Sokoban: Search in a complex domain

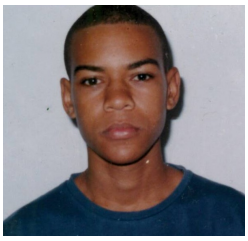
Yann Chazallon, Nicolas Dossou-Gbété, Tony Chan Ki Hong and Michal Staniaszek

October 18, 2013

Abstract

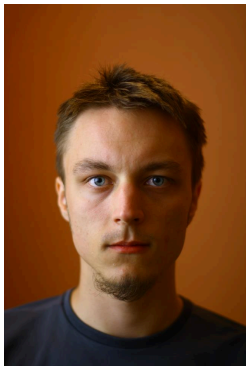
In this report we describe our implementation of a *Sokoban* solver. We use a two stage method to find the solution. A bi-directional best-first search with manhattan distance as a heuristic is used to find the path of boxes on the map. This disjoint path is then linked by using A* to find the path between successive box moves. The solver is currently able to solve 47 of the 100 evaluation maps on Kattis, and 81 of the 100 maps in the simpler test set within the 11 second time limit.

Yann Chazallon
91/01/16



Nicolas Dossou-Gbété
89/09/26

Tony Chan Ki Hong
92/07/25

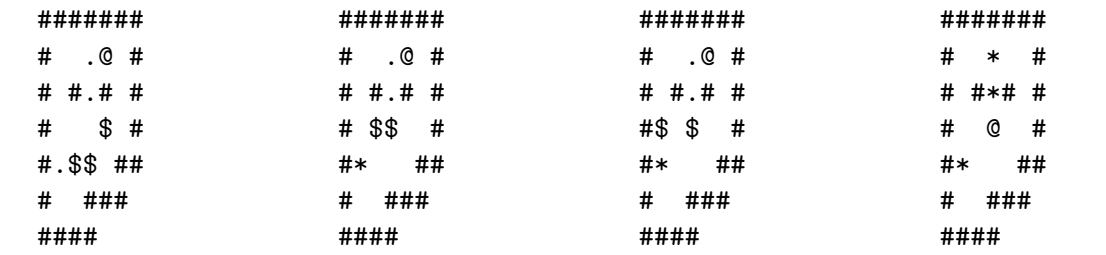


Michal Staniaszek
90/12/07

1 Introduction

Sokoban is a puzzle game which was written in 1981 by Hiroyuki Imabayashi. First published in 1982, it is now a very popular game, with many clones available on the internet. The player controls a warehouse keeper (for which *sokoban* is the Japanese word), whose job it is to push boxes onto goal locations on the map. The player can move in four directions (up, down, left or right) on the map, which is split into discrete cells. The player can push boxes, but is unable to pull them. To be able to push a box, the player must be adjacent to it, and there must be an empty space behind the box into which it can be pushed. Only one box can be moved at a time; if two boxes are contacting each other, pushing one box does not move the other. While there are many graphical implementations, there are also many text-based implementations which use symbols for representing parts of the board, an example of which is shown in Figure 1.

The game has garnered some interest in the artificial intelligence (AI) community due to the difficulty of finding automatic solutions. Games are an ideal platform for the development and testing of AI techniques, as game environments are much simpler and less (or not at all) noisy, and also have sets of simple rules for interaction with the world. Working in such environments



(a) A basic map (b) Half solved (c) In an unsolvable state (d) The solved map

Figure 1: A typical *Sokoban* map. The player is represented by `@`, or `+` when on a goal, boxes by `$` or `*` on a goal, walls by `#`, goals by a period, and empty space by a blank. The map in (c) is unsolvable, as the two boxes on the left are both against a wall. This is called a *deadlock*.

allows for more control and closer investigation of the relevant parts of the problem that is being considered. *Sokoban* in particular is an interesting problem due to its high branching factor, and the depth of the search trees that are generated when attempting to solve a problem. Even relatively simple problems can take upwards of 100 moves to solve, and more complex problems can exceed 500 moves, even in the optimal case. Additionally, it is possible to extend the search tree indefinitely—there is no time limit, and boxes can be pushed without restraint, so long as the push is valid. The branching factor is the total number of valid actions that can be applied to any box reachable from the player's current position. In the worst case, if there are N boxes, all of which are accessible and can be pushed in any direction, the factor is $4N$ (each box can be pushed in 4 directions). Although in practice it is not possible to access all boxes, or push them in an arbitrary direction, but most maps have a large enough number of boxes that the branching factor has a large impact on the number of states expanded.

2 Method

The solver should be able to read string-based map representations into a manipulable object. It must be possible to move boxes and the player on this map representation. In addition, the solver must be able to deal with both static and dynamic deadlocks. A heuristic is required to drive the search towards the goal. Finally, a search method is required, which, given a start and goal state, must be able to find a path between the states, or indicate that a path does not exist.

2.1 Map Representation

To be able to solve a *Sokoban* map, it is necessary to have a map representation which can be manipulated and used to gain information about the state of the board. The representation of the state is required by any search method in order to be able to generate successor states and find a path to the goal state. Our map representation uses a two-level model, where dynamic objects (boxes and the player) are separated from the static part of the map, that is, the goals, empty space, and walls. By separating these two parts of the map, we are able to reduce the storage space required for each board, as the static map is represented only once. When move validity must be checked, the dynamic objects are "collapsed" onto the static map.

The static map also stores a costmap which relates each position on the board to a set of costs to reach each goal. The costmap is calculated by doing a flood fill which attempts to pull a single box from each goal to all points on the map, with no other boxes present. This gives an optimistic estimate of the number of steps required to move a box from any position on the

map to any goal when the map is populated with boxes, as in practice the path of the box may be obstructed by other boxes.

Functionality for checking board equality is also required to be able to perform search, as it is necessary to check whether states have been visited previously in order to prevent unnecessary re-evaluation of states. Our map representation has different types of equality based on the state expansion procedure being used.

The state expansion defines how successors of a given state are generated. The player space expansion is based on the motion of the player alone; the player can move up, down, left or right, and so the maximum number of successors is 4. In this expansion, exact equality is used—the position of all dynamic objects on the board is checked, and the player and all boxes must be in exactly the same locations for the boards to be equal.

The board space expansion is based on the motion of all boxes in the state. With this expansion, the maximum number of successors is $4M$, where M is the number of boxes which can be reached by the player, as each box could be pushed in one of four possible directions. To determine the successor states of this state, it is first necessary to find all the positions which are accessible by the player. This is done by starting a flood fill from the current player position and finding points at which a box can be accessed. With this expansion, the board equality no longer checks the exact location of the player. Instead, the top leftmost position accessible by the player is used to represent the player. This means that states in which the player has the same accessible area, and where boxes are in the same positions are considered equal.

The implementation of the board space equality check was done using string hashes. Each board generates two strings at its creation, one including the player, and one without, to be used for goal checks. This hash can be seen as a linear rather than grid based representation of the map, but having only the boxes and player's position, and whitespaces instead of the other elements. We chose this implementation because their generation and comparison is fast and trivial, and they can also be used in hashtables, delegating the repeated state detection to the Java API rather than a heavier custom method.

2.2 Deadlock Detection

A deadlocked state is a state from which further expansions can never give a solution. If they are detected early, they can be removed from the search tree, pruning big sections of it and making the game much easier to solve. We identified two classes of deadlocks:

2.2.1 Static deadlock

A static deadlock is a deadlock caused by the positioning of a box relative to the walls of the map. For example, pushing a box into a corner, or against a wall which ends in two corners makes it impossible for the box to be moved away from that position.

We detect them using the static costmap, created and filled at the launch of the program. The positions for which we do not have a cost are those out of reach of the goals, and as a result those where a box will be deadlocked.

2.2.2 Dynamic deadlock

Dynamic deadlocks are created by the positioning of a box relative to all the others. They have many possible manifestations and thus are much harder to detect. While some implementations rely on matching a dictionary of known locked patterns, our approach is more generic and logical.

Since this algorithm is used for pruning search branches, it is important to avoid false positives that could lead to losing solution states. The basic idea is that every box has two

degrees of freedom, horizontal and vertical. If any of the boxes loses it's two degrees of freedom permanently then it is considered locked. Expanding a state containing such a box is pointless. We consider a degree of freedom to be locked if one of its sides is occupied, because we need to have a receiving space for the box and a space for the player to push from at the same time. For example, a box is horizontally locked if its left or right side are occupied. A wall results in a locked degree of freedom without any other consideration. In the case where the locking is caused by other boxes, it is necessary to consider whether these boxes are locked themselves. We can use the same criteria recursively on these locking boxes until we find a free box. A box is considered free as soon as it has a free degree of freedom. Since this describes a cyclic graph, we need to use a closed list to avoid infinite recursion. This is of course relevant to the algorithm, as box A cannot unlock box B if the freedom of B freedom depends on that of A.

Something that must be considered when using this algorithm is that a box dynamically locked on a goal should not trigger the pruning of the state for obvious reasons. However, these boxes must still be considered locked in order for the recursion to work for nearby boxes.

One last thing to keep in mind is that this algorithm is not suitable for backwards expansion, as it is highly reliant on the dynamic aspects and constraints of *pushing* boxes. The implementation is given in Algorithm 3.

2.3 Heuristics

2.3.1 Manhattan Distance

The Manhattan distance d between two positions a and b in a grid is the number of junctions in the grid which must be crossed to reach one from the other by using only vertical or horizontal movements, computed by

$$d(a, b) = |x_b - x_a| + |y_b - y_a|$$

Our heuristic simply returns the sum of the distances to the closest goal for each box, using the above formula.

2.3.2 Static Cost Evaluation

At the launch of the program, when initializing the static map, we compute the real cost from each goal to any reachable point of the map, by using a flood fill. This is used to look for unreachable points (for locked state detection), but also to obtain a better cost estimation for the heuristic, as the boxes must take valid trajectories of motion and cannot ignore the presence of walls. As with the manhattan distance heuristic, this heuristic returns the sum of the distances to the closest goal, but with a better approximation of these distances this time.

2.3.3 Min Matching

This heuristic tries to match each box to a goal before estimating the distance between them.

We want to find an assignment of boxes to goals such that the sum of the distances between them is minimised. A good solution to this problem can be obtained using the Hungarian method. But it also requires recomputation of the distances from the boxes to the goals, because the boxes can block each other's paths, which can necessitate path modifications and even make some goals unreachable.

We tried a quick and dirty alternative based on the static costmap, that would avoid having to do heavy calculations. We ordered the boxes and goals by number of possible matches and attempted to make pair them in that order, returning $+\infty$ if the matching failed.

Early results were not convincing and the results with the other heuristics were better than expected, so we abandoned further development of this heuristic.

2.4 Search Method

The search methods form the core of the solver. Using them, we find a path between a starting board state and the goal state, where all boxes are on goals. We implemented three different search methods; A*, best-first search and bi-directional search. A* expands nodes in a specific order based on the value of a state evaluation function $f(s) = g(s) + h(s)$, where s is some board state. The value $h(s)$ is the heuristic value of the state, and represents an estimate of how far the state s is from the goal. The value of $g(s)$ depends on the path that was taken to reach this node—the longer the path, the higher the value. A* places states into a priority queue and expands them starting from nodes with a lower f value; the lowest estimated total cost to a solution. This evaluation function means that so long as $h(s)$ is an underestimate (an *admissible* heuristic) of the true cost to the goal state, A* is both optimal and complete, that is, it will always find a solution if one exists, and that solution will be optimal[2].

In best-first search, the cost to reach the current node is not considered in the expansion, and so the evaluation function simplifies to $f(s) = h(s)$. This results in the search always expanding the state with the lowest cost to the goal, regardless of how far it is from the start state. This results in a loss of optimality and completeness [2], but can lead to a goal being found much faster than in the case of A*. Since we do not care about finding an optimal solution, and speed is of the essence in this problem, best-first search is more appropriate than A*. Our implementation is shown in Algorithm 1.

To further improve the speed of the search, we implemented a bi-directional search. While both A* and best-first search have a complexity of $\mathcal{O}(b^d)$, running two searches, one from the start state to the goal, and one from the goal state to the start, reduces this to $\mathcal{O}(b^{d/2})$ [2], where b is the branching factor, and d is the solution depth. The search terminates when the frontiers of the two searches meet. This can be understood intuitively by considering a circle centred on the front node, with a radius d, with the goal node on the edge of the circle. The area inside this circle is very large compared to two smaller circles centred on the start and the goal meeting in the middle.

One could think that searching backwards consists only of inverting start and goal states, and using a different expansion method that pulls boxes instead of pushing them. In actual fact, when using backwards search we need to consider *multiple* start states, as we do not know what area the player should be in after solving the map. Another detail to consider is that the algorithm we use to check for deadlocks are not usable as is. The cost map for static checks must be filled based on the reachability of box starting positions instead of goals, while the dynamic algorithm is completely irrelevant, as boxes can be pulled out of deadlocked states. Our implementation of bi-directional search is shown in Algorithm 2.

Our approach involves splitting the problem into two distinct search problems. First, we find the actions required to move the boxes from the start state to the goal state using a best-first bi-directional search. Once this intermediate path has been found, we run a series of A* searches to find the path of the player between each successive box movement, which gives us the final path from the start to the goal. When finding the player path, we consider boxes as immovable objects.

Algorithm 1: Best-first search

```
input  : An open list O, containing a start node S. A goal node G. A closed list C.
output: A list of action, point tuples (A, P), containing the path from S to G.
Function findPath()
    while O ≠ ∅ do
        successors = step()
        if G ∈ successors then return successors.get(G).unwind()
    end
end
Function step()
    front = open.front()
    successors = front.expand()
    closed.add(front)
    foreach n ∈ successors do
        if n ∉ C ∧ n ∉ O then
            // nodes in O sorted by heuristic value
            n.costEstimate = h(n.state, G)
            O.add(n)
        end
    end
    return successors
end
```

3 Implementation

In our implementation, we use several objects which allow for cleaner code and easier access to required data. The most basic part of the implementation is the `Board` class, which holds methods to deal with moving the player and accessing various information about the state of the board. The `Board` object holds a list of dynamic points, as well as a reference to the static map. The static map is implemented as a singleton, so all objects created have a reference to that same object. This reduces memory usage, as the static part of the map is not duplicated for each `Board` object.

As we are using bi-directional search, the search methods have been implemented in a way that allows their usage in both a bi-directional and a single directional search. This is important as we do not perform only a single search to find the path to the goal. To achieve this, we use a `step` method to perform a single step of the search. This step, in the case of A* and best-first search, is to expand the front of the priority queue, and add the successors to the open lists, performing any relevant checks. The best-first search implementation is shown in Algorithm 1. Algorithm 2 shows our implementation of bi-directional search, which makes use of best-first search.

The search methods make use of a custom `SearchNode` class, which stores a parent node, a `Board` representing the state of that node, and a `BoardAction` object, which represents the action used to generate the board by a point object (the location of the pushed box) and an associated `Action` enum, which is one of $\{up, down, left, right\}$, and indicates the direction of the push. The `unwind` method of the `SearchNode` recurses up to the start node, and reconstructs the path from there to the node on which the method was called.

Algorithm 2: Bi-directional search

Data: Search implementations forwards and backwards, initialised with start and goal.
Result: A list of tuples of (`point`, `int`) containing motion of boxes from start to goal.
begin
  current ⟵ forwards
  opposite ⟵ backwards
  new ⟵ ∅
  key ⟵ `null`
  repeat
    swap current and opposite
    new ⟵ current.step()
    if *node* ∈ *new* ∧ *node* ∈ *opposite.open* **then** key ⟵ node
  until *key* ≠ ***null***
  fkey ⟵ forwards.open.get(key)
  bkey ⟵ backwards.open.get(key)
  fActions ⟵ fkey.unwind()
  bActions ⟵ bkey.unwind()
  return merge(fActions, bActions.reverse())
end

Algorithm 3: Single box dynamic lock detection

input : box position B.
output: boolean representing whether or not the box at point B is locked.
Function *isLocked(Point B)*
  if *B is on goal* **then**
    return false
  end
  closed ⟵ {∅}
  return ¬ isMovable(B, *closed*)
end
Function *isMovable(Point P, Set closed)*
  if $P =$ *WALL* **then**
    return false
  else if $P =$ *BOX* **then**
    if $P \in$ *closed* **then**
      return false
    else
      closed.add(P)
      return ((isMovable(P.left, closed) ∧ (isMovable(P.right, closed)) ∨
      ((isMovable(P.up, closed) ∧ (isMovable(P.down, closed))
    end
  else
    return true `// free space`
  end
end

	Time limit		
Search Method	5 sec	11 sec	15 sec
A*	12	15	16
Best First	56	60	64
Bi-directional Best First	76	81	82
Bi-directional A*	39	41	43

Table 1: The number of test cases solved out of 100 available in the offline test set, running various search implementations with the manhattan distance heuristic. Note the lack of large differences in the number of solved maps with different time limits.

4 Evaluation

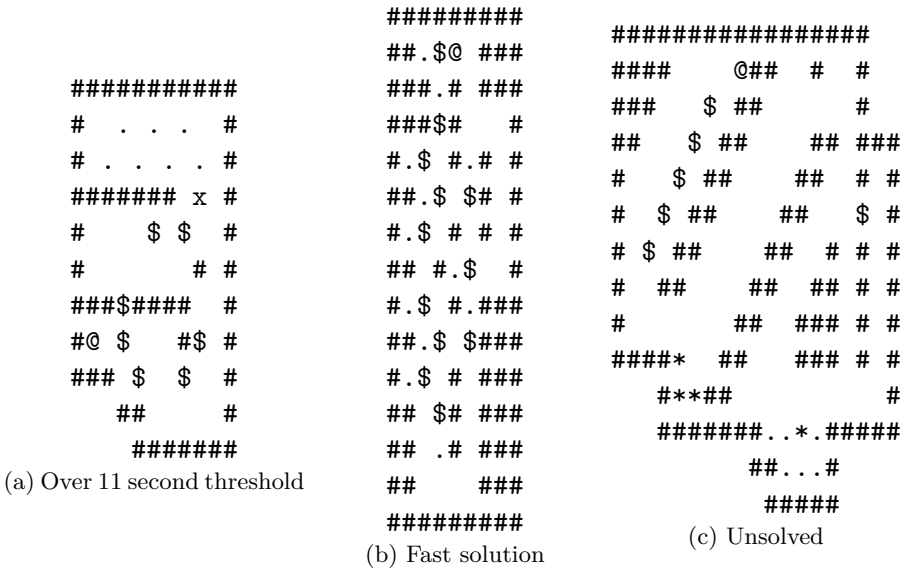
We evaluated our solver by running it on the 100 test maps provided. Table 1 shows the number of maps successfully solved by different search implementations using the manhattan distance as a heuristic. Clearly, implementations with best-first search give the best results. Surprisingly, the single direction best-first search solves more maps in 5 seconds than bi-directional A* does in 15. This is most likely due to best-first search rapidly reaching the solution depth, whereas A* takes a long time to reach an equivalent depth due to its checking many possible paths through the space. Forwards A* performs badly, solving only 16 maps. Interestingly, all of the searches have very little change in terms of the required time to a solution—if the map can be solved at all by the search, then it will be solved very quickly. This is most likely due to the search getting stuck in certain situations which result in very long unfruitful paths due to insufficient pruning of useless states.

Table 2 shows more detailed information about each of the experiments. The number of nodes opened in total by the bi-directional searches is approximately half the number expanded by the single direction searches. Average solution times for the bi-directional searches were also better than their single direction counterparts. The average number of nodes expanded per second was higher for the best-first searches, with 5370 per second for single direction, and 3800 for bi-directional, compared to 2700 and 2300 for the A* implementations. The reduction in total nodes expanded results in increased speed for the bi-directional searches.

While our solver can solve some maps very quickly, it has problems with others. Figure **??** shows some examples of various maps. Map 54 of the test set causes a problem due to the need to place a box in a key position very close to a goal, and not move it right until the end. This map is a border case which can be solved within 15 seconds, but not within 11. Map 66 is very easily solved, as there are not many box movements to do. Each box, with the exception of one requires only a single movement to push it onto a goal and generate a valid solution. As we are using a box movement expansion, this makes the number of states the search must check very limited. Map 93 is not solved within 15 seconds. This is due to a goal area in an area of the map which must be bypassed to solve the problem. We do not match boxes to goals, and so the boxes are pushed towards the intermediate goal area, and not moved out until absolutely necessary because they must be moved away from the goal, increasing the heuristic cost, to do so. The boxes on the right side of the map must also be moved onto the goals to allow the player to push boxes onto the goals. In addition, this is problematic for the reverse solver as it will attempt to remove the boxes on the intermediate positions even though this is unnecessary.

Search method	Avg expanded	Avg opened	Avg rejected	Avg time
A*	9149	33382	17929	3.4
Best First	10741	40473	19499	2.0
Bi-directional Best First	3801	12727	4136	1.0
Bi-directional A*	4591	16462	4340	2.0

Table 2: Some statistics for the search nodes, and the average solution time in the tests in Table 1. Expanded nodes are those nodes whose successors have been generated. Opened nodes is the total number of nodes placed onto the open list. Rejected nodes are successors not added to the open list because they would generate locked states.



(a) Over 11 second threshold

(b) Fast solution

(c) Unsolved

Figure 2: Examples of edge cases for our solver. Maps 54 (a), 66 (b) and 93 (c) from the test set. (a) is a border case, solvable within 15 seconds, but not 11. `x` indicates a critical location. (b) is solved in only 0.09 seconds. (c) is not solved even with a 15 second time limit.

5 Discussion

During our first meeting, we discussed several ideas about how to solve the problem, based on brief readings of some previous work [3, 1]. We decided that in the ideal case we should attempt to implement some of the techniques mentioned in these papers to produce an effective solver. The first step we took was to create the foundations for the solver, by implementing a board representation and some search algorithms, which we knew we would need based on our discussions. Work was distributed by each group member choosing something to work on, and then working on the parts of the solver related to that. This worked reasonably well, but meant that some members did not fully understand the code that was being used, because they had no part in writing it. We attempted to amend this by commenting methods with enough information to understand what the method was *supposed* to do. In order to check that important methods were doing what they were supposed to be doing, we used JUnit tests, running them whenever there were significant changes to the code to ensure that everything was working as expected.

Our approach to developing the solver led to some problems. Because each member was working on a somewhat different task, we did not meet directly very often, which resulted in too little communication and only a vague idea of what we were aiming for. As such, we spent the first couple of weeks adding unnecessary code to the solver, which would be removed later. This left us with a solver which, at the milestone deadline, could solve the four basic maps, but nothing else. Once we started meeting more often, we were able to make progress on the solver, but most of the progress was fixing errors that were caused by misunderstandings of the intended use of methods, and various other bugs which could not be directly detected by the unit tests. Fixing these problems left us with little time to consider any more complex techniques that we could use to improve the solver, which negatively impacted on the final result.

Our first approach to the problem was to use the motion of the player as the state expansion in an A* search implementation. This solver was able to complete very simple maps, but the size of the search space was far too large to solve any of the maps in the test set. Our second attempt was to use an expansion which considered all valid moves of boxes in a given board state. This reduced the search space to only those moves which were relevant to solving the problem. Once all bugs had been fixed, this version of the solver could solve a small number of maps in the test set, but continued to be hampered by slowness. This version was improved by using a more efficient equality check based on a string representation of the board, and a best-first search implementation. With this, we were able to solve approximately 15 of the maps from the online test set. Our current approach was reached by our need to improve the speed of the algorithm, and we thought that rather than using improved heuristics, changing the search algorithm would be more effective. We implemented a minimum matching heuristic, but the additional complexity resulted in the search taking longer despite the improved heuristic estimates. The performance measure used during the process was the number of maps of the Kattis test set solved by the solver. Profiling was also used to judge the effects of more efficient implementations of parts of the solver.

References

[1] Andreas Junghanns and Jonathan Schaeffer. “Sokoban: Enhancing general single-agent search methods using domain knowledge”. In: *Artificial Intelligence* 129.1 (2001), pp. 219–251.

[2] S.J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall series in artificial intelligence. Prentice Hall, 2010.

[3] Timo Virkkala. “Solving Sokoban”. MA thesis. University of Helsinki, 2011.