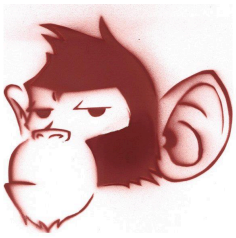
Sokoban: Search in a complex domain

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

October 17, 2013

Abstract

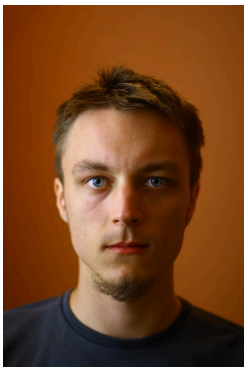
The aims of the project is to develop an agent that can solve Sokoban. By now, 33 out of 100 sokoban maps have been solved with 12 in Sokoban 1, 6 in Sokoban 2, 3 in Sokoban 3, 10 in Sokoban 4, 2 in Sokoban 5.



Yann Chazallon
91/01/16

Nicolas Dossou-Gbété
89/09/26

Tony Chan Ki Hong
/07/25

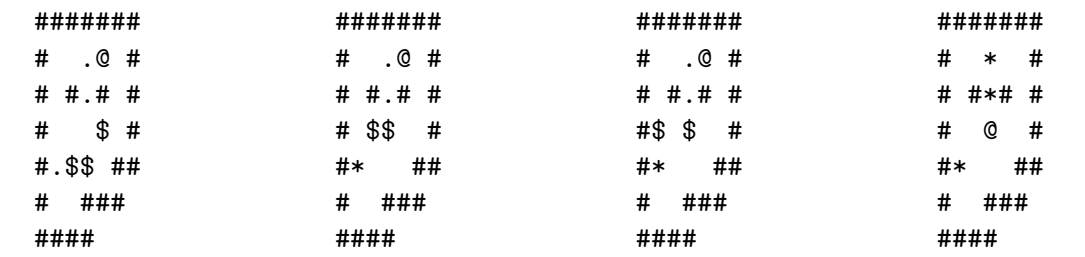


Michal Staniaszek
90/12/07

1 Introduction

Sokoban is a puzzle game which 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 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



(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*.

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

Concept to tackle the problem

1. Read and Store the Map 2. LockState Detection 3. Use Search Method to find solution 4. Use Heuristic to evaluate node by giving value 5. Expand node with highest Heuristic value first 6. Find out and Output Result

2.1 Map Representation

According to the requirement, the map is recieved as a txt file with symbols representing different objects. They can be separated as two type of objects: Dynamic and Static. We have created two class "Board" and "Static Board" to store the objects respectively.

2.2 Deadlock Detection

The exists of Deadlock makes Sokoban even more hard to solve. As Player cannot pull a box, sometimes player may pushes the box into a position that the box can never reach the goal.

Static Lock

The meaning of static lock is that we cannot never push the box to the goal even without other boxes.

While recieving the map, the program will try to put a box on each goal and check the points that is reachable by the box by pull (i.e. do a BFS for each goal with reverse action). Reachable points are stored in a map for each goal. The cost for a box at Non-reachable points to a specific goal will be Positive infinity.

Dynamic Lock

In this case, the point of a box is supposed to be reachable to some specific goal. However, there are some boxes blocking in the middle and it is impossible to remove that blocking box.

2.3 Search Method

Two-stage search

2.4 Heuristics

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.

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 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
```

Figure 2: timing for experimental results

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
```

4 Evaluation

experimental evaluation and a thorough analysis of your results. This section should include: How well did your agent do on the different maps?

5 Discussion

During our first meeting, we discussed several ideas about how to solve the problem, based on brief readings of some previous work [2, 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 too 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. 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.

reflection section dealing with questions: How did you plan to solve the problem? How did you plan the work in the group? What other methods did you try out? What did work/ did not work? Why? How did you end up with the current approach? How did you measure the performance and success during the process? How would you solve it if you were asked to do it again given what you know now?

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