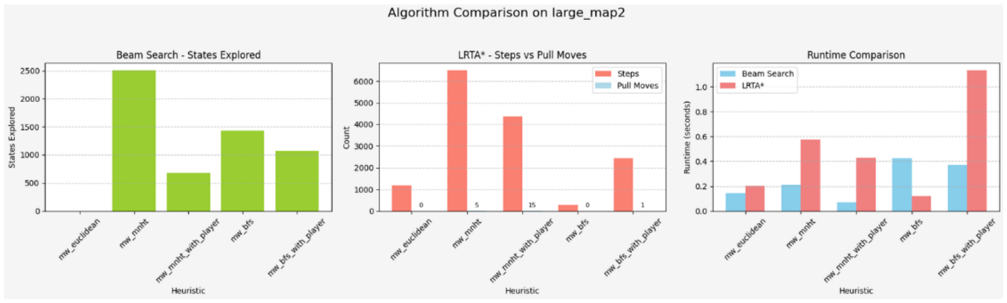

Figure 6: **large_2**

One critical conclusion I came up with is that, while Min-Weight Perfect Matching is a good heuristic, it's very important how these distances are calculated. As seen and described above, the performance of the algorithm is proportional to the quality of the method of calculating this distance. Therefore we can say that: BFS > Manhattan > Euclidean in terms of admissibility with a slight increase in computational time.

2 LRTA*

- When designing the LRTA* algorithm, I started off similar to Beam Search. I began by analysing its performance when using Min-Weight Perfect Matching where the distance is calculated using the Euclidean Distance.
- It should be noted that I allowed pull moves for LRTA* as I couldn't find optimal solutions for the all maps without them.
- One way to manage minimizing the pull moves I used is to assign an heuristic cost of 10 for pull moves, trying to avoid them as much as possible. Furthermore, I also considered that the heuristic cost of every move is 4 $(c(s, s'))$ - the cost of an edge in the graph search space. I did this to try to lower the number of "switches" between two adjacent states as this is a common pattern in a LRTA* search.

When using Min-Weight with Euclidean distance, the results were as follows in comparison to Beam Search with the same heuristic:

- The number of steps taken was significantly higher than Beam Search's explored states for more complex/ larger maps, as expected since LRTA* is an online search algorithm and it doesn't have the luxury of looking ahead in the search space.
- One interesting observation was that the number of steps was significantly *lower* for the easy maps.

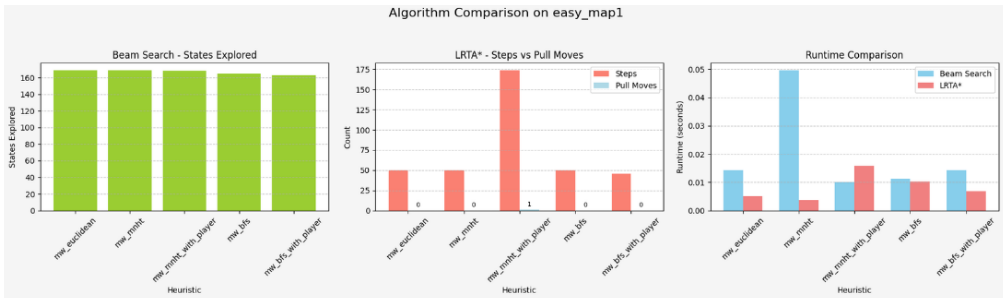

Figure 7: **easy_map_1**

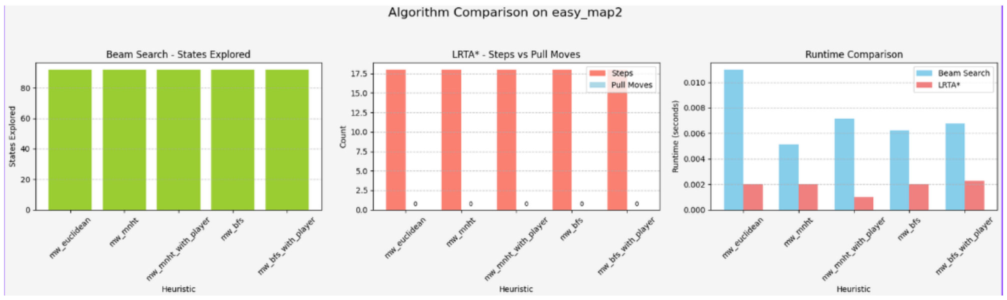

Figure 8: **easy_map_2**

- Next, as we move higher in the heuristic quality hierarchy, we'll analyse the performance using the Manhattan distance heuristic:

- As expected, the number of steps and implicitly the runtime for the majority of maps was lower than the Euclidean distance, though the difference was not as significant as I expected, in contrast to Beam Search where the difference was more pronounced.
- The only outlier was `super_hard_map2` where the number of steps was much lower than the Euclidean counterpart i.e. a **0.5x decrease factor** for the steps taken, a **0.7x decrease factor** for the runtime and **0.5x decrease factor** for the total pull moves used. This is probably because the map is so complex that it benefits from the diagonal moves assumptions that the Euclidean distance heuristic has in some way.

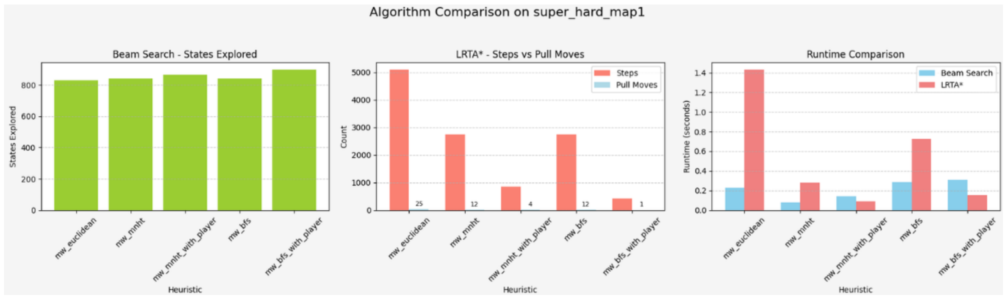

Figure 9: **super_hard_map**

- Next, the heuristic I was looking forward to testing the most: Distance with BFS:
- Surprisingly, the results were not as expected. On average, if not equal on some maps, the number of steps and implicitly the runtime was higher than both the other distance heuristics. The only outlier seems to be `large_map2` where BFS has a better performance, maybe because the map is the largest so it benefits more from distance that takes into account the walls and other boxes.

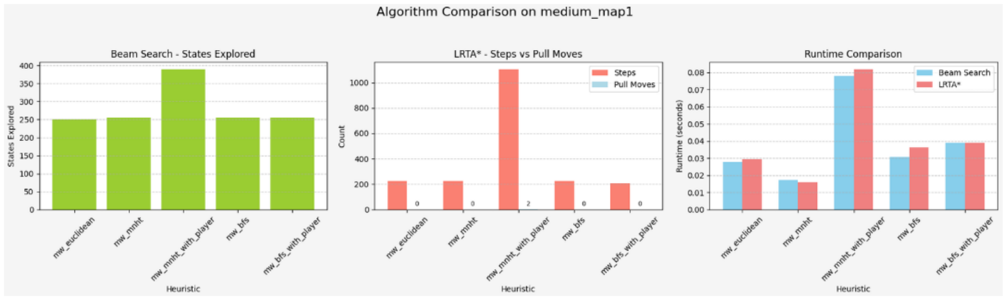

Figure 10: **medium_map_1**

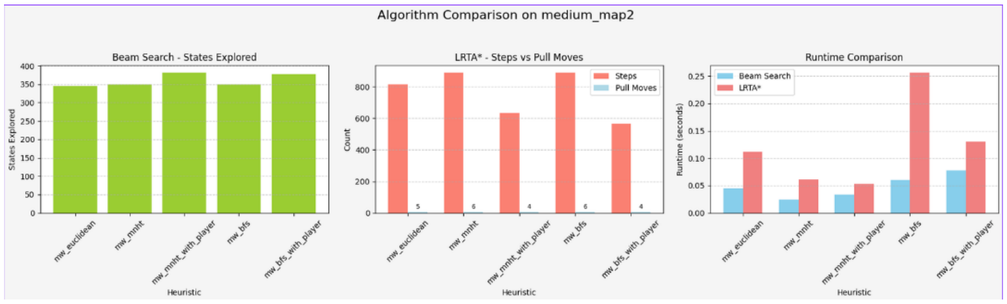

Figure 11: **medium_map_2**

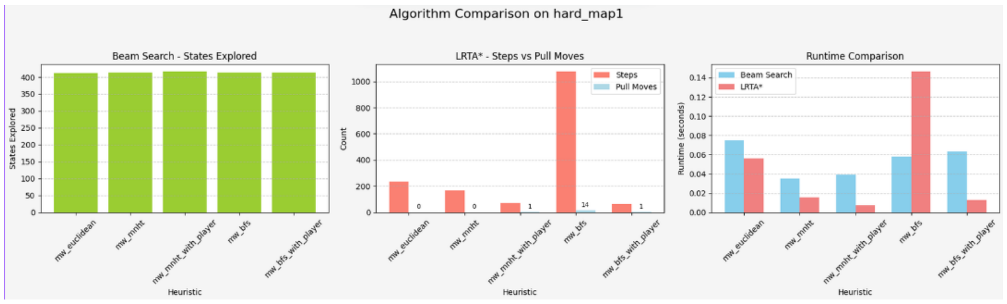

Figure 12: **hard_map_1**

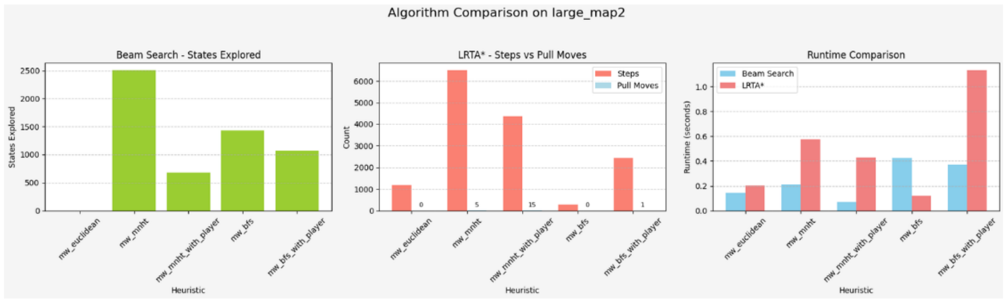

Figure 13: **large_map_2**

After comparing the performance of these three heuristics using LRTA* and Beam Search, I concluded that:

- The runtimes of the LRTA* approach using a given heuristic is evidently higher than the Beam Search approach using the same heuristic.
- I explained this by the intrinsic nature of LRTA*: it is an online search algorithm. While Beam Search is an offline one that has the advantage of looking ahead in the search space and check multiple paths corresponding to multiple neighbouring states, LRTA* only looks at the current state and its neighbours.
- After analysing the way LRTA* moves the player around the map, I observed *plenty* of redundant moves, especially in the case of larger maps where the player just moves back and forth between two states, the adjacent states switch I talked about earlier. I'm pretty sure this happens often because LRTA* gets stuck in a local minima and keeps "switching" until it can break free from it. To try to speed up this process, I increased the "cost of the edge" in the graph search space to 4 from an initial value of 1. After this modification, I've seen a good decrease in the number of steps taken in each map and also seen a reduced amount of redundant switches between states. The graphs attached to this document showcase LRTA*'s behaviour with a move cost of 4.

- After all this, I was still not satisfied with the behaviour of LRTA*, especially after observing a tendency of the algorithm to make the player just freely roam around the map without actually doing any work.
- One way I thought about reducing this behaviour is to add a penalty for those moves that lead the player away from the closest box, thus introducing two new heuristics:
  - `min_weight_manhattan_distance` + penalty for moves that lead away from the closest box = `mw_mnht_with_player`
  - `min_weight_bfs_distance` + penalty for moves that lead away from the closest box = `mw_bfs_with_player`
- The main difference between these two heuristics, beside the obvious fact that one uses BFS and the other Manhattan to calculate the cost matrix is how the distance away from the closest box is

calculated:

  – For the BFS heuristic, BFS is used to calculate.
  – For the Manhattan heuristic, the distance is calculated using the Manhattan distance formula.

- The "player-to-box proximity" penalty of a certain state is proportional to the sum of the minimum_weight perfect matching of the boxes and the target positions, multiplied by a factor "α" which differs in each case.

After implementing these two heuristics of LRTA* and analysing their results, here are the main conclusions:

For `mw_manhattan` with player-to-box proximity penalty:

- Surprisingly, the only expected result was in `large_map2` where the number of steps taken was reduced, meanwhile for all the other maps, even for large maps, the heuristic performed *worse* with a higher number of steps and runtime and even introducing more pull moves. Quite shocking.

For `mw_bfs` with player-to-box proximity penalty the results were more nearly to my expectations:

- For the majority of maps, the player-to-box proximity addition introduced performance benefits, averaging a slight lower number of steps taken to its counter-part heuristic on the easier maps.
- Meanwhile, for medium, hard, super-hard, and large maps we can see a *very* neat performance increase, ranging from **0.5x** to even a **0.2x** (`hard_map1`) runtime decrease multiplier and even *cutting out on the pull moves.*
- But, we still have the most counter-intuitive outlier as we've seen earlier during the Manhattan analysis, that is *large_map2* the map that even triggered me to come up with these proximity penalty. Amazingly, it performed *much worse* than its counter-part as well, seeing a **5x runtime multiplier** increase and even introducing a pull move that was absent before.

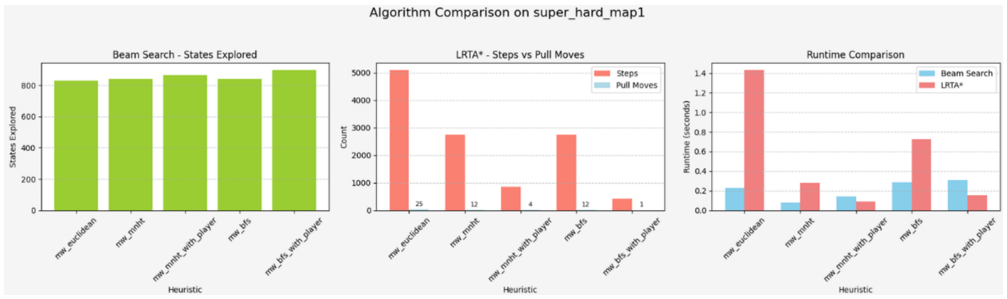

Figure 14: **super_hard_map**

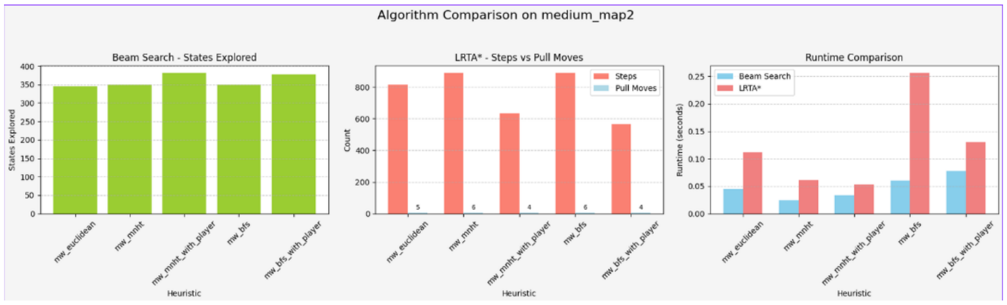

Figure 15: **medium_map_2**

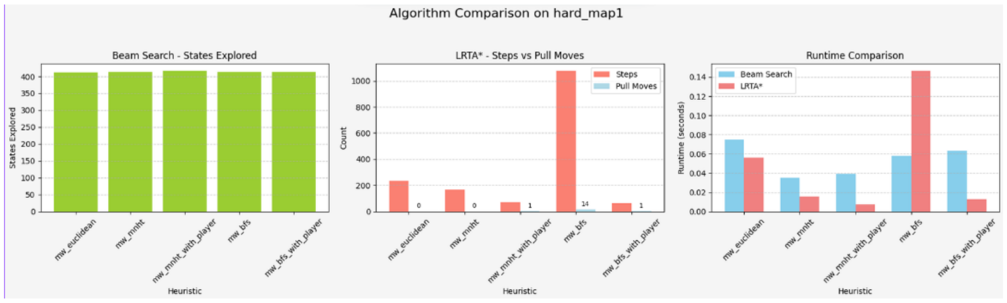

Figure 16: **hard_map_1**

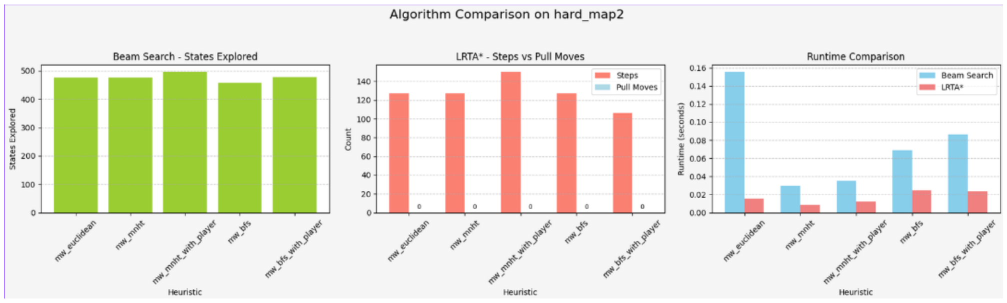

Figure 17: **hard_map_2**

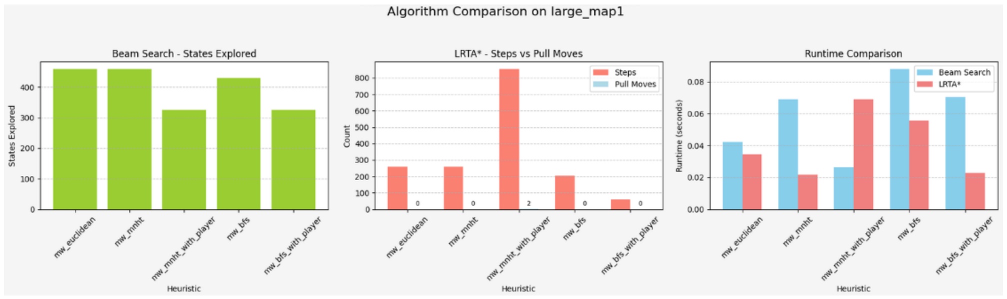

Figure 18: **large_map_1**

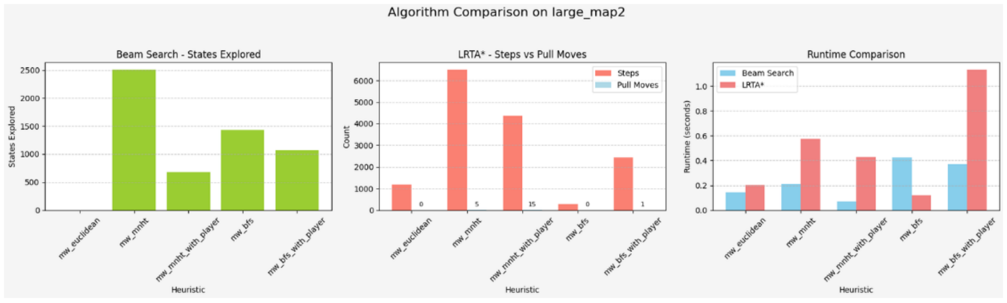

Figure 19: **large_map_2**

3 Beam Width

Even though I came up with this player-to-box proximity penalty mainly for LRTA* due to its online nature, I thought about running tests on the maps using this two new heuristics with the Beam Search algorithm. These were the results that I was most impressed with and I find them very interesting:

- The number of states explored for the easy, medium, hard and super-hard maps remained quite similar, with slight differences where the heuristic with no proximity penalty performed a bit better.
- But, for the larger maps we can see a very significant increase in performance. The proximity penalty heuristics performed much better, where the Manhattan one has seen an average of **0.5x decrease multiplier** in the number of states explored for both maps, meanwhile we see a **0.7x decrease multiplier** for the BFS ones.

Seeing this performance increase I was curious about how these heuristics will perform inside Beam Search if I lower the `beam_width` from an initial value of 15.

Firstly, I reduced the `beam_width` to 10 and here's what I noticed:

- The Euclidean distance heuristic now also fails on `medium_map2`, having previously failed only on `large_map2`.
- Now the BFS with and without proximity penalty heuristic fails on `large_map2`.

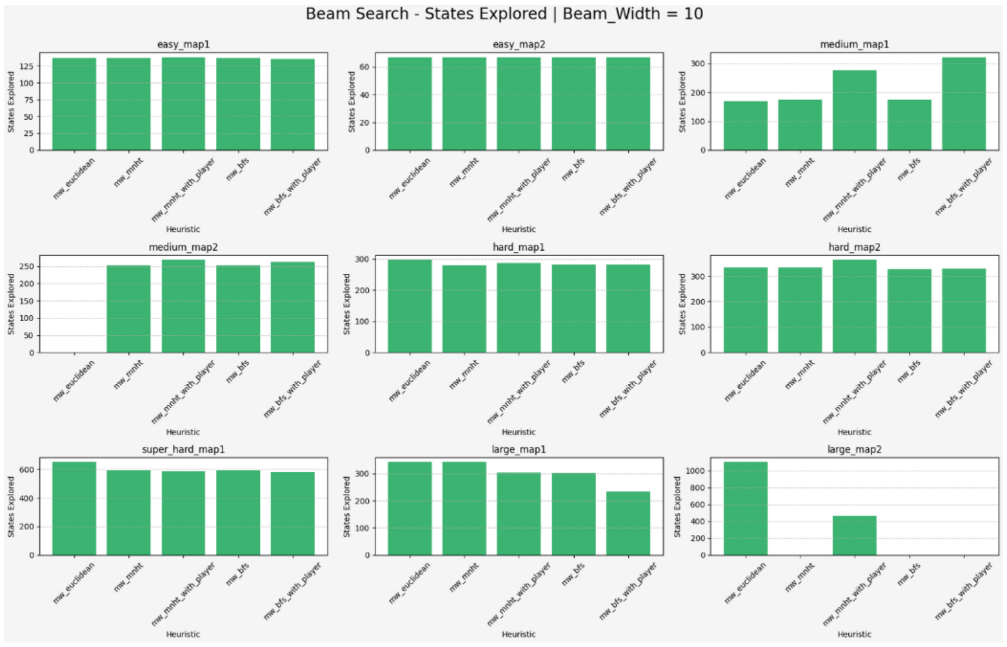

Figure 20: Beam Width = 10

Further, we'll reduce the beam width to 5, surfacing a very interesting attribute of one of the heuristics:

- Now, all the heuristics fail on `medium_map2` and `large_map2` and even on `super_hard_map1`, with the only heuristic that keeps on prevailing being: Min-Weight-Matching with Player-to-Box Proximity Penalty. Very interesting.
- Furthermore, now, with a **3x smaller beam width** than initially this heuristic even performs better than before averaging a smaller explored state space which was expected.

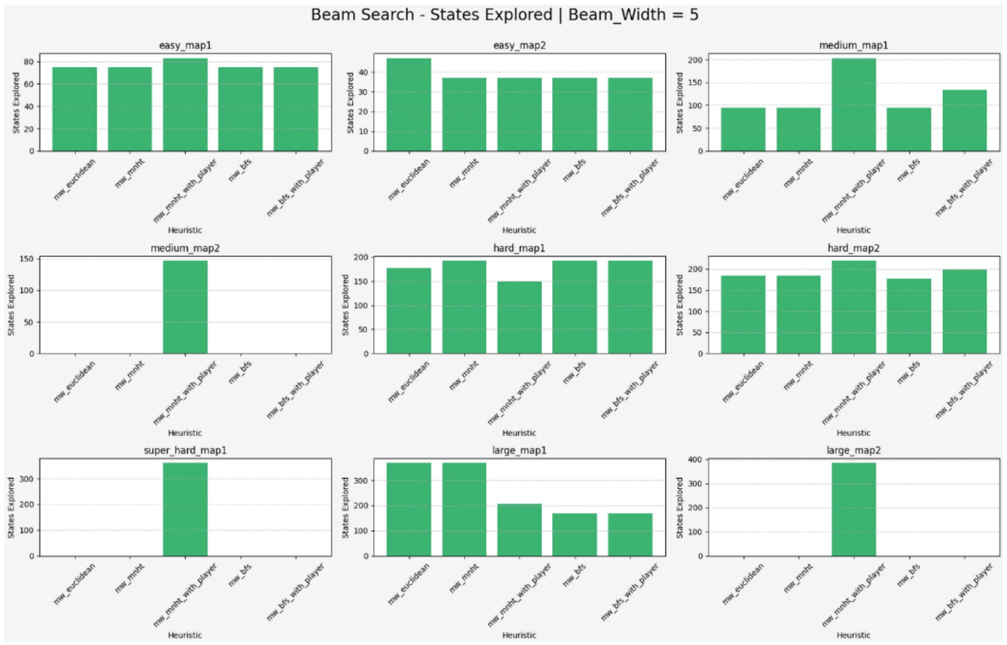

Figure 21: Beam Width = 5

In conclusion, the best heuristic for the Beam Search algorithm remains the Min-Weight Matching where distances are calculated using the Manhattan Distance, taking into account a Player-to-Box Proximity Penalty proportional to the sum of the previuosly mentioned assignments. Quite shocking since I designed this heuristic for LRTA*, not for Beam Search, how interesting!