Heuristics and Search Algorithms for Sokoban

Adrian-Nicolae Ariton
POLITEHNICA Bucharest.
adrian.ariton@stud.acs.pub.ro
adrian.ariton0@gmail.com

Abstract

This paper explores various heuristics for **Sokoban** and builds on top of them in order to reach a valid ϵ-admisible heuristic. This further compares K-BeamSearch and LRTA*. The code for this report is located at github.com/adrianariton/sokoban

1 Heuristics

A simple approach, which is not included in this report, is to compute the manhattan distance between each box and each target, and apply the Hungarian algorithm to do a least score match. This approach has been discussed in many papers, so another approach has been chosen for this report. The scope of this report is finding a complex heuristic that correctly approximates the number of moves it would take a player to move a box.

2 A pseudo-Dijkstra heuristic

In the case of a single box and a single target, the go-to algorithm is a modified Dijkstra that calculates the distance from the box to the next box position closest to the target and the player position to the square that can push the box towards that position, i.e. a 'temporary' cost.

This constitutes the **box_to_targets** function that takes as input the map, a box and a player position and a list of targets (possibly only one) and yields a minimal correct cost to get the box to one the targets and the position the player needs to be in in order to do the firt push of the box.

In: A. Ariton; https://github.com/adrianariton/sokoban

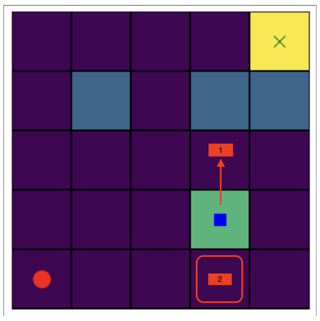

Figure 1: The 'temporary' cost to move the box towards the target is calculated as the sum between the distance from the box to the next square (the one marked with '1') and the distance from the player to the square it needs to be to be able to push the box forward. i.e. $4 + 1 = 5$

2.1 The box_to_targets function

The algorithm for this iteratively calculates the 'temporary score' of pushing a box closer to the target, by using a heap that stores the current cost, the box position and the player position for the current branch of the state-tree. It then calculates the next position of the box by making sure the one symmetric to it w.r.t the box is also unobstructed by walls and bounds.

For example in Figure 1, the algorithm doesn't take into consideration moving the box right, because afterwards, there is no way of escaping the right edge of the map.

2.1.1 Sampling initial directions.

Because this approach proved to only work for the single-box tests for LRTA*, a more informed heuristic was built on top of this. We need to figure out the position the player needs to be in order to push the box in the most optimal manner.

This is done by iterating through the 4 possible initial box moves and then applying the **box_to_targets** algorithm to figure out the optimal cost. After that the cost from the player to the corresponding square is calculated and returned by the function.

Algorithm 1 box_to_targets_bfs

```
Require: sokoban_map, box_pos, player_pos, targets, blocked
Require: box_block (default False), update_reachable (default True), verbose (default False)
 1: if box_block then
 2:     reachable, player_dists ← reachable_positions(sokoban_map, player_pos, blocked ∪ {box_pos})
 3: else
 4:     reachable, player_dists ← reachable_positions(sokoban_map, player_pos, blocked)
 5: end if
 6: if box_pos ∈ targets then
 7:     return (0, box_pos)
 8: end if
 9: min_box_dists ← empty list
10: for each direction $({}_dx,{}_dy)$ ∈ $\{(-1,0),(1,0),(0,1),(0,-1)\}$ do
11:     cost ← find_min_push_cost(sokoban_map, box_pos, player_pos, blocked, targets, $({}_dx,{}_dy)$, box_block)
12:     Append $(({}_dx,{}_dy), cost)$ to min_box_dists
13: end for
14: min_cost ← 10050.0
15: min_ppos ← box_pos
16: for each (dir, cost) in min_box_dists do
17:     $ppos$ ← box_pos $-dir$
18:     if cost < min_cost then
19:         min_cost ← cost
20:         min_ppos ← ppos
21:     end if
22: end for
23: if min_ppos ∉ reachable then
24:     return (10050.0, min_ppos)
25: end if
26: return (min_cost, min_ppos) =0
```

Algorithm 2 find_min_push_cost

```
Require: sokoban_map, box_pos, player_pos, blocked, targets, direction $({}_dx,{}_dy)$, box_block
 1: $bpos$ ← box_pos + $({}_dx,{}_dy)$
 2: $ppos$ ← box_pos $-({}_dx,{}_dy)$
 3: if not inbound(bpos) or not inbound(ppos) then
 4:     return 1000.0
 5: end if
 6: if is_obstacle(bpos) or is_obstacle(ppos) then
 7:     return 1000.0
 8: end if
 9: best ← {bpos: 0}
10: heap ← [(0, bpos, player_pos)]
11: while heap is not empty do
12:     (cost, (bx, by), (pl_x, pl_y)) ← pop minimum heap element
13:     if (bx, by) ∈ targets then
14:         return cost
15:     end if
16:     if box_block then
17:         reachable, player_dists ← reachable_positions(sokoban_map, (pl_x, pl_y), blocked ∪ {(bx, by)})
18:     else
19:         reachable, player_dists ← reachable_positions(sokoban_map, (pl_x, pl_y), blocked)
20:     end if
21:     for each $(dx, dy)$ ∈ $\{(-1,0),(1,0),(0,1),(0,-1)\}$ do
22:         $nx, ny \leftarrow bx + dx, by + dy$
23:         $px, py \leftarrow bx - dx, by - dy$
24:         if not inbound(px, py) or is_obstacle(px, py) then
25:             continue
26:         end if
27:         if not inbound(nx, ny) or is_obstacle(nx, ny) then
28:             continue
29:         end if
30:         if (px, py) ∈ blocked or (nx, ny) ∈ blocked then
31:             continue
32:         end if
33:         if (px, py) ∉ reachable then
34:             continue
35:         end if
36:         if (nx, ny) ∈ targets then
37:             sc ← 0
38:         else
39:             sc ← 1
40:         end if
41:         new_cost ← cost + player_dists.get((px, py), 1) + sc
42:         if new_cost < best.get((nx, ny), ∞) then
43:             best[(nx, ny)] ← new_cost
44:             push (new_cost, (nx, ny), (px, py)) to heap
45:         end if
46:     end for
47: end while
48: return 1000.0
    =0
```

The **box_to_targets_bfs** function returns the minimum cost of pushing a box to any of a given list of targets, by ignoring the **blocked** set of squares and, depending on the *box_block* parameter, the box position as well.

The variant with *box_block* set to *True* is used in the context of this paper because of its correctness, however using the False variant can also yield good results, as it makes the heuristic more likely to be admissible for multiple boxes.

2.2 Reductions for multiple boxes

For each box, the above box to targets function is called and reduced using either **sum** or **min**. Min makes the heuristic admissible however it introduces possible jumps in scores that discourage pushing a box to a target when it is neighboring the target.

This is best seen in the **Medium-1** test.

```
_  _  _  B(2)  X(2)  _  _        _  _  _  B(2)  X(2)  _  _
_  _  /   _    X(1)  _  _        _  _  /   _    B(1)  _  _
_  _  /   _    B(1)  _  _        _  _  /   _     P    _  _
_  _  _   _     P    _  _        _  _  _   _     _    _  _
_  _  _   _     _    _  _        _  _  _   _     _    _  _
```

Figure 2: The figure on the left is the initial setup, and the one on the right right is after pushing the box. When the box is pushed, the **'min'** reduction can either:
- 1. ignore B(1) as it matched with the target, case in which the cost goes from 1 to $8 + 1 = 9$, so the agent may chose not to push the box which is bad, or
- 2. not ignore B(1) cas in which the agent would choose not to leave to go to the other box after it pushes B(1) because the cost would increase. In both cases without an incentive to match the boxes and a more complex reduction such as sum, the player would oscilate aimlessly between two states.

In order to fix this, the **sum** reduction is used along with an incentive cost that makes the player push boxes to target. (for SOKOBAN_PLAYER_SEEDED_TARGET 5 is added to the cost for each unmatched box or -100 for each matched box for SOKOBAN_PLAYER_TARGET - which makes the function not work with LRTA*).

The first approach makes the heuristic ϵ-admissable with a pretty large epsilon of $5 * number\ of\ boxes$, but because the reduction used is **'sum'**, and given that the boxes cannot all be neighbours (i.e. all in a square shape with 4 angles), the minimum cost for an unsolved puzzle with a random player position will always exceed $5 * number\ of\ unmatched\ boxes$, we are guaranteed that we get to the solution after a finite amount of iterations of LRTA*.

The incentive was added to speed up the process of learning and require no restarts of the algorithm.

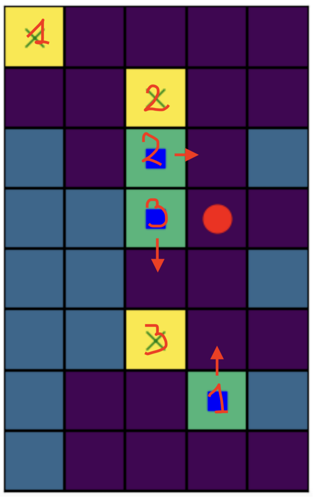

Figure 3: The boxes for the **medium_2** map have complicated paths to their respective targets; their respective first reached target found by the *box to targets* function can change resulting in oscillating states for the player.
The boxes 1 and 2 would fight over the target_2, because box_1 is closer to the target_2 than it is to target_1. The matches should be either (1-2, 2-1) or (1-1, 2-2). Thus permutation matching is needed.

2.2.1 Player to target cost reduction

As discussed previously, the player needs to be able to reach its target box in order to be able to push it. This cost translates in the BFS distance between the player and the 'behind-position'[1] of the box it decides to push. This is reduced using **min** for multiple boxes to not overestimate the cost of the player.

2.2.2 Oscillations

Oscillations can occur when the **box to targets bfs** is calculated for each box w.r.t all targets.

2.3 Medium_2 test and Oscilations

In cases when the paths from boxes to their targets are not so simple, oscilations of the player's position may result.

This happens because the *box to targets* function takes into account all of the targets for a single box, so two boxes can reach the same target when the algorithm is applied, resulting in skewed cost functions and in fights between 2 boxes for the same target.

An example is the **Medium_2** test shown in Figure 3

[1]By behind position we mean the position the player needs to be in order to be able to push the box

Algorithm 3 Sokoban Player Target Cost Function

```
0: function SOKOBAN_PLAYER_TARGET($sokoban\_map$)
0:   $player\_pos \leftarrow (player.x, player.y)$
0:   $boxes \leftarrow listofboxes$
0:   $targets \leftarrow listoftargets$
0:   $obstacles \leftarrow listofobstacles$
0:   $boxposes \leftarrow listofboxpositions$
0:   $free\_targets \leftarrow [t \mid t \in targets, t \notin boxposes]$
0:   if is_deadlocked($sokoban\_map$) then
0:     return 100000.0
0:   end if
0:   $box\_in\_targets \leftarrow 0$
0:   $cost \leftarrow +\infty$
0:   $sums \leftarrow []$
0:   for all $box \in boxposes$ do
0:     if $box \in targets$ then
0:       $box\_in\_targets \leftarrow box\_in\_targets + 1$
0:       continue
0:     end if
0:     $other\_boxes \leftarrow boxes \setminus \{box\}$
0:     $(score, direction) \leftarrow$ box_to_targets_bfs(box, player_pos, free_targets, blocked = obstacles $\cup other\_boxes, box\_block = True)$
0:     $dist \leftarrow$ bfs_distance(player_pos, [direction], forbidden = boxposes $\cup obstacles)$
0:     $box\_cost \leftarrow score + dist$
0:     $cost \leftarrow \min(cost, box\_cost)$
0:     Append $(score, dist)$ to $sums$
0:   end for
0:   if $box\_in\_targets = len(boxes)$ then
0:     return $-(box\_in\_targets) \times 100$
0:   end if
0:   $fsc \leftarrow 0$
0:   $max\_d \leftarrow 0$
0:   for all $(s, d) \in sums$ do
0:     $max\_d \leftarrow \max(max\_d, d)$
0:     $fsc \leftarrow fsc + s$
0:   end for
0:   return $cost - (box\_in\_targets) \times 100 + max\_d$
0: end function=0
```

2.4 Permutations matching for sokoban_player_target

In order to bypass the oscillation problem, a reduction is made over the minimum of all box to targets permutations (similarly to a brute force Hungarian algorithm.), and the **sokoban_player_target** is applied for each permutation, and the minimum is chosen as the cost.

This makes the function take very long for a big number of boxes or large state space, (test **Large_2**), so for bigger tests, the Hungarian algorithm is to be used.

To make the heuristic positive the -100 incentive for each correct box is replaced with a +5 incentive for each unmatched box, resulting in the **sokoban_player_seeded_target** function.

3 Deadlock Detection

In order to detect invalid maps a deadlock detection is employed, and such a map is considered invalid if:

1. a box is cornered by walls or bounds and is not in its asigned target
2. a box is stuck on an edge and there is no asigned target on that edge.
3. there is no path from the player to the assigned 'behind position' of a given box

4 Redundancy Detection

In order to help LRTA* learn faster, a redundancy detection is added to each player move. A player move is considered redundant if:

1. the resulting player position is surrounded 3 ways by walls and the other way is not a box. 2. the player oscillates (only used for BeamSearch as LRTA* already employs learning into it's core)

5 Search Algorithms

The search algorithms used in this report are K-BeamSearch for $K = 10$ with stochastic climbing, and LRTA*.

5.1 Stochastic K-BeamSearch

The BeamSearch keeps a queue of current states and a queue of candidate states, all with length K. Next states for each state in queue are sampled w.r.t their cost and the temperature, similarly to Simulated Annealing, and if the new minimum score is an improvement, the new candidate queue gets updated with the candidate state, else, there is a small probability based on the temperature that we update the candidate queue regardless of weather the candidate is an improvement or not.

This probability approaches 0 as the number of iterations increases. Each queue element maintains a state a score and the moves queue, in order to be able to trace back the moves from the initial state. Further more, on top of the **SOKOBAN_PLAYER_SEEDED_TARGET** a redundance cost is added, to discourage oscilations (UP-DOWN-UP-DOWN) and (LEFT-RIGHT-LEFT-RIGHT).

5.1.1 Why Stochastic?

In order to be able to compete with LRTA* in speed, and keep K small, we need some hill climbing heuristics and some restart mechanisms.

The mechanisms are as such:

1. When all candidate states have reached an invalid state, we restart.

2. When a slalom tolerance is reached (i.e. the best state oscillates between two states for some amount of iterations), the temperature is increased.

3. When an amount slaloms happen we restart.

4. When no improvement is reached and the temperature is smaller than a threshold we restart.

Algorithm 4 Simulated Beam Search Solver

```
0: Initialize $queue$ with $(score(start), start, [])$
0: $temperature \leftarrow initialtemperature$
0: $iterations \leftarrow 0$
0: while True do
0:   Increment $iterations$
0:   if $iterations > max\_iterations$ then
0:     return best state from $queue$
0:   end if
0:   $candidates \leftarrow []$, $is\_improvement \leftarrow False$
0:   for each $(score, state, moves)$ in $queue$ do
0:     Sample next valid moves
0:     for each move do
0:       Compute new state and score
0:       if new score improves or random chance with $temperature$ then
0:         $is\_improvement \leftarrow True$
0:       end if
0:       Add to $candidates$
0:     end for
0:   end for
0:   if any candidate is solved then
0:     return solved candidate
0:   end if
0:   if not $is\_improvement$ or $candidates$ empty then
0:     return best state from $queue$
0:   end if
0:   Choose new $queue$ from best candidates
0:   Adjust $temperature$ based on slalom tolerance
0: end while=0
```

5.2 LRTA*

The LRTA* employed uses maximum 4 restarts and is a text book LRTA* implemented as in [UPBIA] and [AIMA]. Although 4 restarts are used, none are needed for most tests, the single exception being the **Large-2** which need one restart to learn the map more thoroughly.

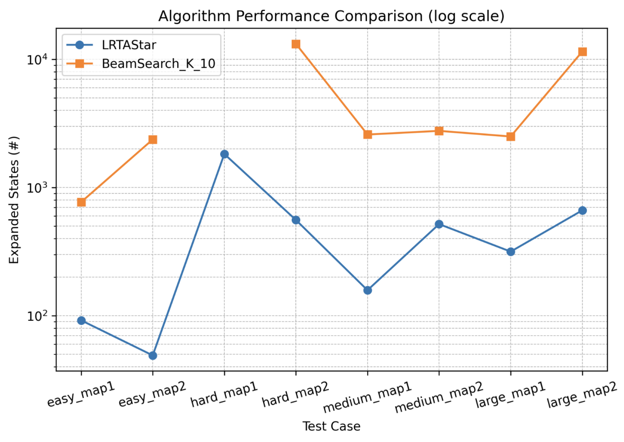

Figure 4: Logarithmic representation of the runtime of algorithms for LRTA* (dark circle) and K-Beam-search(10) (light square). LRTA* outperforms KBS by a factor of nearly 10.

6 Results

6.1 Heuristic Validation

The resulting SOKOBAN_PLAYER_SEEDED_TARGET solves almost all the test cases for both algorithms with 0 pulls. Because the heuristics were created for cases when 0 pulls are permitted, we present the following table showing the number of pulls for each test's found solution.

	E1	E2	H1	H2	L1	L2	M1	M2	SH
L*	0	0	0	0	0	0	0	0	-
KB	0	0	-	0	0	0	0	0	-

LRTA* managed to solve all the tests with 0 pulls except **SuperHard**, for which its runtime exceeded 5 minutes and was thus interrupted while **KBS(10)**[2] managed to solve all the tests with 0 pulls except **SuperHard** and **Hard1**, for which its runtime exceeded 5 minutes and was thus interrupted.

Similarly for other heuristics, not presented in the report or the code but nevertheless tested, the SH test was solved by a simple Manhattan distance heuristic with pull moves in way too many moves to be included in the code.

6.2 Runtime

Using a python3.11 environment, with a MACOS M1 and LRU Cache on heuristic functions we obtain the runtimes for the SOKOBAN_PLAYER_SEEDED_TARGET heuristic as shown in Figure 4.

[2]K-Beam-Seach for K=10

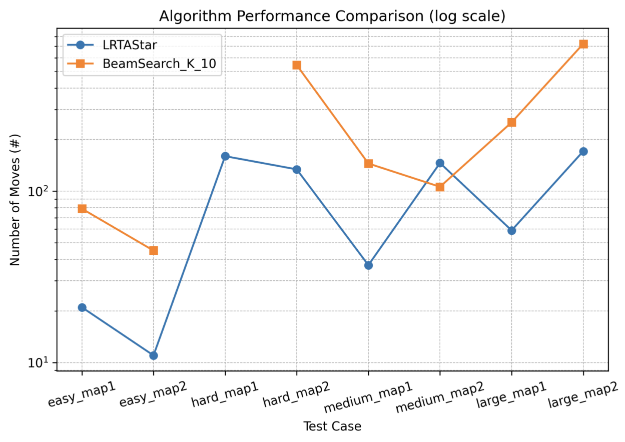

Figure 5: Logarithmic representation of the number of moves of algorithms for LRTA* (dark circle) and K-Beam-search(10) (light square). LRTA* has a move cap of 200 set in code, and uses no restarts for all tests with the exception of Large_2 which uses one restart.

6.3 Moves Number

The number of moves for the SOKOBAN_PLAYER_SEEDED_TARGET heuristic is as shown in Figure 5.

The number of restarts for the LRTA* algorithms is shown here:

	E1	E2	H1	H2	L1	L2	M1	M2
L*	0	0	0	0	0	1	0	0

6.3.1 Moves Number Analysis for KBS(10)

The maximum number of moves for KBS(10) is for the **Large_2** test which is 724 moves, second being **Hard_2** with 544. This discrepancy between KBS and the compactness of LRTA*[3], happens because KBS, is more likely to allow oscil;lations for small K. This is partially solved by stochasticity.

6.4 Expanded States

Every time we calculate a state's neighbors, we say we *expand* it. That being said, the less states an algorithm expands, the better and more precise it is[4].

The number of expanded states for the SOKOBAN_PLAYER_SEEDED_TARGET heuristic is shown in Figure 6.

6.5 Acknowledgements

This report was prepared as part of a project for the Artificial Intelligence class at POLITEHNICA University of Bucharest.

[3]LRTA* yields solutions under 200 moves for non super-hard all tests

[4]e.g. A* vs Dijkstra

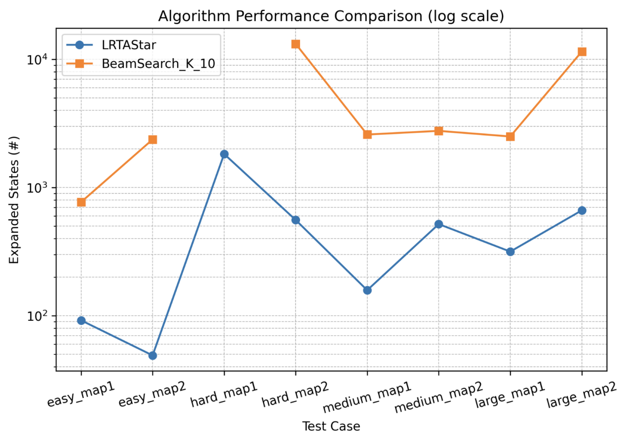

Figure 6: Expanded States for LRTA* (dark circle) and K-Beam-search(10) (light square). Hard-2 and Large-2 expand 13196 and 11487 respectively for KBS, but are only 663 and 558 respectively for LRTA*. By sheer luck, for Medium-2, LRTA* expands more states then the BeamSearch.

References

[AIMA] P. Norvig, S. Russel. LRTA*, K-BeamSearch *Artificial Intelligence - A Modern Approach (3rd Edition).*

[UPBIA] A. M. Florea LRTA*, K-BeamSearch *Artificial Intelligence Course (2025) - POLITEHNICA Bucharest.*

7 Appendix 1

```
0: function SokobanPlayerSeededTarget($map$)
0:   $player \leftarrow map.playerposition$
0:   $targets, boxes, obstacles \leftarrow mapdata$
0:   $freetargets \leftarrow targetsnotoccupiedbyboxes$
0:   if map is deadlocked then
0:     return large penalty
0:   end if
0:   function Solve($perm$)
0:     initialize cost, box_in_targets
0:     for all boxes do
0:       if box at correct target then
0:         increment box_in_targets
0:       else
0:         compute move cost via BFS and store box-target and player- box costs
0:       end if
0:     end for
0:     if all boxes placed then
0:       return 0
0:     else
0:       return reduce(box- target costs) + reduce(player- box costs)
0:     end if
```

```
    end function
    Initialize best score as $\infty$
    for all box-target permutations do
        compute score with Solve
        if score improves then
            update best score
        end if
    end for
    if map redundant then
        add small penalty
    end if
    return best score
end function=0
```