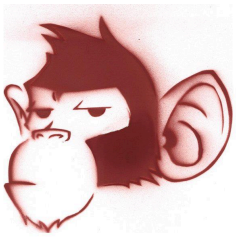
Sokoban: Search in a complex domain

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

October 16, 2013

Abstract

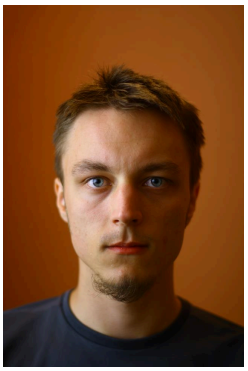
a brief summary of your project and obtained results



Yann Chazallon
91/01/16

Nicolas Dossou-Gbété
89/09/26

Tony Chan Ki Hong
/07/25

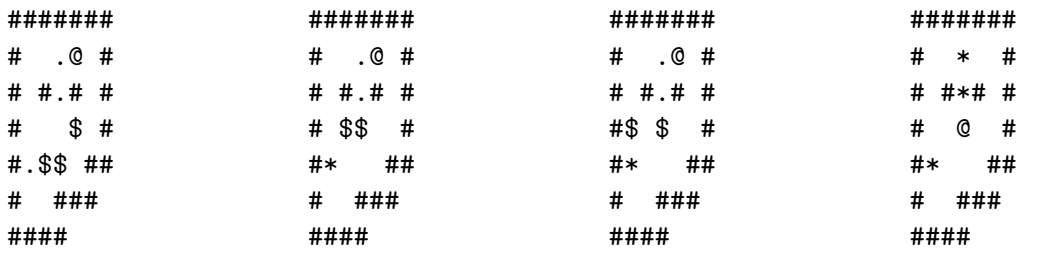


Michal Staniaszek
90/12/07

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 of 100 moves to solve, and more complex problems can exceed 500 moves, even in the optimal case.



(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*.

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

2.1 Map Representation

2.2 Deadlock Detection

a description of your approach to the problem and your method

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

```
findPath while open not empty do
    successors ⟵
    if goal in successors then return successors.get(goal).unwind()
end
step front ⟵ open.front()
successors ⟵ front.expand()
closed.add(front)
foreach node in successors do
    if node not in closed and node not in open then
        // open is a priority queue, nodes sorted by heuristic value
        node.costEstimate ⟵ h(node.state, goal)
        open.add(node)
    end
end
return successors
```

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
```

implementation details (if there are any important ones)

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

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?