Internal Report 2011–05 June 2011

Universiteit Leiden
Opleiding Informatica

Hashing Methods
in
Sokoban

Edwin Veger

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Hashing methods in Sokoban

Edwin Veger (eveger@liacs.nl)

June 16, 2011

Abstract

The game of Sokoban has been the subject of many scientists' fascination, because of the complexity of the nature of the puzzle despite its apparent simplicity. When trying to solve a Sokoban puzzle, remembering where one has already been is crucial. In this thesis, we present our findings with regards to hashing states with the use of minimum memory resources. We will go beyond the subject of hashing a state and attempt to efficiently map the state graphs of entire puzzles using several methods. We will also consider the impact of different hashing methods on these state graphs and their corresponding strongly connected components. Furthermore, we calculate a difficulty indication for a puzzle called the solution factor.

1 Introduction

Sokoban is a popular and complex game. First, we will familiarise the reader with the rules of Sokoban. Then we ponder the nature and prevention of deadlocks, which will aid us, along with the introduction of ED2.2 encoding, to efficiently and reversibly store a Sokoban game state. Our efforts in these subject will then enable us to generate the state graph of a Sokoban puzzle. Before we get into the game of Sokoban, it is important that we look a little bit closer to how we are going to represent a graph of Sokoban states.

2 Basic rules of Sokoban

The word Sokoban means *warehouse keeper* in Japanese, and that covers the game quite rightly. The player must find a way to push his/her boxes to their targets within the warehouse.

The player controls the warehouse keeper in a two-dimensional $m*n$ grid (the warehouse). There is exactly one man in puzzle, and outer sides consist of walls. The goal is to push all of the boxes onto target squares. This results in the seven possible configurations of a square. In Table 1, all possibilities are shown. Two additional constraints are that the number of boxes must be equal to the number of target squares, and that the number of `$`'s is equal to to number of `.`'s plus the number of `+`'s. Furthermore, the player cannot pull a box, and can only push a box if the square behind it is empty. In Figure 1, a basic example illustrates our rules.

		Square types		
		normal	target	wall
Occupants	empty		.	#
	man	@	+	*n/a*
	box	`$`	*	*n/a*

Table 1: Possible square configurations in Sokoban.

As described by Culberson [1] and Hearn [2], Sokoban is PSPACE-complete. This qualifies Sokoban to be among the hardest problems in

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

Figure 1: A simple Sokoban puzzle.

PSPACE (which is the set of decision problems solvable in polynomial space).

In this paper we shall refer to two sets of puzzles: the original Sokoban set [6] and the Microban set [5].

2.1 State graphs

This paper aims to illustrate the structures in which Sokoban states relate to one another within one puzzle. We shall call this structures *state graphs.* Although it sounds rather new, we have solid ground on which we can base our state graph definition. We will use the concept of *finite automata* (as described by [4]).

Definition The state graph of a Sokoban puzzle is defined as quintuple G = (Q, Σ, q_0, S, δ), where

- Q is finite set of ordered strings of occupation of non-wall squares, prepended with a binary representation of the man's room number[1];
- Σ is a finite set of box moves[2];
- $q_0 \in Q$ (the initial state);
- $S \subseteq Q$ (the set of solved states);
- δ is a function from $Q \times \Sigma$ to Q (the transition function).

In Section 6, we will elaborate further on this definition.

[1] For a expanded explanation and additional methods, see Section 4.

[2] We define a box move as the coordinates of the box and the direction of the move. The positions of the man before and after the move can be derived from this.

3 Deadlock

There are situations from which a solution can no longer be reached. We call these states *deadlocks*: for some reason, it is impossible to finish the game with all boxes on a target square. Since any state in deadlock is a waste of time at best when looking for a solution, it is well worth the effort of identifying a deadlock. We distinguish between the two deadlock categories *deadspot* and *positional deadlock*.

3.1 Deadspots

Deadspots are squares from which, once a box has entered it, the game can no longer be solved. So, to create a deadlock by means of deadspot, one needs only to move a single box. Because of this, deadspots are a very useful and cheap way of detecting deadlock. In the first subsection, we define the simplest type, *static deadspot*. In the second subsection, a more complicated type is introduced, called *indirect deadspots*.

Note that all deadspot types are independent of the position of the man: once a box is placed on such a square, it can no longer reach a target square.

Definition Let P be a Sokoban puzzle. Let S be the set of static deadspots of P, and let I be the set of indirect deadspots of P. Then, $S \cap I = \emptyset$ and $S \cup I = D$, where D is the set of all deadspots of P.

3.1.1 Static deadspots

A static deadspot, simply put, is a corner. It effectively immobilizes a box, and it is obvious that if a box can no longer be moved and it is not on a target square, the game is frozen. The enumeration below defines a static deadspot's features:

1. the square is not a target square;
2. the square is adjacent to at least two walls;
3. at least two of the adjacent walls are not on the opposite side of the square.

```
#######          #######          #######
#     #          #x   x#          # xxx #
# .   #          # .   #          #x.  x#
#   $ #          #     #          #x   x#
# @   #          #x   x#          # xxx #
#######          #######          #######
```

Figure 2: An example puzzle and its static and indirect deadspots.

When a box is placed on a static deadspot, it causes *static deadlock*. For an example, see the middle panel in Figure 2, where four static deadspots are denoted with an `x`. Since only one box is involved, static deadlock is very easy to detect.

3.1.2 Indirect deadspots

The extent to which corners cause deadlock are significant but by no means exclusive. Therefore, we discuss indirect deadspots. The reason we call them indirect is because, to a layman, the deadlock is not yet visible. Perhaps all boxes can still be moved to some extent. Along some walls boxes can still be moved without causing static deadlock, and the player will only realise after a certain number of box moves that the game has become unsolvable. Because of their complicated nature, we have to define indirect deadspots recursively, rather than statically. A square is an indirect deadspot when:

1. the square is not a target square;
2. the square is not a static deadspot;
3. a box placed on the square cannot reach a square which is not a deadspot (either static or indirect).

In defining deadspots, we do not take any other boxes into account except the one that is hypothetically placed onto the relevant square. For a visualisation, see the puzzle on the right hand side in Figure 2, where `x` is used to denote the 10 indirect deadspots.

```
mark all non-walls as deadspots
mark all non-walls as not yet processed
mark all target squares as non-deadspots

for (every square which is not yet processed and non-deadspot) do
    sq1 = this square
    for (each direction) do
        neighbour = square in direction
        if (neighbour is not yet processed and
            box from neighbour could be pushed to sq1)
            mark neighbour as non-deadspot
        fi
    od
    mark sq1 as processed
od
```

Figure 3: The algorithm for finding all deadspots in pseudocode.

3.2 Algorithm

To find all deadspots within a puzzle, we cannot instantly determine a square to be one based on static definitions. We have to go look for them. To this aim, we have developed the following algorithm (Figure 3).

This algorithm uses a reversed solving technique (as demonstrated by [8, 7]. It works back from the target squares to find squares which can lead to those targets. Since any square which can lead to a target is valid, any square which can lead to any of these squares it automatically valid as well. The algorithm takes advantage of this property. Note that static and indirect deadspots are detected equally, since both satisfy the property of not having a valid box move path to a target square. Figure 4 illustrates how this works in practice.

If the box is pushed to one of the four walls, it can no longer reach the target square. Therefore, all the squares immediately adjacent to the walls are marked as deadspot, since they produce a deadlock. For the sake of simplicity, we will call all squares within the walls of the puzzle which are neither deadspot nor wall *valid squares*.

```
####      ####      ####      ####      ####
#  ##     #xx##     #xx##     #xx##     #xx##
# @ ##    #xxx##    #?xx##    # ?x##    #  x##
#. $ #    #?xxx#    # ?xx#    #  ?x#    #   x#
#    #    #xxxx#    #xxxx#    #xxxx#    #xxxx#
######    ######    ######    ######    ######
```

Figure 4: An example puzzle and our deadspot search algorithm animated. First the regular representation (with `.`, `@` and `$` as the target square, the man and the box). After that, four iterations of our deadspot search algoritm as described in Figure 3. After the regular representation (the first panel), all non-wall squares are marked as deadspots (panel two). A `?` denotes that that square will be examined in the next iteration. After four iterations, no question marks remain so the algorithm finishes.

3.3 Positional deadlock

Whereas deadspots rely on a single box to produce deadlock, positional deadlock concerns two or more boxes. A positional deadlock is the presence of one or more boxes on a non-target square, which cannot be moved to a target square because of other boxes which cannot be moved. Figure 5 portrays examples of positional deadlock. The boxes shown in this example cannot be moved to any target square. The squares on which they stand are in themselves harmless, and it is because of the other boxes that they trigger a positional deadlock.

Note that these deadlocks do not follow directly from the locations where the boxes are, but rather from the combination of box locations and their relation to each other. Because of this, positional deadlocks are much more complicated to detect than deadlocks caused by deadspots. They require a complete check of the warehouse after every box move to see if deadlock patterns have appeared. Alternative methods exist, but further exploring them is not the aim of this paper and we will not go into further detail here.

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

Figure 5: Examples of positional deadlock.

4 Representing a state

When it comes to states, there are several ways in which we can describe them. The most obvious one is rather inefficient, and we shall discuss the techniques we used to reduce memory problems while maintaining all necessecary information.

4.1 Basic description

One can view the puzzle as a two-dimensional grid in which each square can have one of the following values: `@, +, $, *,  , .` or `#`. In the example of puzzle 1 of the original testset [6] (Figure 6), the grid size is 11 by 20. If we assign a *char*, which is 1 byte, for each square, this sums up to 220 bytes.

4.2 Reduction of redundancy

If we reason that the walls and target square will remain unchanged through the length of the game, we can store them externally and, thus, only once. From this, it follows that only the positions of the boxes and of the man are unique for each state. Now, when we store these, if would be efficient if we only take valid squares into account (so no deadspots). If we look at Puzzle 1, the valid squares would look like those in Figure 6. There are 41 squares left. Notice that by ignoring the deadspots we save 15 squares.

Of these squares, we really only need to know whether there is box on it or not. Now that is quite a decrease, from 220 bytes to 41 bits. In addition, the man's position can be recorded either as the (rather expensive) x and y coordinates or as a single *room number*. Since the man can move freely in any room without making irreversible moves, the exact location of the man does not matter. As long as the room, or a group of squares is known, this is enough to uniquely identify the state. Note that the man suffers no penalty for standing on a deadspot: therefore, deadspots are taken into account when calculating room numbers. In any given puzzle, room numbers are assigned deterministically, and in this process boxes as treated as walls. To see this in practice, see Figure 6. In addition to the flags of the valid squares, one only needs to add the room index to fully describe the state. The starting state of puzzle 1 would be thus represented:

```
10000101010000000000100100000000000000000
```

followed by room number 3.

4.3 Encoding Digits algorithm

In the previous section, we desribed a rudimentary way of representing a state. One can easily see there is a tremendous amount of redundancy packed in this string. For example, after encountering six one's, meaning six boxes, one does not have to look any further since all boxes are accounted for.

For this situation, we have developed the *Encoding Digits* algorithm. Remember that we are still trying to encode as much information into as few bits as possible *without losing any data*. This means that all states that differ from one another will be uniquely represented in this way.

We shall be using the ED2.2 algorithm. The distances between subsequent boxes will be encoded in binary. To be able to see where one distance ends and another begins, the base number of `0`'s and `1`'s is set to 2 (hence the .2). To accomodate distances greater than 3 ($= 2^2 - 1$), the value of two ones is used to denote that the value of the additional two digits following this block should be added to this value. In this fashion, we would describe a distance of 4 as `11 01` (which means 3 + 1), a distance of 3 as `11 00` (3

```
0     #####          |                 |     _____
1     #   #          |    111          |     _XXX_
2     #$  #          |    $11          |     _..._
3   ###  $##         |    11$          |   ___...__
4   #  $ $ #         |  22$1$3         |   _X....X_
5 ### # ## #   ######|  2 4  3         | ___._.__._   ______
6 #   # ## #####  ..#|222 4  3     3333| _X.._.__._____X..._
7 # $  $          ..#|2$22$333333333333| _X................_
8 ##### ### #@##  ..#|    3   3 @  3333| _____.___._X__X..._
9     #     #########|    33333        |     _XXXXX_________
0     #######        |                 |     _______
0         _ _ _ _ _
1         _ . . . _
2         _ 1 2 3 _
3     _ _ _ 4 5 6 _ _
4     _ . 7 8 910 . _
5 _ _ _11 _12 _ _13 _       _ _ _ _ _ _
6 _ .1415 _16 _ _17 _ _ _ _ _ .181920 _
7 _ .21222324252627282930313233343536 _
8 _ _ _ _ _37 _ _ _38 _ . _ _ .394041 _
9         _ . . . . . _ _ _ _ _ _ _ _ _
0         _ _ _ _ _ _ _
```

Figure 6: Puzzle 1 of the original testset. Clockwise, starting top left: the puzzle, its room numbers, deadspots and valid squares. Note that we have omitted the walls outside enclosing walls for readability purposes.

```
0     #####           |          _ _ _ _ _
1     #   #           |          _ . . . _
2     #$ $#           |          _ 1 2 3 _
3   ###   ##          |      _ _ _ 4 5 6 _ _
4   #   $  #          |      _ . 7 8 910 . _
5 ### # ## #   ###### |  _ _ _11 _12 _ _13 _       _ _ _ _ _ _
6 #   # ## #####  ..# |  _ .1415 _16 _ _17 _ _ _ _ _ .181920 _
7 # $       @$    ..# |  _ .21222324252627282930313233343536 _
8 ##### ### # ##  .*# |  _ _ _ _ _37 _ _ _38 _ . _ _ .394041 _
9     #     ######### |          _ . . . . . _ _ _ _ _ _ _ _ _
0     #######         |          _ _ _ _ _ _ _
```

Figure 7: Puzzle 1 of the original testset, partially solved: the puzzle (left) and valid squares (right).

+ 0) and a distance of 10 as `11 11 11 01` (which means 3 + 3 + 3 + 1).

When we look at Puzzle 1 (Figure 6), the valid square numbers on which a box stands would be 1, 6, 8, 10, 21 and 24. From this it follows that the distances between the boxes are 0, 4, 1, 1, 10 and 2. Keep in mind that we start counting from square 0, and that we define distance as the amount of valid squares which lie in between. Our previous distance list is encoded into `00 11 01 01 01 11 11 11 01 10`, (which means 0, 3 + 1, 1, 1, 3 + 3 + 3 + 1, 2). For further illustration, Table 2 is included.

Valid square numbers	1	6	8	10	21	24
Distance from previous	0	4	1	1	10	2
Encoded value (2 bits)	`00`	`11 01`	`01`	`01`	`11 11 11 01`	`10`

Table 2: Encoding of puzzle 1

However, this method is still flawed. Take a look at Figure 7, which shows again Puzzle 1, but now partially solved. The encoding of this state using two bits as the standard length is shown in Table 3.

Note the many `11`'s in the middle; these bits are costly. If we could occasionally increase the bit length of each distance, this could decrease the overall length of the total bitstring, i.e., the value 6 could be encoded as `110`

Box number		1	2	3	4	5	6
Valid square number		1	3	9	21	30	41
Distance from previous		0	1	5	11	8	10
Encoded value	2 bits	`00`	`01`	`11 10`	`11 11 11 10`	`11 11 10`	`11 11 11 01`
	3 bits	`000`	`001`	`101`	`111 100`	`111 001`	`111 011`
	4 bits	`0000`	`0001`	`0101`	`1011`	`1000`	`1010`

Table 3: Encoding of Puzzle 1 as shown in Figure 7.

instead of `11 11 00`. To achieve this, each string will be preceded by two *multiplier bits*, which contain a binary encoded integer. This integer stands for the number of *additional* bits used to encode each distance. This new way of encoding is shown in Table 3 as well, with multiplier values of one and two. Note how the total length of the bitstring decreases. By prepending the two multiplier bits, we add flexibility to the encoding algorithm.

The room number of the man's position is encoded within 8 bits and prefixed to the previously discussed string. We get:

```
00000001 10 0000 0001 0101 1011 1000 1010
```

We have two reasons for choosing the ED2.2 algorithm[3]. Because of the short base length, puzzles with a relatively large number of boxes (which imply small distances between) can be encoded efficiently. Secondly, puzzles with relatively few boxes can be encoded efficiently as well, as the two multiplier bits allow for maximum distance of 30[4]. Within the context of Sokoban, this is assumed to be sufficient.

5 Generating all possible states

We use the algorithm described in Figure 8 to find all states in a puzzle. This method is based on breadth-first search. The algorithm processes a queue of Sokoban states, and for each of these states, their children (states that are

[3] The first 2 stands for the number of multiplier bits and the second 2 stands for the base length of a binary distance.

[4] 30 is encoded as `11100`, 31 is encoded as `11111 00000`, and so on.

one box move away) are generated and pushed to the queue.

```
input: Sokoban state $S$
output: set of hashes

queue $Q \leftarrow \emptyset$
hash_set $H \leftarrow \emptyset$
$Q$.push($S$)

while ($Q$ is not empty) do
    state $current \leftarrow Q$.pop();
    if ($current$ is in $H$) then
        break
    fi

    $H$.push($current$)
    for (each box in $current$)
        for (each direction)
            if (the box can be moved in this direction) then
                state $N \leftarrow current$ // create a new state
                $N$.moveBox($x$,$y$,direction)
                if ($N$ is not in deadlock) then
                    $Q$.push($N$)
                fi
            fi
        rof
    rof
od
```

Figure 8: Pseudocode for our exhaustive breadth-first search algorithm.

6 Strongly connected components

In the previous section we described a manner in which we can generate all possible states of a Sokoban puzzle: the state graphs as defined in Section 2.1. These graphs can be cycles: groups of states in which any state can lead to another. We call these groups *strongly connected components*.

```
#######  #######  #######  #######
#     #  #     #  #     #  #     #
# $  .#  #  $ .#  #   $.#  #    *#
#######  #######  #######  #######
```

(a) (b) (c) (d)

Figure 9: Examples of equivalence. For illustration purposes, the man has been left out.

6.1 Equivalency

If there is a cycle in the state graph, it implies that there is at least one state which can reach itself through one or more other states. Imagine a box that can be pushed freely between three empty spaces (Figure 9 (a), (b) and (c)). Note that (a) can lead to (b) and vice versa. Also note that (b) can lead to (c) and vice versa. This implies that any state that can be reached from (a) can also be reached from (b) and (c), and vice versa. A move to state (d) is irreversible. In this way, in a state graph, we can treat (a), (b) and (c) as the same state. They are what we shall call *equivalent*.

Definition Let G be a state graph (Q, Σ, q_0, S, δ), and $p, q \in Q$. If and only if for $n, m \geq 0$, there are s_1, s_2, ..., $s_n \in \delta$ and t_1, t_2, ..., $t_m \in \delta$ such that
$p \xrightarrow{s_1} p_1 \xrightarrow{s_2} p_2 \rightarrow \ldots \xrightarrow{s_n} p_n = q$ (with $p_i \in Q$ for $i = 1, \ldots, n-1$) and
$q \xrightarrow{t_1} q_1 \xrightarrow{t_2} q_2 \rightarrow \ldots \xrightarrow{t_m} q_m = p$ (with $q_j \in Q$ for $j = 1, \ldots, m-1$),
then $p \sim q$ (p and q are *equivalent* to each other). Note that this is indeed an equivalence relation:

1. for all $p \in Q$, $p \sim p$ (*reflexivity*);
2. for all $p, q, r \in Q$, if $p \sim q$ and $q \sim r$, it also holds that $p \sim r$ (*transitivity*);
3. for all $p, q \in Q$, if $p \sim q$, it also holds that $q \sim p$ (*symmetry*).

6.2 Equivalence classes and the SCC graph

In this paper, among other things, we are looking for the number of *equivalence classes* of a given puzzle. We define equivalence classes as subgroups in which each state can lead to any of the others through some number of transitions; in other words, strongly connected components.

Definition Let $G = (Q, \Sigma, q_0, S, \delta)$ be a state graph of a Sokoban puzzle. The equivalence class $E \subseteq Q$ of an element $s \in S$ consists of all $s' \in S$ such that $s \sim s'$.

In this way we define the generalized state graph of a puzzle, where the These equivalence classes are the nodes of the so-called *SCC graph.* There is an edge from class E_1 to class E_2 if and only if there are $s_1 \in E_1$ and $s_2 \in E_2$ with an edge from s_1 to s_2.

6.3 Tarjan's Algorithm

An efficient algorithm to detect strongly connected components in a graph is Tarjan's algorithm [9]. See Figure 10 for a detailed description of the algorithm. It basically returns a list of lists of equivalent nodes/states.

The algorithm makes a depth first traversal of the graph, labeling each node with its index. The lowlink label of a node n is intended for the smallest lowlink of any of its children. If a child node has already been assigned an index, this is assumed to be the highest up in the graph one can get and this value is stored, through the recursion, in all of the node's predecessors, up until the node itself (since it is the parent of the others). This first node is the root of the strongly connected component.

7 The application of Tarjan's algorithm

We will combine the techniques discussed and displayed in the previous chapters: the generation of an exhaustive breadth-first search, which produces nodes and edges (the state graph). When we apply Tarjan's algorithm to this graph, cycles are detected and replaced by a single node.

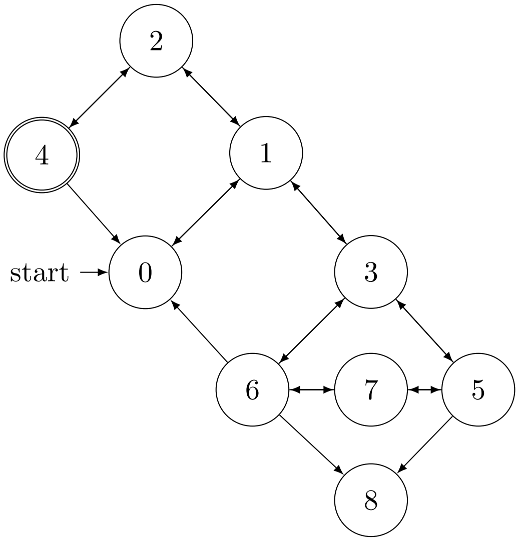
```
input: graph $G = (V, E)$
output: set of strongly connected components (SCC; set of vertices)

$index \leftarrow 0$
$S \leftarrow \emptyset$ // an empty stack of nodes
for each $v$ in $V$ do
    if ($v.index$ is undefined) then
        tarjan($v$)
    fi
od

function tarjan($v$)
    $v.index \leftarrow index$ // set the depth index for $v$
    $v.lowlink \leftarrow index$
    $index \leftarrow index + 1$
    $S$.push($v$)

    for each $(v, w) \in E$ do // consider successors of $v$
        if ($w.index$ is undefined) then
            tarjan($w$)
            $v$.lowlink $\leftarrow$ min($v$.lowlink, $w$.lowlink)
        else if ($w \in S$) then
            $v$.lowlink $\leftarrow$ min($v$.lowlink, $w$.index)
        fi
    od

    $scc \leftarrow \emptyset$ // an empty list of integers
    // if $v$ is a root node, pop the stack and generate a SCC
    if ($v$.lowlink = $v$.index) then
        do
            $w \leftarrow S$.pop()
            $scc$.push($w$) // add $w$ to current SCC
        while ($w \neq v$)
        od
        output $scc$
    fi
end function
```

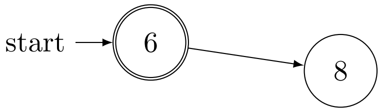
Figure 10: Pseudocode for Tarjan's algorithm.

7.1 Microban puzzle #2

In Figure 11, we can see Microban puzzle #2 and its state graphs. The extent to which the original state graph with potential cycles is reduced to an SCC graph is characteristic. Since any of the first eight states (states 0 to 7) can be reached from any of the other seven, they are all equivalent to each other and thus form an equivalence class. Since state 4 is a solution, the entire class is marked as a solution in the SCC graph. State 8, the last state, has no outward arrows and forms its own equivalence class.

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





Node	Contains
8	8
6	6 7 5 3 4 2 1 0

Figure 11: The starting state of Microban puzzle #2 (top left), its state graph (top right) and SCC graph (bottom left). The table (bottom right) contain the nodes that correspond to the original state graph.

It is important to note that, though we have elimated deadlocks via static and indirect deadspots, positional deadlocks remain. Since state 8 has no outward arrows which could lead to a solution, it is in positional deadlock. A little mental box-pushing will show you that in state 8 all three boxes are adjacent to the wall in the middle of the puzzle. They obediently avoid the deadspots, yet cannot, by themselves, prevent the positional deadlock which immobilizes them all.

7.2 Microban puzzle #1

The previous example animated the process in a comprehensive way. Since Microban puzzle #1 is a little more complicated, the state and SCC graphs are a bit larger, as can be seen in Figure 12. These graphs are obviously dwarfed by graphs of real Sokoban puzzles, but depicting those countless tangling lines and entangled nodes would serve no purpose.

7.3 Interesting cases

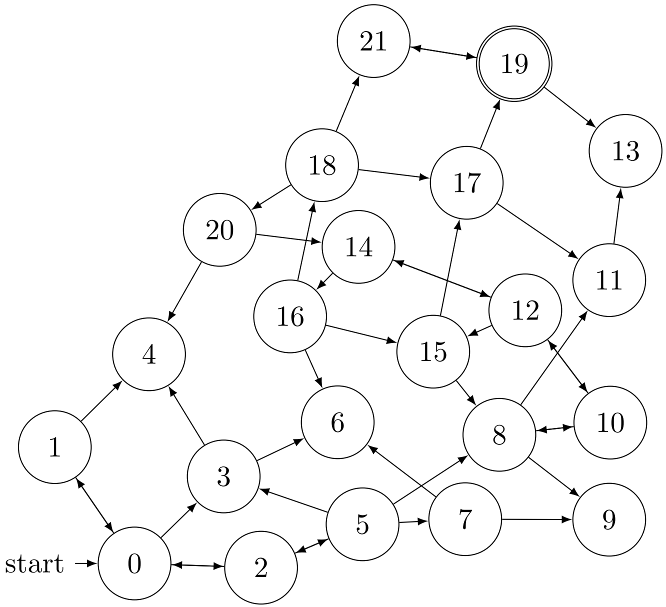
All the puzzles we have seen thus far produce state graphs with a single accepting state. But multiple different solutions are not impossible. All solutions share one feature, namely that all boxes should be placed on a target square. But the final position of the man is not fixed. Figure 13 displays a custom made puzzle, whose SCC graph exemplifies the above statement.

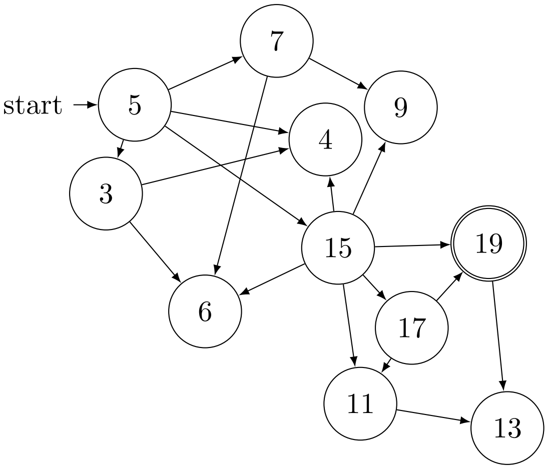
8 Difficulty indicators

As we have seen in the previous chapter, puzzles of serious complexity can be reduced through Tarjan's algorithm to quite comprehensible and simple graphs. Any puzzle's state graph can be explosive in size, but the SCC graph more closely represents the true nature of the puzzle. Can we use the SCC graph to indicate a difficulty level?

The number of unique paths through the SCC graph from start to solution is relevant in this context. In addition, the number of non-accepting states with no outward arrows (or *sinks*) is useful here as well. The size of the graph is also a viable indicator, since any increase in size will most likely increase the proportion of wrong paths which do not lead to solutions. This goes for the reverse as well: any reduction in SCC graph nodes will raise the percentage of solution paths.

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

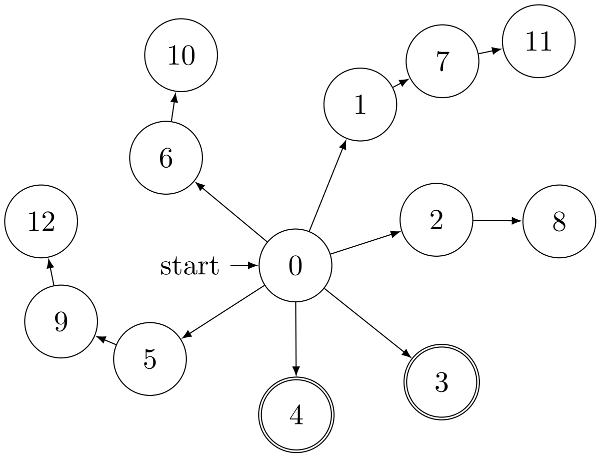




Node	Contains
4	4
6	6
3	3
9	9
7	7
13	13
19	19 21
11	11
17	17
15	15 20 18 16 14 12 10 8
5	5 2 1 0

Figure 12: The starting state of Microban puzzle #1 (top left), its state graph (top right), its SCC graph (bottom left), and a list of nodes corresponding to SCC's (bottom right).

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



Node	Contains
0	all other 248 nodes
1	1
2	2
3	3
4	4
5	5
6	6
7	7
8	8
9	9
10	10
11	11
12	12

Figure 13: The starting state of a custom made puzzle (top left), and the two possible solutions (top middle and top right). The SCC graph (bottom left) and the corresponding node table (bottom right).

8.1 The solution factor

We have combined the aforementioned properties into a recursive algorithm. This algorithm labels each node from the SCC graph with a difficulty indication of the puzzle at that game state. This marking expresses the probability of reaching a solution by randomly choosing next nodes until a solution is encountered or the game becomes unsolvable[5]. We shall call this measure of solvability *the solution factor*. The solution factor of a puzzle is defined as

[5] We assume that the probability of reaching a next node is equally distributed, although in reality some transitions/arrows might have greater probability than others.

the solution factor of its starting node. The solution factor of a node n is calculated by summing all the solution factors n's children, divided by the number of children;

Definition The solution factor of a node n is defined as $SF(n)$, where

- $SF(\mathrm{n}) = 1$ is n is a solution;
- $SF(\mathrm{n}) = 0$ is n is a sink and not a solution;
- $SF(n) = \sum_{i=1}^{q} \left(\frac{SF(m_i)}{q} \right)$ with $m_1, \ldots, m_q$ being the q children of n with $q \geq 1$, if n is not a solution.

When we apply this algorithm to the SCC graph of Microban puzzle #2, we get results as presented in Figure 14. The sinks have been represented by a "forbidden" sign to emphasize their null solution factor, and the solution state has a 1 by default.

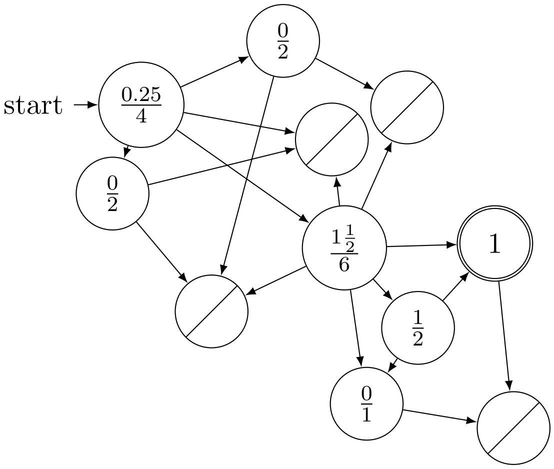


Figure 14: The SCC graph of Microban puzzle #2 with added solution factors. The solution factor of the starting state is $\frac{0.25}{4} = 0.0625$. Nodes marked with a / are a sink (since they have no outward arrows).

We would like to note that this recursive algorithm only works on acyclic graphs. Fortunately, SCC graphs are acyclic by default so we do not have to concern ourselves with this limitation. One might imagine to have a technology similar to Google's PageRank algorithm [3] to eliminate the dangers of recursion and produce a balanced solution factor for all nodes. PageRank uses an iterative approximation method instead of a recursive one.

Furthermore, if a graph's solution factor is 1, this does not mean that there are no sinks in the graph: sinks may very well occur within such a graph. It is important to realize that any sink positioned *beyond* a solution node is not represented in the solution node's solution factor, since that is always equal to 1. Imagine the starting node of Figure 14 to be a solution node: the graph's solution factor would be 1, even though there are several sinks in the graph.

It is also clear that staying in one SCC node might be very hard. If an SCC node contains a lot of outgoing arrows and but a few internal arrows, it is more likely that you will perform an irreversible move (i.e., to another SCC or node). The internal structure of an SCC is not taken into account in the solution factor calculation.

9 Method

It is time to put all pieces of the puzzle together, and see what results that yields. The variable we are going to test will be the hashing of a state. We will use several methods, using ED2.2 as a one-to-one method, as well as other real hashing methods in which collisions will appear. If a collision does appear, i.e., we think that we have already visited this state before, we will assume this to be true. In this way, when hashing, we can encounter false positives which will invalidate our state graph. Note that false negatives (i.e., we deem a state to be unvisited although we have seen it before) remain impossible. We are interested in the extent to which the hashing method has an impact on our state graph.

9.1 Data

We will compare our true graph, obtained by perfect hashing (referred to as one-to-one), to other graphs, which are generated using many-to-one hashing methods. We will use three methods:

1. Using *No man hashing*, we omit the man's room number from the hash. In puzzles with a high box-to-square ratio, this should have quite an impact since box configurations with different positions of the man are treated as though they were the same.

2. With the *2-digit hashing* method, we combine every two bits into one bit using a binary AND operation, effectively halving the original string.

3. *3-odd-parity hashing* takes three bits. The resulting bit is set to `1` when the amount of `1`'s is odd.

The results of these hashing methods compared with their "perfect" counterpart can found in Table 4. Also given is the solution factor for the perfect hash method.

Some puzzles have a solution factor of 1. This may seem impossible, since almost every puzzle is subject to positional deadlock. However, if the solution is in the same SCC as the starting state (each can result in the other), the solution factor is 1 by definition (see Section 8.1).

It is obvious that the 2-digit hashing (which loses over half of the data) is overshooting the target. The 3-odd-parity hashing destroys over 75% of the data, yet it results in the same figures as the 2-digit hashing. There is a significant difference between the results of the first and the second hashing method and the third and fourth. From these results, we might interpret methods three and four to lack accuracy and therefore usefulness.

Method "No man" achieves reasonable results, even though it omits 8 bits. An important feature is that this method distinguishes between the importance of bits: it always excludes the man room data, whereas the other two methods randomly exclude bits of information, boxes and man room alike.

			One-to-one		**No man**		**2-digit**		**3-odd-parity**		
Level	**Size**	**Boxes**	**Nodes**								**Solution factor**
			state graph	*SCC graph*	*state graph*	*SCC graph*	*state graph*	*SCC graph*	*state graph*	*SCC graph*	
1	9	2	22	11	21	11	3	1	3	1	0.062500
2	5	3	9	2	9	2	1	1	1	1	1
3	11	2	83	61	44	32	5	4	5	4	0.002315
4	8	3	56	4	56	4	4	2	4	2	1
5	12	4	496	10	495	9	1	1	1	1	1
6	14	3	465	143	212	66	4	1	4	1	0.000457
7	12	6	922	18	922	18	28	2	28	2	1
8	15	2	133	102	64	44	2	1	2	1	0.000290
9	8	2	19	17	14	11	1	1	1	1	0.125000
10	14	3	161	73	66	41	2	2	2	2	0.001157
11	13	2	92	21	62	18	5	4	5	4	0.035714
12	9	2	35	32	24	14	1	1	1	1	0.038580
13	14	3	409	125	275	96	1	1	1	1	0.003105
14	6	2	15	3	12	4	3	1	3	1	1
15	8	2	36	12	21	15	1	1	1	1	0.125000
16	21	3	1515	780	1267	536	1	1	1	1	$0.287833 \cdot 10^{-7}$
17	9	3	72	19	64	19	3	1	3	1	0.340278
18	13	2	91	43	71	27	2	1	2	1	0.002344
19	11	2	75	47	16	6	1	1	1	1	0.002604
20	12	2	74	35	44	14	7	1	7	1	0.029412
21	5	2	11	3	8	4	3	2	3	2	1
22	12	2	72	13	66	15	6	2	6	2	0.010204
23	11	2	24	8	24	8	2	1	2	1	1
24	10	2	63	42	44	26	1	1	1	1	0.007937
25	9	3	97	17	84	25	1	1	1	1	1

Table 4: This table describes general information about the first 25 puzzles of the Microban testset. The number of nodes in the state graph and the SCC graph are shown, next to a puzzle's solution factor. Calculating these data took under 6.5 seconds on a 2.2 GHz machine.

10 Conclusions and future work

In this paper, we have applied the theory of finite automata to Sokoban puzzles to produce state graphs. We defined the aspects of deadlock and its causes and applied a primitive form of deadlock prevention. To improve encoding efficiency, we introduced the ED2.2 algorithm. Using Tarjan's algorithm, we can create SCC graphs which more accurately represent a puzzle's structure. We have introduced the solution factor as a difficulty indicator.

The mapping of Sokoban puzzles into comprehensible graphs bring us more insight into the different kinds of puzzles. Although the ED2.2 algorithm is a leap forward in efficient state storage, more improvement can be made. For example, the starting square and the direction from which the distances are counted can be made dependent on the optimal orientation. Instead of from left to right and top to bottom, the algorithm could count bottom to top, right to left if that would yield smaller distances or take less squares before having "touched" all the boxes.

The selective hashing method (the no man hash) performs quite well. The greedy hashing methods exclude too much data, and destroy a puzzle's characteristics. Perhaps other selective methods (such as removing the last or first box from the hash) can yield similar performance and memory savings.

The solution factor as a difficulty indicator can further be explored. It might be exploited in relation to the state graph, in addition to the SCC graph as demonstrated here. The threat of cycles in state graphs might be handled using an approach similar to PageRank [3]. Whether or not the solution factor converges with human perception of difficulty might be investigated in an experiment. One could also compare the solution factor for different hash methods and see if it correlates with the level size, box-to-square ratio, etc. There is a plethora of possibilities for data analysis.

Acknowledgements

This Bachelor Project was done under supervision of dr. Walter Kosters from LIACS, Leiden University. I would like to thank him for his support and patience over the extensive period that this paper has required to come to be. I would also like to thank Frank Takes for sharing his experience and programming knowledge.

References

[1] J. Culberson. Sokoban is PSPACE-complete. In *Fun With Algorithms*, volume 4, pages 65–76, 1999.

[2] R.A. Hearn and E.D. Demaine. *Games, Puzzles, and Computation.* AK Peters Ltd, 2009.

[3] A.N. Langville, C.D. Meyer, and P. Fernández. Google's PageRank and beyond: the science of search engine rankings. *The Mathematical Intelligencer*, 30(1):68–69, 2008.

[4] J. Martin. *Introduction to Languages and the Theory of Computation.* McGraw Hill, third edition, 2003.

[5] The original Microban testset. `http://www.sourcecode.se/sokoban/levtext.php?file=microban.slc`, 1982.

[6] The original Sokoban testset. `http://www.sourcecode.se/sokoban/levtext.php?file=Original.slc`, 1982.

[7] F. Takes. Sokoban: Reversed solving. *Bachelor Thesis, Leiden University*, 2007.

[8] F. Takes. Sokoban: Reversed solving. *2nd NSVKI Student Conference*, pages 31–36, 2008.

[9] R. Tarjan. Depth-first search and linear graph algorithms. In *Twelfth Annual Symposium on Switching and Automata Theory*, pages 114–121. IEEE, 1971.