Sokoban solver - AI Homework 1

This collection of scripts implements a Sokoban solver using two common search algorithms: LRTA* and Beam Search. The goal is to find a solution to the Sokoban puzzle, which involves moving boxes to designated target locations in a grid-based environment.

In order for these algorithms to work, we need to define good heuristic and cost functions. The heuristic function should take a state of the game and return a value. The state consists of the position of the player along with the positions of the boxes. The returned value should be smaller as the box approaches the target. The cost function should return the cost of moving a box from its current position to the next required position.

Part 1: LRTA*

This section is dedicated to implementing the LRTA* algorithm.

First intuition

The most basic heuristic function one can think of is the Manhattan distance between the box and the target. But the player should also move towards the box. So we need to take into account the distance between the player and the box. The heuristic function should return the sum of the Manhattan distance between the box and the target and the Manhattan distance between the player and the box.

For the first attempt, we will leave the cost function as 1.

Sneak peek of the heuristic function:

```
def h1(s: Map, visited: Optional[dict] = None) -> int:
    return (
        # distance from player to closest box
        min(manhattan(s.player, box) for box in s.boxes.values()) +

        # distance from each box to its nearest goal
        sum(min(manhattan(box, goal) for goal in s.targets) for box
in s.boxes.values())
    )
```

With this parameters, the results are the following:

In [ ]:
```
# Results for h1, c1
```

```
Algorithm: lrta_star
States explored: 111
Duration: 0.0102 seconds
Pulls: 6
Solution found: True
```

As it can be seen, the algorithm is not very efficient. It visits a lot of states before finding the solution. We can search for a better algorithm.

Second intuition

The second heuristic function is a bit more complex. Instead of computing the distance between the player and thhe box, it searches for the position where the player can push the box towards the goal. The distance is then computed relative to that position. The heuristic function is somewhat like this:

```
def h2(s: Map, visited: Optional[dict] = None) -> int:
    ...
    # Now check the direction in which the box should be pushed, so
that the player
    # should go behind the direction of the box as much as possible
    diff = (box.x - goal[0], box.y - goal[1])
    if abs(diff[0]) > abs(diff[1]):
        direction = (sign(diff[0]), 0)
    else:
        direction = (0, sign(diff[1]))

    required_player_pos = (box.x + direction[0], box.y +
direction[1])

    # Add the manhattan distance from the player to the required
position
    ...
    return total
```

However, after performing the test, we can see that no improvement was made. The algorithm is still slow and visits a lot of states before finding the solution:

In [ ]:
```
# Results for h2, c2
```

```
Algorithm: lrta_star
States explored: 145
Duration: 0.0137 seconds
Pulls: 8
Solution found: True
```

This may be because the now there are additional constraints on the player position.

Third intuition

This is the most complex variant of them all. After long trial and errors, I found that the following ideeas will result in a better and faster solution:

1. The player should be positioned in a way that he can push or pull the box towards the goal.
2. When moving a box, we should try to move it towards the goal, but also to a position where the player can push it again.
3. Because of this, avoid the corners.
4. Boxes (not states) should avoid going back to a position where they were before.
5. Previously visited states (including the player) should be avoided.
6. The cost MUST be computed based also on the previous state. A standalone state cannot be evaluated, because a move may be good but push the box in a state with a higher heuristic.

Knowing all these, I have separated the heuristic function from the cost function. The heuristic function is now a simple Manhattan distance between the box and the target. The cost function is more complex and always gives a small value if the all the above conditions are met. The cost function uses the `_best_move` function. This one attempts to find a box position that is closer to the goal and a player position that can push the box towads the goal. The cost function can be found inside the `heuristics.py` file.

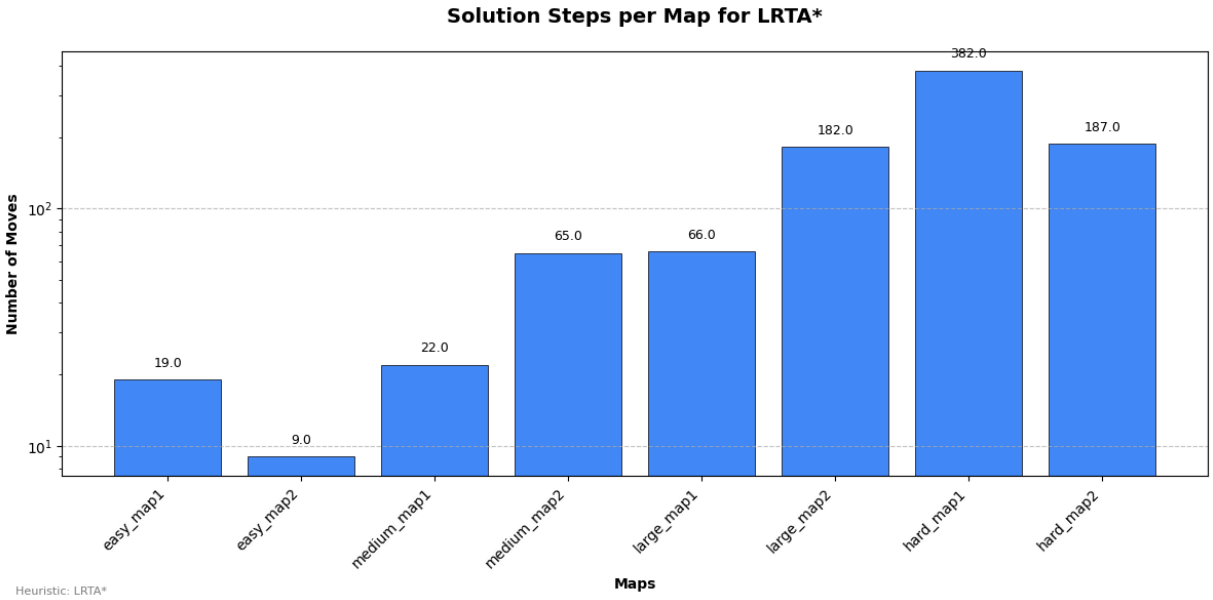
I also had to slightly modify the original variant of the LRTA* cost function, in order to adapt it to this game. unknown states are computed based on the next state (in order to avoid randomness) and I added a value of 50 for already visited states (to avoid going back to them).

Analyzing again, we see the following:

In [ ]:
```
# Results for h3, c3
```

```
Algorithm: lrta_star
States explored: 19
Duration: 0.0036 seconds
Pulls: 0
Solution found: True
```

In [ ]:
```
# Results plot for h3
```

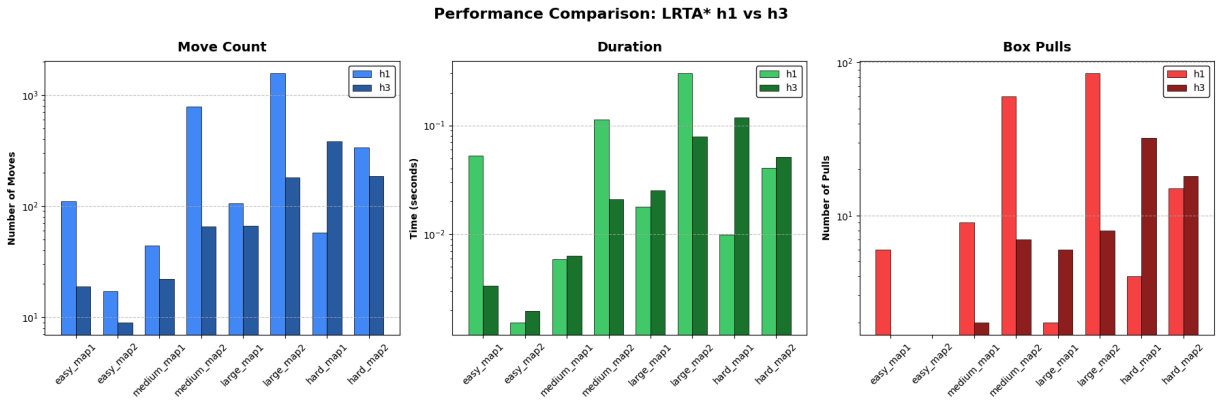


Way better, this solution is the best possible one.

Analysis

Putting the first heuristic near the third one, we can see the following:

In [ ]:
```
# Comparison between h1, c1 and h3, c3
```



We can see that there is an improvement on average for the third heuristic. Als, note that I chose to use logarithmic scale, in order to scale down the values.

The gif of the first map can be seen in `main.py`.

Part 2: Beam Search

This section is dedicated to implementing the Beam Search algorithm.

Intuition (Knowing what we want from LRTA*)

In order to implement the Beam Search algorithm, we should note the following:

1. The algorithm should stop on success, not in a local minima. Because of that, never go in a previously visited state (keep a Set of visited states).
2. Use the third heuristic function from the LRTA* algorithm. Because of how the algorithm works, other heuristics will loop indefinitely.

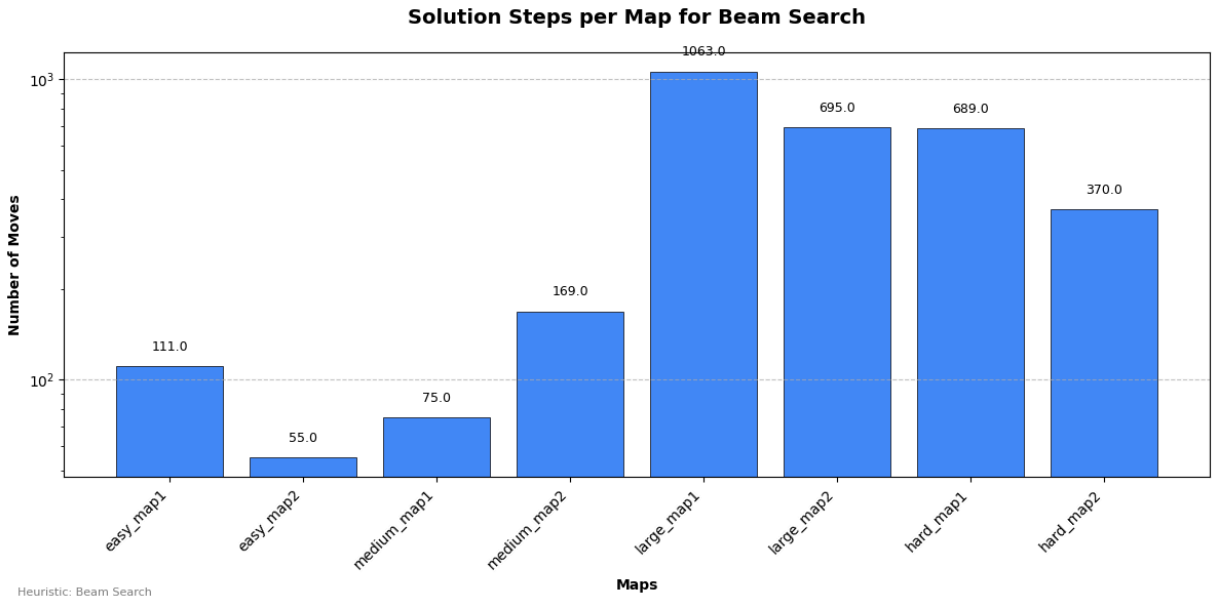
For beam, search, I did not include a pulls analysis because it would have been too complex for our small implementation.

We can perform different tests for different values of the beam width K. The results are the following (a value of K under 6 is too small for the last tests):

In [ ]:
```
# Results for beam search
```

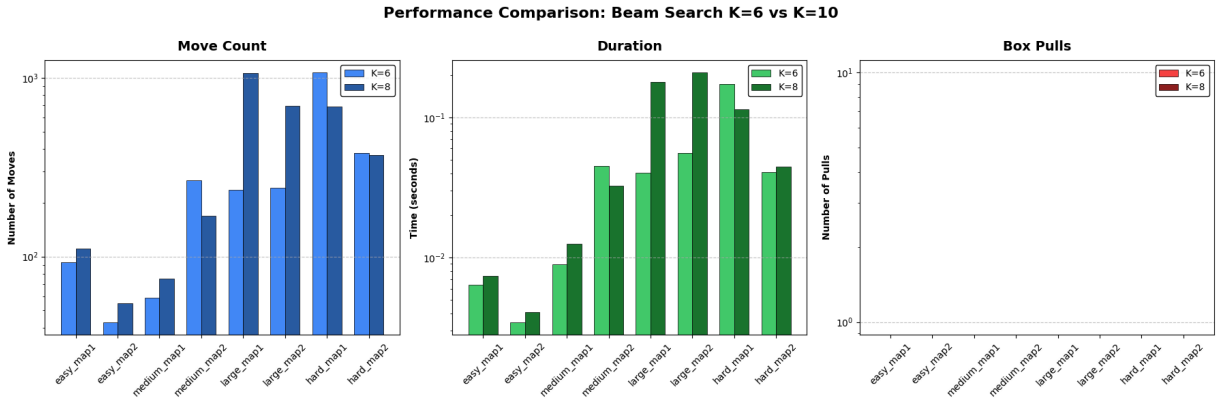
```
Algorithm: beam_search
States explored: 93
Duration: 0.0066 seconds
Pulls: 0
Solution found: True
```

In [ ]:
```
# Results plot
```



In [ ]:
```
# Comparison between K=6 and K=8
```

```
/Users/matei/Desktop/sokoban/analysis/utils.py:153: UserWarning: Data has no
positive values, and therefore cannot be log-scaled.
  ax.set_yscale('log', base=10)
```

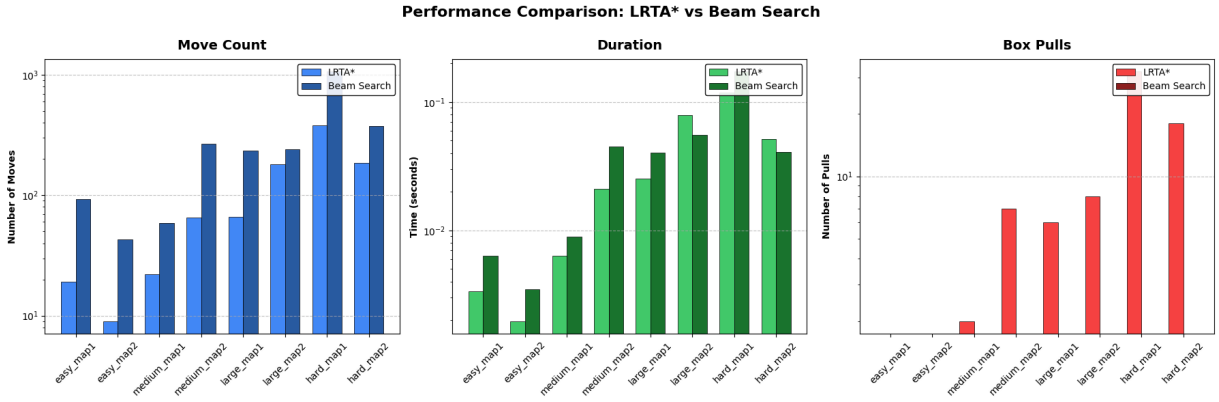


We can see that, for our case, a value smaller than 6 is taking way too long and a value larger than 6 does not improve the results. The best value is 6, which is also the default one.

Comparison: LRTA* vs Beam Search

We can put the two algorithms against each other. The results are the following:

In [ ]:
```
# Comparison between LRTA* and Beam Search
```



Conclusion

The LRTA* algorithm is the best one for this case. The Beam Search algorithm is not very efficient and it does not improve the results, but can be used for other cases.

Images

Here is a "GIF" of the first simulation of LRTA* on the second map:

In [ ]:
```
# The solution for easy_map1
```

```
Algorithm: lrta_star
States explored: 9
Duration: 0.0021 seconds
Pulls: 0
Solution found: True
```