# Sokoban Solver—a short documentation

Pavel Klavík

http://pavel.klavik.cz/projekty/solver.html

This is a short text describing technics used for solving Sokoban levels. Sokoban is a PSPACE-complete problem [1]. The main effort is to search the space somehow efficiently and to solve at least small levels. Most of the ideas are taken from the Junghanns's thesis [2]. We are also grateful the authors of [3], especially the correspondence with the author of the solver YASS was very useful.

A Sokoban level contains walls, boxes and goals. The goal is to place all the boxes on the goals. The storekeeper (called in Japanese Sokoban) can only push boxes forward and only one box at the same time.

Sokoban has many specific properties unlike other similar problems as Rubik's cube or Lloyd Fifteen Puzzle. Moves are irreversible which means a bad move can made the level unsolvable. Also, the number of different positions can be even for small level of size $20 \times 20$ approximately $10^{100}$. Moreover, it is non-trivial to find a good heuristic for distance of the position from the solution. All these properties makes Sokoban an interesting problem to solve.

## 1 How to use the solver

To compile the solver, use the makefile. To run the solver, use:

```
./solver input_file [output_file]
```

where `input_file` contains levels in the standard Sokoban format. For example, this is the first level from the original game:

```
    #####
    #   #
    #$  #
  ###  $##
  #  $ $ #
### # ## #   ######
#   # ## #####  ..#
# $  $          ..#
##### ### #@##  ..#
    #     #########
    #######
```

The parameter `output_file` is optional, used for writing solutions.

The input consists of the following characters:

- "`#`" is a wall,
- "` `" is a free space,
- "`$`" is a box,
- "`.`" is a goal place,
- "`*`" is a boxes placed on a goal,
- "`@`" is for Sokoban and
- "`+`" is for Sokoban on a goal.

Also, the file may contain more levels, separated by at least one empty line.

The output solution contains a line for every level. The solution is a string representing sequence of moves. The lowercase letters "`uldr`" are used for moves and the uppercase letters "`ULDR`" for pushes. For example, this is a solution of the level above.

```
ullluuuLUllDllddrRRRRRRRRRRRRlllllll
luuululldDDuullddrRRRRRRRRRRdrrrUldl
lulllllluuulLulDDDuullddrRRRllddrrrr
uurrrrrurDlllllldllllluurrRRRRRRRdRUl
lllllluuullluuurDDlldddrrruuuLLulDDDu
ullddrRRRRRRRRRRdrRlluuRDllllllllll
lulldRRRRRRRRRRRRRRRlllllllluuulluuul
DrddrrdddrrrrrrdRlullllllluuulluulDDDD
uullddrRRRRRRRRRR
```

A solution can be displayed using a simple tool `draw_solutions`. It has the following usage:

```
./draw_solutions level_file solution_file
```

## 2 Overview of the algorithm

The solver basically works in the following way. It searches the game tree of all the reachable positions. For each position, we generate all its children created by a push of one box. The number of the positions grows very quickly. To fasten the solving, we use the following technics:

**Hashtables.** To avoid searching a position several times, we store an information about it to a hash table. Before inserting a position to the game tree, we check whether it was not found before. The hash function is very simple. In the beginning, we assign a random number to every field. To calculate the hash key, we take the number of all the position with a box (+ some other number for a field with Sokoban) and xor all of these numbers. This gives a resulting index in the hash table modulo a big prime number.

**Distance heuristic.** Since we do not have enough time to explore the whole game tree, we would like to prefer the positions that are closer to the goal. As a simple heuristic, we sum the distances of all the boxes for their closest goals. Notice that this is a lower bound for the distance, we cannot solve the level faster. On the other hand, it is not likely that the mapping of boxes to the closest goal field is a bijection.

**Deadlock tables.** A position (or a part of it) is called a *deadlock*, if it makes the level unsolvable. The solver tries to detect small deadlocks which consist of few boxes. We note that to detect deadlocks efficiently is similarly hard as solving the level.

First, a trivial deadlock consists of just one box. For every field, we pre-calculated whether it is possible to push a box from it to any goal (even if the rest of the boxes would be removed). If it is not possible, we call such a field a *dead field*. We denote dead fields "`x`". They are walkable for Sokoban, but no box can be pushed on them. Moreover, these field can be a part of the input file.

The solver contains several non-trivial deadlocks for two, three and four boxes. For example, two boxes aside a wall, for which at least one box is not at a goal, make a deadlock. Areas where these deadlocks can appear are pre-calculated. Using that, deadlocks are tested very fast during the solving.

**Tunnel macros.** A tunnel is a small part of the level that look like this:

```
 #   #          #   #
 #####          #####
@$       =>        @$
 #####          #####
 #   #          #   #
```

It makes no sense to push box only by one field. We either do not push the box at all, or we push it to the end of the tunnel. All the tunnels are pre-calculated and considered as a single field.

**PI-corral pruning.** This heuristic reduces the number of the positions that are expanded. Most of the time, there are locations, called *corrals*, that are inaccessable for Sokoban. A corral is called *PI-corral* if all the boxes on its border can be pushed only inside and all the moves are possible right now, not blocked by another box. For example, the left corral is unlike the other two a *PI-corral*:

```
######      ######      ######
#   $       #   $       #   $
#    $      #    $      #   $$
#   $$      #   $       #   $
####        ####        ####
   #           #           #
```

The key observation is the following. If the position inside a PI-corral is not solved yet, we have to push a box on its border, sooner or later. So, we can do it immediately, ignoring all the moves except pushing of the boxes on the border of this corral. This significantly reduces the number of moves generated from the position. It is not true that PI-corral pruning expands the positions that lead closer to the solution. It expands positions for which it is not clear whether they are dead. The size of the position is reduced and a small static deadlock may be detected. So, PI-corrals helps to prune the search tree and to kill dead positions.

# 3 Basic structure of the code

The code has the following structure. The file `solving_routine.h` contains the main routine. A tree contains nodes of type `move`. Every `move` contains a pointer to its parent and a structure called `position` which contains a 0-1 vector describing the positions of the boxes.

In the beginning, a level is loaded, using functions in `level.c`. It is slightly modified and several things are pre-calculated, using `level_info.c`. Also, deadlock tables in `deadlock_table.c` are initialized. Then the solving routing is initialized. Every position in the queue is analyzed. Functions in `crs.c` searches through the position and find all the possible moves from this position. For each move, we create a descendant and store it in the tree and to the queue. The memory is allocated by big chunks in `allocator.c`.

# 4 Non-implemented ideas

The last section contain several ideas which were not implemented yet. We plan to try them in future.

**Goal pushing.** This approach is key to be able solve more levels that look like the level shown above. This level has the following pattern. It contains a storeroom where all the goals are placed. There are other

rooms that contain the boxes. We can split solving of this level to a pair of independent problems: To find a way to push every box to the entrance of the storeroom and to push all the boxes from the entrance inside the storeroom. This could help to solve more difficult levels.

**Dynamic deadlocks.** This is another benefit of PI-corrals. If we detect that some bigger configuration of boxes is a deadlock, we can add this deadlock to the database and check it for future positions. For some levels, this can significantly improve the solving speed. On the other hand, if we add to many positions to the tables, it could lose the efficiency.

**Better tunnels.** We could generalize the tunnel detection even for other shapes of tunnels, for example a tunnel can be bended. This could lead to rather making macro moves than pushing boxes by one field.

**Better heuristics.** If we calculate the heuristic as the sum of the closest goal distances, we mostly obtain a mapping of boxes to the goal places which is not a bijection. We could construct a bipartite graph having the boxes as one part and the goals as the other part. An edge is connecting a box and a goal, if it is possible to push the box from the current position to this goal (even if the rest of the level is empty). We can even weight the edges by distances. Now, the more accurate heuristic is the weight of a smallest perfect matching. Moreover, if no perfect matching exists, the position is a deadlock. It is little slower to calculate this heuristic but the solver could explore more relevant positions.

**Level symmetries.** This is our original idea. Several levels of many authors are very symmetric—it is natural since symmetries are considered to be very beautiful. We could identify the positions that are symmetric. We believe that it would be possible to check all the symmetries very fast, using some bitwise operations. Moreover, we could extend this idea even for subpositions, for example rooms.

**Normalized positions.** In the current version, the positions can be distinguished even by the position of Sokoban. In many cases, it does not matter where Sokoban exactly stands. The positions having all the boxes placed the same and Sokoban stands in the same area of the level should be considered the same. We could calculate a normalized position of the Sokoban, for example in the top left corner of the area. This approach has a con, since to make a correct numbering we need to expand the positions before inserting them to the tree (and the queue). In the current version, we can do this by calculations with vectors.

**Parallelization and backward pushes.** It would be nice to split the work between several processes, each calculating some small subset of positions. As almost every parallelization, the main problem with this idea is that we need to share common data, in this case the hash tables. If we do not share them, we would calculate the same positions more times. If we share them too often, we lose too much time.

The following idea could be used to split the problem between two processes: The first process searches in the normal manner, the other one searches from the level in the solved positions (there may be more positions of Sokoban), using backward pushes. If these calculation meet and actually need to share data, the level is already solved.

Also, we could split the work in a non-symmetric way. One process would be solving the level and the other ones would try to prove that the current position is a deadlock.

**Better implementation.** Several parts of the solver can be implemented in a much faster way. For example, we could try to make the solver more cache friendly. Also, we could use the SSE instructions to compute with positions and deadlock faster. This could be very important if the deadlock tables would be significantly larger.

# References


[1] Joseph C. Culberson, Sokoban is PSPACE-complete, 1997.

[2] Andreas Junghanns, Pushing the Limits: New Developments in Single-Agent Search, Ph.D. Thesis, University of Alberta, Department of Computing Science, 1999.

[3] Sokoban Wiki, `http://www.sokobano.de/wiki/`.