Documentation for Sokoban solving

using Beam Search and Learning Real Time A\*

Overview

For both algorithms I tried enhancing the heuristics in order to avoid pull moves.

For both algorithms, I tried two different heuristics regarding the distance between the boxes and the targets.

First of all, the problem was, which target belongs to which box?

For the first draft of the algorithms, I tried mapping them randomly. The problem was, if I mapped them from the beginning, maybe there was no possible solution for that mapping. Thus, I mapped them randomly at each step when I was calculating the heuristic value of a state. To match the randomness, when having to make a choice between states with same cost, I chose randomly as well (for LRTA\* algorithm especially). Needless to say, the solutions varied a lot, and the number of explored states was huge.

Thus, I changed the mapping to be more intelligent and after some research, I used Hungarian algorithm, where the assigned cost was the shortest path between a box and a target (calculated using BFS algorithm). I did this mapping at the beginning and maintained it throughout.

Then, I observed that just taking into consideration the distance from a box to a target was not enough, because there were a lot of states with the same cost, states where the player had different positions. So I thought about adding to the cost, the distance of the path between the player and the boxes. At first, I added all distances between the player and all the boxes, but it was not very productive because there were too many variables to take into consideration, so if the player got closer to a box he got farther from another and the total distance may have remained the same. Thus, I needed to concentrate on one box at a time.

So here came the discussion, which box to consider?

There were multiple variants to solve this, and I thought of two: either take into account the closest box at that specific moment, or set a predefined order to put the boxes in place.

Having a predefined order sounds useful because as I have researched, the sokoban solution more often than not consists of multiple sequences of moves applied to the same box. However, it’s hard to calculate an order good enough to be possible to be solved. Whereas, using the closest box at each moment was a more flexible approach, as the “followed box” could dynamically change, and the distance between the player and the box would still follow a decreasing curve.

That’s why I found that following the closest box was a better approach.

Beam Search

Beam Search is like a BFS, for which you maintain only the best k results on every level.

For beam search, using the closest box approach worked very well.

Also, for this distance between the player and the box I used just BFS, as I wanted a more accurate result, because in this case there was no interaction between player and box to be taken into account, like there is when pushing boxes to targets.

For the distance between the boxes and the targets I tried two different approaches, using Manhattan distance and BFS distance. From my experience, BFS distance gave better results, as it was closer to the reality (less states explored and most importantly less pull moves).

To improve the number of pull moves, I implemented a BFS variant that calculates paths in form of “tunnels”, that is paths where a box could be pushed by the player (at each step, it was verified that a square was available both in the direction of movement as well as in opposite direction). This worked really well with beam search.

Also, for the same reason, I added to the heuristic a term that took into account the number of pull moves performed from the start state to current state. I used a factor of 10, in order to give more weight.

Another thing I observed, was that there were moments when on the path to the solution, a box that was placed on a target was moved again. So I thought about avoiding this. Thus I subtracted from the heuristic value of a state the number of boxes on the map for each box that was placed on a target. My idea was that this term I subtracted should be proportional with the total number of boxes on a map, rather than being a constant. This way, the issue was solved.

For beam search, the heuristic does no need to be admissible, so there were no worries about over evaluating the cost. Also, the exploration has the form of a tree with at most k nodes on each level and the solution is the path from the root (start state) to a leaf.

Another important aspect was the value of k. A smaller k gives better performance timewise, whereas a larger k gives the possibility of finding / finding a better solution.

Here is the analysis for different values for k. Note that a -10 number of pull moves means no solution was found.

* O imagine care conține text, captură de ecran, diagramă, Interval

  Conținutul generat de inteligența artificială poate fi incorect.K = 50

O imagine care conține text, captură de ecran, diagramă, Interval

Conținutul generat de inteligența artificială poate fi incorect.

* O imagine care conține text, captură de ecran, diagramă, Interval

  Conținutul generat de inteligența artificială poate fi incorect.O imagine care conține text, captură de ecran, diagramă, Interval

  Conținutul generat de inteligența artificială poate fi incorect.K = 25
* O imagine care conține text, captură de ecran, diagramă, Interval

  Conținutul generat de inteligența artificială poate fi incorect.K = 100

O imagine care conține text, captură de ecran, diagramă, Interval

Conținutul generat de inteligența artificială poate fi incorect.

Studying the pull moves results, here are some conclusions.

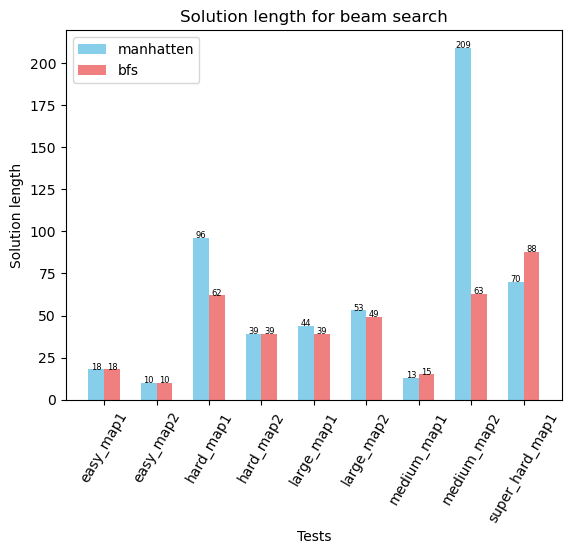
Firstly, for k = 25, it can be seen that a beam not wide enough can lead to tests that are not solved, because the solution does not lie in the searched area. This is an expected behavior. However, it would be also expected that the greater the k, the better the solution and it is visible that this is not always the case.

On easy\_map1, easy\_map2 solutions are exactly the same, given the fact that this maps are the easiest and require less states to be explored.

For Manhattan heuristic, there are surprising cases like hard\_map1, large\_map2, medium\_map1, where the k = 25 did unexpectedly well. This is most likely due to the fact that Manhattan distance is not so well informed, as it does not take into consideration obstacles on the way.

For the BFS heuristic, there are surprising cases like medium\_map2 where k = 100 did worse than k = 50. A reason for that would be that although BFS is more informed, it is not sufficiently informed and there can be multiple states with the same expected cost and the beam chooses them only by generated order.

On super\_hard\_map1, hard\_map1, large\_map1, the expected behavior is visible for both heuristics.

O imagine care conține text, captură de ecran, diagramă, Interval

Conținutul generat de inteligența artificială poate fi incorect.Here are the execution time results, only for K = 50, as for other cases they are proportional to the number of explored states.

Additionally, more often than not, the BFS heuristic produces a shorter path than the Manhattan one.

For further analysis, I chose k = 50, because the results were good enough, especially considering BFS distance and all maps were solved.

LRTA\*

For LRTA\* I chose the same two heuristics, Manhattan distance and BFS distance.

At first, I ran LRTA\* just once, and I considered the path being all the explored states. However, LRTA\* has its learning component, where it explores a path and then it comes to the conclusion that it’s better to return to a previous path. I considered this learning component should not be included in the solution, as in case of restart it would not be taken. Thus, I used a stack to eliminate the cycles in the path from the start state to the end one. This made the results comparable with the ones produced by Beam Search.

For the function that calculates the returning cost, I added 1 if the return move would be a push/normal move and 5 otherwise, in order to discourage returning by pull moves. 5 was chosen because I thought about avoiding a pull move, which usually means getting on the other side of the box, which in best case takes 5 moves.

With LRTA\* I saw that the algorithm had a hard time getting closer to any box, it would firstly explore the open space. Thus, I modified the weight of the distance between the player and the box, making it 2 instead of 1. This accelerated the results, but the cost would be that the heuristic may not be admissible. It would be admissible if we considered that, for every box, the player pushes it to the target and then it returns to the initial position and then gets to another box. Then the weight of 2 would be justified by the returning path. But that would imply a heuristic that would make the player return every time, which in this case is not implemented.

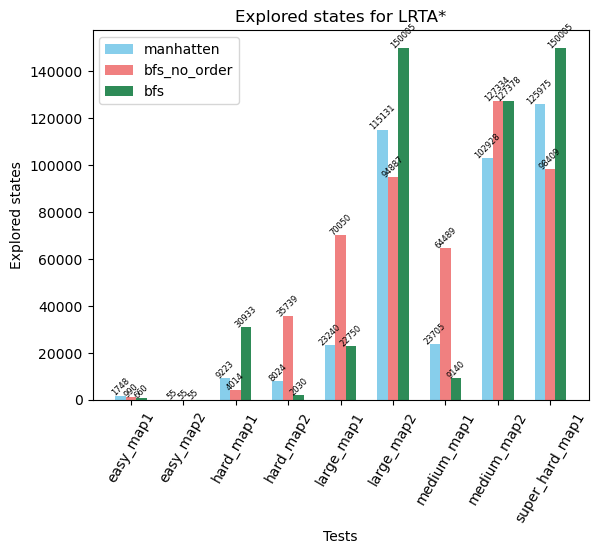
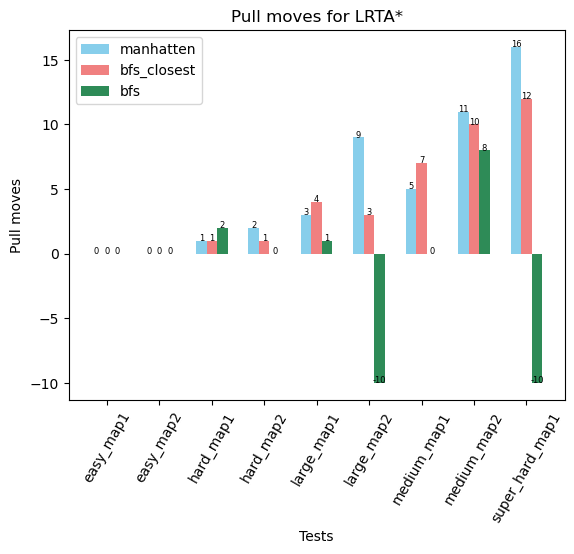
Moreover, I observed that Manhattan distance, although less informed and with worse results, would solve most maps, whereas BFS distance with tunneling did not. That is because BFS with tunneling may return “inf” cost when target is not reachable from specific state (that is it is not accessible without pull moves), and from there it’s more difficult to find a solution. I decreased the returning cost added for a pull move (that initially was 10 instead of 5) and I had results for more tests than before.

The “inf” has the purpose of limiting the number of pull moves; however, because “inf” is still considered a cost and added to the other terms, the algorithm can still converge if all available moves get to a value greater than “inf”, where it basically cancels and the other components of heuristic function make the difference.

The test that would fail with BFS distance was a medium one, where I observed that the boxes were very close initially to each other. I figured out that the player would toggle between states when I used the “closest box now” approach. That is why I implemented the predefined order. It solved this problem, but there were other more difficult tests that would remain unsolved, so it was a trade-off.

Then I thought about running LRTA\* multiple times, because ideally it gets better by learning. It worked, sometimes it gave better results, but the curve of them was not increasing as I had expected. Most probably the problem comes from the heuristic function, that as I have said is not necessarily admissible.

These are the LRTA\* results after 5 iterations for each heuristic, storing the best result. For the number of explored states, I added them all for the 5 iterations, although not very suggestive, as there are iterations where no solution is found and the 30000 states limit is reached.



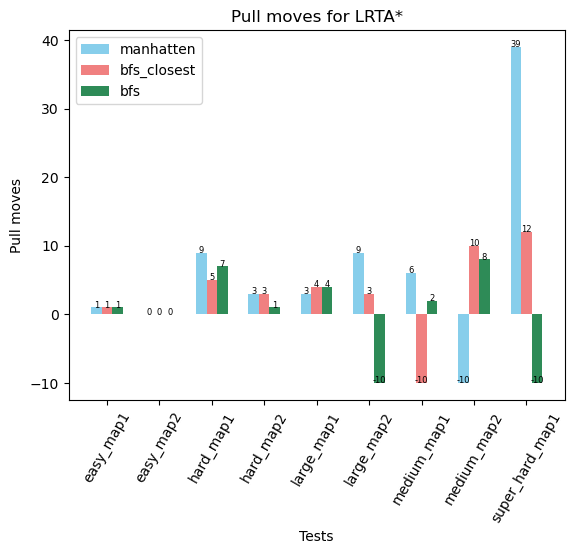
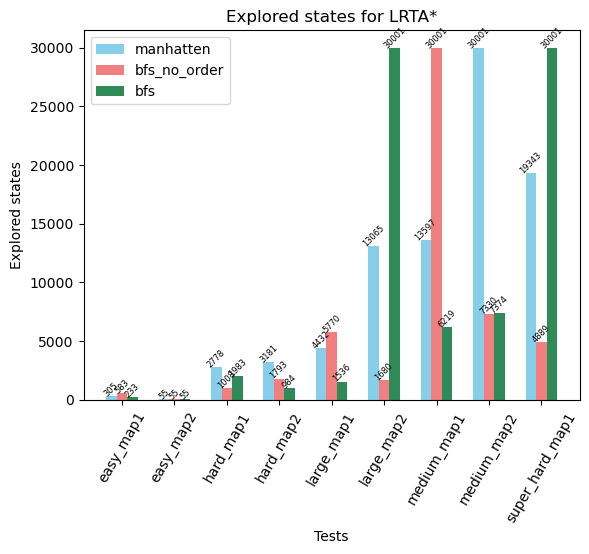
O imagine care conține text, diagramă, captură de ecran, Interval

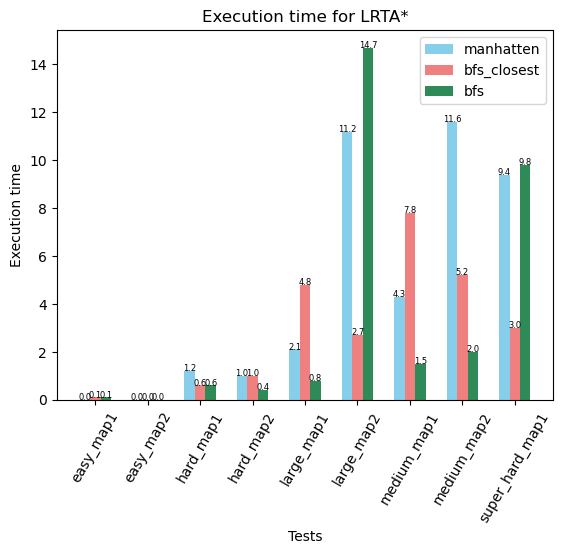
Conținutul generat de inteligența artificială poate fi incorect.

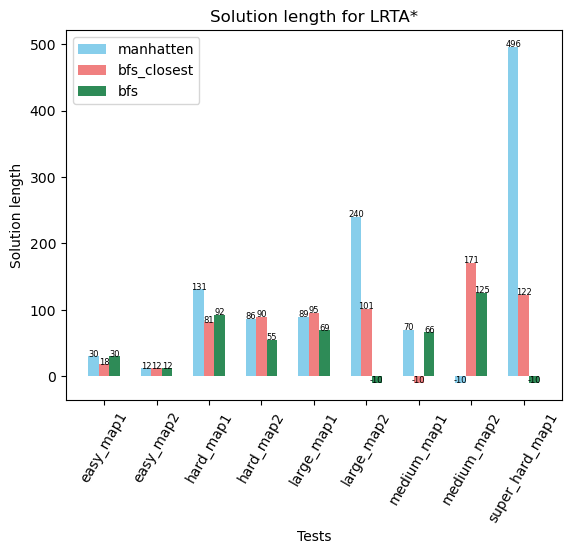
Execution time cumulates execution time for the five iterations.

These graphics merely show that the solution usually contains multiple sequences of moves that target specific box, thus the number of pull moves is smaller when considering a predefined order. However, we can see there are maps that are not solved this way.

That’s why for future analysis I chose “closest box now” approach.

Below are the results for just one lrta\* iteration, without restarting:





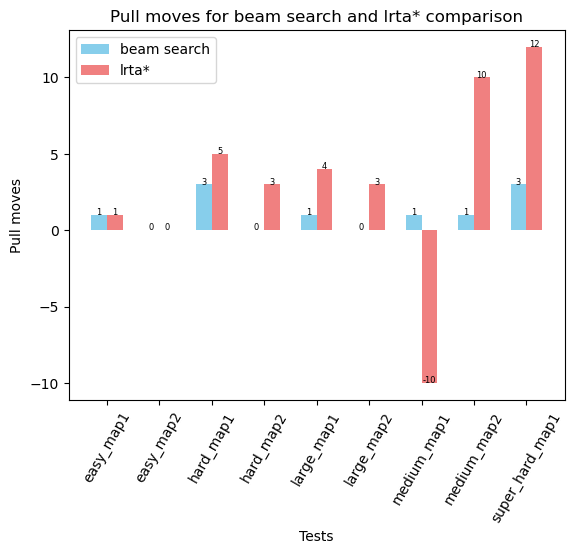
The above mentioned behavior can be seen, medium\_map1 where the boxes were close to each other fails with “closest box” approach. Good thing is each method fails on another test, they do not fail simultaneously. The reason would be that neither heuristic takes into account the overall form of the map.

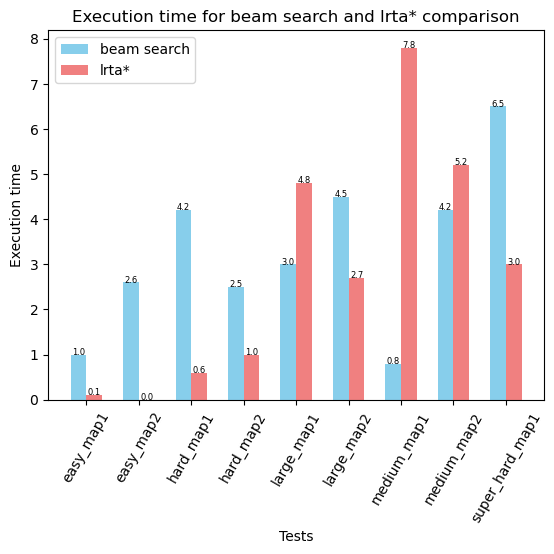
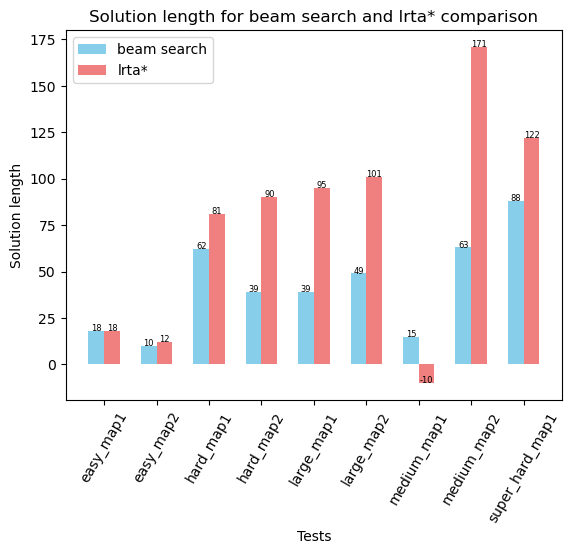
Comparison

For the execution time, for easy maps, where the solution is pretty straightforward, lrta\* executed way faster, whereas beam search explored more states than necessary. That is because beam width was a constant, not taking into account the size of the map.

O imagine care conține text, captură de ecran, diagramă, linie

Conținutul generat de inteligența artificială poate fi incorect.For larger cases, results are comparable, both having its advantages depending on the map form.





On average, beam search finds better results. A reason for this would be that learning real time A\* is suitable for situations when the searching area is not known or maybe it changes dynamically.

In our case, the maze is static and such an “exploration” approach is not necessarily needed.

Beam search implements a BFS, it finds a shorter path that does not contain cycles, whereas LRTA\* is more of a DFS, which explores in a more unordered manner. It can be seen that Beam search finds shorter solutions as well.

Beam search makes a choice from the beginning regarding the area where the solution may lay, meaning it sometimes does not find the optimal solution, as it was seen with a greater k, where sometimes the optimal solution was not found because on a previous level an incorrect pruning of the backtracking tree took place.

Learning real time A\* would converge to an optimal solution, but if it was run multiple times. I have tried restarting it heavily, with less steps to do and the overall results were better.

Bibliography

* <https://weetu.net/Timo-Virkkala-Solving-Sokoban-Masters-Thesis.pdf>
* <https://www.ijcai.org/Proceedings/05/Papers/0764.pdf>