An Analysis and Implementation of Iterative Deepening A* Solvers for Sokoban

Joseph Nash, Anton Vasick, and Faraz Zaerpoor

December 6, 2016

Abstract. Sokoban is a classic puzzle with a high computational difficulty. In this paper, we analyze the effectiveness of applying the iterative deepening A* algorithm to finding solutions for arbitrary Sokoban configurations, as well as suggest optimizations to speed up this process.

1. Introduction

Sokoban is a single-player game taking place in a 2-dimensional grid populated by the player, walls, boxes, and storage locations. The player can make moves in any of the cardinal directions. The player cannot move into a wall. The player can move into a space occupied by a box by pushing the box one space in the same direction, given that its new location is not already occupied by a box or wall. The object of the puzzle is to make a sequence of moves such that all boxes are occupying storage locations.

2. Approach

Sokoban can be modeled as a graph by taking as nodes the positions of the player and boxes, and as edges the (legal) moves for the player. The initial node is the starting position of the game. The goal nodes are those where every box position corresponds to a storage location for that puzzle.

To solve Sokoban we implemented an iterative deepening A^* (IDA*) algorithm, which combines the iterative deepening depth-first search (DFS)

with the A* algorithms use of a heuristic function to increase efficiency by focusing first on the most promising nodes.

IDA* always finds the optimal (that is, lowest cost) path to a goal state when given an admissible heuristic function, so we experimented with various heuristics in order to find an effective balance between informedness and speed of calculation.

Our benchmarking system was a 2012 macbook air with 8GB of RAM and a 2.0GHz CPU. Our code ran in a single thread. We chose to implement our solver in Julia, a new language which is jit compiled to LLVM IR. We hoped the languages strong typing features, robust compiler backend, and friendly syntax would be an asset.

3. Algorithm Definition

(a) Iterative deepening A*

In A*, any node n has a value $f(n) = g(n) + h(n)$, where $g(n)$ is the cost to reach that node from the starting node, and $h(n)$, the heuristic function, is the estimated cost to reach a goal node.the depth limit is incremented and the DFS is run again.

IDA* runs a DFS from a starting node, but rather than halting at nodes which exceed given search depth, it halts when the node's f value exceeds a given cost. If no goal node is found at this cost depth, the maximum cost is incremented and the DFS is run again.

```
node              current node
g                 the cost to reach current node
f                 estimated cost of the cheapest path (root..node..goal)
h(node)           estimated cost of the cheapest path (node..goal)
cost(node, succ)  step cost function
is_goal(node)     goal test
successors(node)  node expanding function

procedure ida_star(root)
  bound := h(root)
  loop
    t := search(root, 0, bound)
    if t = FOUND then return bound
```

```
    if t =  then return NOT_FOUND
    bound := t
  end loop
end procedure

function search(node, g, bound)
  f := g + h(node)
  if f > bound then return f
  if is_goal(node) then return FOUND
  min :=
  for succ in successors(node) do
    t := search(succ, g + cost(node, succ), bound)
    if t = FOUND then return FOUND
    if t < min then min := t
  end for
  return min
end function
```

(b) Heuristics

Our heuristics are designed to give lower values for more desirable states. If a heuristic evaluates to 0, then the board is solved.

The first heuristic we used was an oversimplified heuristic, we call the **on switches heuristic** heuristic. It was computed based on how many boxes were currently on switches, $h() = boxesnotonswitches$. The more boxes not on switches, the higher the h value. This heuristic did not change very often, and offered very little difference between states. However, due to the simplicity, it calculated quickly.

We also tried a modified version of this naive heuristic which took into account the distance from the guy to the closest box not on a goal, a **closest switch heuristic**. In this case, $h() = boxesnotonswitches + distanceofguyfromnearestbox$. This heuristic was slightly smarter because it could help avoid cycles. With the completely naive approach, assuming one switch and one box, the guy walking towards the box and away from the box would always evaluate to the same thing. However, towards the box will almost always be a better move than moving away. This would also evaluate walking in a circle as neutral instead of poor.

For a more accurate heuristic we used a variant of the **Gale-Shapley**

Algorithm for finding stable pairings of two sets. For Sokoban we treated the boxes and the switches as the two sets. Preference lists needs to be made for each element of each set, that is the order from best to worst of the items of the other set to be paired with. The lists were made based on the distance between a box and a switch. That is the order of preferences for a box were closest switch, followed be the next closest switch and so on. The same was true for the switches. From there we can use Gale-Shapley.

```
Gale-Shapely Algorithm
function stableMatching {
    Initialize all box  B and switch  S to free
    while  free box b who still has a switch s to pair with {
       s = first switch on bs list to whom b has not yet tried to pair
       if s is free
         (b, s) become paired
       else some pair (b', s) already exists
         if s prefers b to b'
            b' becomes free
           (b, s) become paired
         else
           (b', s) remain paired
    }
}
```

After the pairs were made, we computed the distance between each box and its paired switch. By summing these, we got our total h value.

We attempted a compromise with less accuracy, but a less complex algorithm. To do this we used distance from boxes to switches without the stable matching. That is we found the distance from each box to the nearest switch, and summed these distances. This meant that two boxes could calculate their distances from the same switch, even though only one box could be on that switch in the solution. This was faster because it was basically the same as building the preference lists in the Gale-Shapely Algorithm, and stopping before finding the matches.

```
function closestSwitch{
for each box b
     for each switch s
         distance = |b - s|
```

```
            if distance < shortest
             shortest = distance
        h = h + shortest
}
```

We partially implemented a heuristic suggested by [1]. This heuristic is called the **minmatching** heuristic. We consider all mappings between boxes and switches. For each box/switch pair in a mapping, solve a reduced search with only one box and switch. The heuristic value for a state is the sum of the reduced search pair solutions in the mapping which minimizes this value. This is effect calculates a minimum bound for the steps to any solution from the state. This heuristic is admissible since it never over-estimates distance to the solution. Since we know the true destination switch of every box, it is also consistent.

(c) Other Optimization

We also modified our search to speed up our times. The first optimization was to avoid processing repeated states. This was done using a hash table of states we have seen before. Klavik [2] suggested the following hash function, which we implemented. This optimization also avoids cycles, for example a series of U, L, D, R without moving a box. The hash function is based on a matrix of large numbers, indexed by the location of the guy and the boxes. This creates a unique value for any board state, which is then moduloed by a large prime.

Another optimization comes from the realization that there are impossible states in sokoban, such as pushing a box into a wall in such a way it can no longer be pushed to a switch. While a heuristic may still evaluate this as a good state i.e. it may move the box closer to a goal, there is no reason to further expand this node because it can never reach a solution. We do this by treating squares that will cause a deadlock as possible bad squares so searches involving this square are pruned.

4. Algorithm Analysis

(a) IDA*

IDA* is, in terms of time complexity, nearly equivalent to an A* search, since it will expand all of the same nodes out to a given depth for each iteration. This gives it $O(b^d)$ time complexity, where b is the branching factor of the graph (in our case, less than 4) and d is the depth of the solution.

Space complexity is where IDA* has its advantage over A*; since it runs a DFS over the search space out to a given depth entirely separately for each iteration, the only nodes stored are those on the path from the starting node to the current node targeted for expansion, and the children of those nodes. Thus, it has $O(bd)$ space complexity.

Just as in A*, if the heuristic function used in the algorithm is admissible, IDA* will return an optimal solution (if one exists). The path that A* finds to a goal node has an estimate lower estimate than that of any other paths, as since A* opens nodes in order of increasing cost estimate, subsequent explorations of a goal node must have a higher estimate.

Since these estimates are optimistic, meaning they are never higher than the actual cost to the goal node, we can be sure of taking the shortest path. Since IDA* first explores nodes in the same order as A*, given that we are expanding the depth by the lowest possible cost, it will render the same optimal result.

(b) Heuristics

Our heuristic complexity is based on the number of boxes and switches which are both n.

Our naive heuristic, the **on switches heuristic**, was the lowest cost. Because it only required to check each box once assuming the switches were stored with an $O(1)$ look-up time, it could be calculated in $O(n)$. In this case the coefficient of n will be one. This heuristic is admissable as the number of unsolved boxes will always be less than the number of remaining moves.

The **modified naive** heuristic, in which the distance from guy to box, also can be done in $O(n)$ time, however in this case the coefficient will be two. One check of each box with the switches, identical to the naive heuristic, and one check of the guy against each box. This heuristic is also admissable as the guy will always atleast move to a box to finish the puzzle.

The **modified Gale-Shapley** heuristic is more expensive than the first two. Gale-Shapley is an $O(n^2)$ algorithm. In addition we need to generate the preference lists which will also be $O(n^2)$. Therefore this does not change the complexity, however it does increase the time it takes to run. This is not admissable, as it is possible to over estimate if the pairings are incorrect. For example, the closest switch is not the correct switch for a given box.

The **closest switch heuristic** is also $O(n^2)$. However it is equivalent to calculating the preference lists in the Gale-Shepley heuristic, so it's runtime is faster than the Gale-Shapley algorithm.

A downside of the **minmatching** heuristic is it is very expensive. To solve even the 1 box, 1 switch problem we call our IDA* solver. For boards the size of sokoban2 and larger this evaluation is too slow. To alleviate some of this cost, we randomly generated mappings between states and boxes instead of exhaustively testing all of them. It also requires an additional 'bootstrap' heuristic to use for the 1 box/ 1 switch search. We chose the closest switch heuristic for this. Unfortunately, we ran into implementation issues with this heuristic. It erroneously reports heuristic values which are too high, so we fail the admissible criteria, and the solver does not operate functionally.

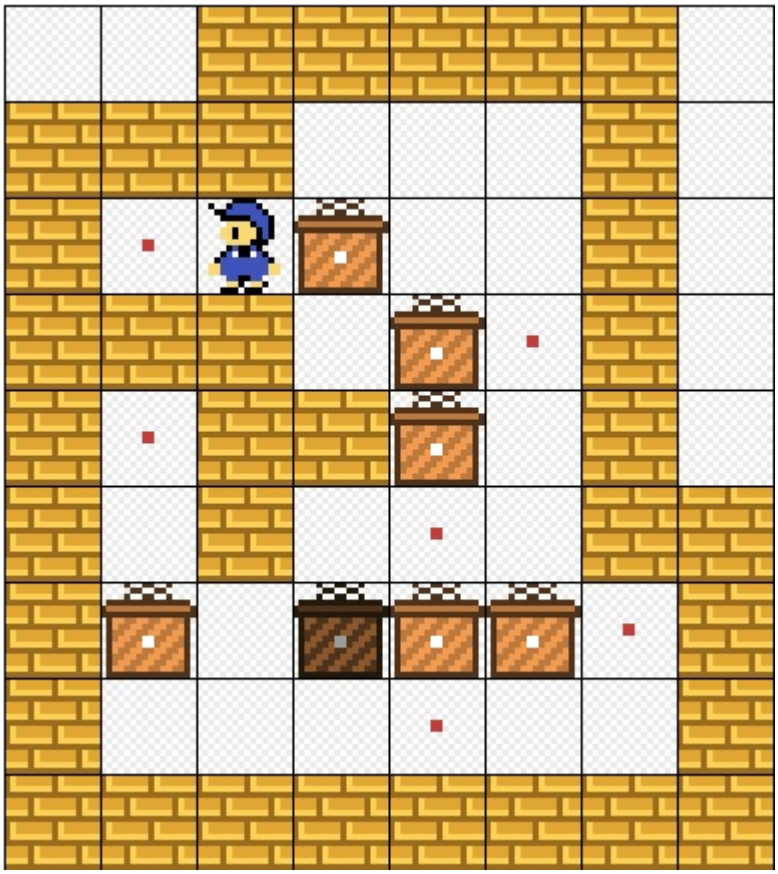
5. Results

Table 1: Runtime results for IDA* with hashing and closest switch heuristic. Precomputation is time spent finding deadlock positions. dnr indicates computation took too long to halt. wikisoko was solved with the modified naive heuristic in 49.4 seconds

Board Name	Precomputation	Total Runtime
sokoban0.txt	0.533839	0.60842 sec
sokoban1.txt	35.29168	dnr (>1 hour)
sokoban2.txt	dnr	dnr
sokoban3.txt	dnr	dnr
sokoban4.txt	dnr	dnr
sokoban5.txt	dnr	dnr
wikisoko.txt	19.84415	113.6* sec

Sokoban2 reached a depth cutoff of 63 after 1 hour of computation. We

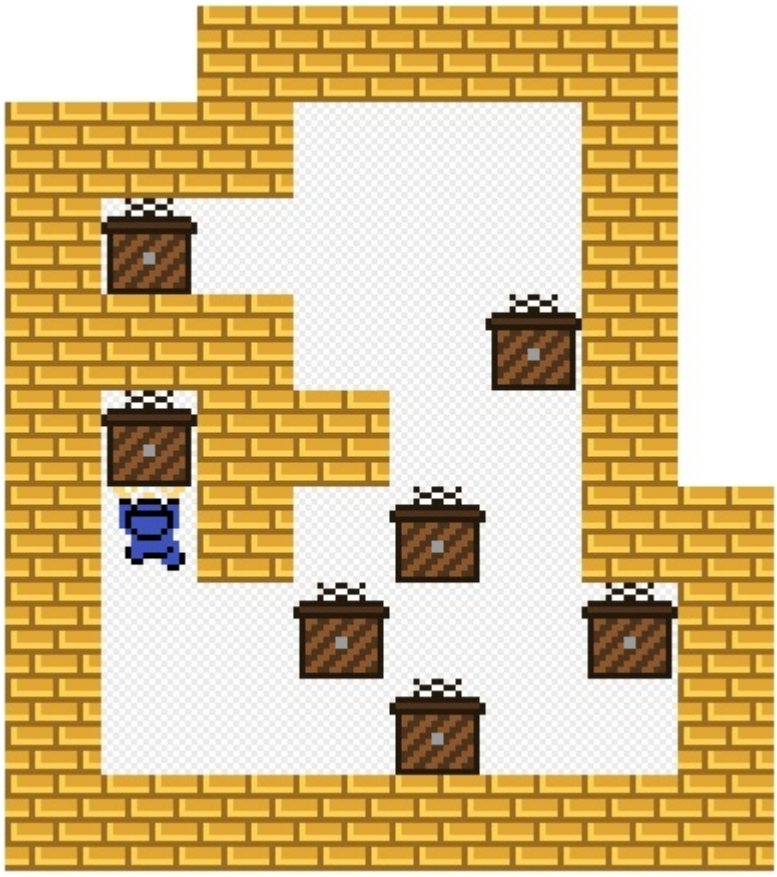
could not find a solution to benchmarks 2-5. We made another puzzle based on the sokoban example on wikipedia. This puzzle is 8x9.



In 49.4 seconds, our program generated the length-34 move list below, with each character corresponding to a player movement:

```
['R','U','R','R','D','D','D','D','L','D','R','U','U','U','U','L','L',
 'L','R','D','R','D','R','D','D','L','L','D','L','L','U','U','D','R']
```

Applying this move list to the original state renders a completed game:



6. Conclusions

We found that our IDA* solver with modified naive heuristic heuristic and visited state hashing was sufficient to solve puzzles of about size 9x9. On puzzles larger than this size, the search space grows too large for our solver. The data structure choice, heuristic choice and hashing gave significant speedups to our algorithm, but more domain specific features are need to solve larger boards.

References

[1] L. Rei and R. Teixeira, "Willy-a-sokoban-solving-agent,"

[2] P. Klavik, "Sokoban solvera short documentation,"