Solving Sokoban screens using ROBDDs

S. Kikitamara (s4561414) P.T. Jager (s4644425)

April 21, 2016

1 Introduction

Reachability analysis is the process of computing the set of reachable states of a system. This can be used to build a model checker for a system, which can determine whether or not a system satisfies some given given properties. This is particularly useful for systems which have a lot of possible states.

In this assignment we will use reachability analysis, or more precise, the symbolic reachability algorithm presented in the lectures, to determine whether or not it is possible to solve a certain Sokoban screen.

The symbolic reachability algorithm requires that states can be expressed as logical formulas. To do this in an efficient way we will use ROBDDS, which are an efficient representation for logical formulas with a large number of variables, both for storage and for operations.

In our implementation we use the ROBDD library Sylvan. Our implementation is written in C++.

2 Implementation

2.1 Encoding

In our implementation Sokoban screens are encoded using variables denoting the x and y position of each block and the man. Initial, error and goal properties are then defined in terms of these variables, as is the transition relation. Below we will show this more clearly and more formally.

Ultimately we will combine these parts to check that there is a path from the initial configuration to a configuration which satisfies the goal property without ever satisfying the error property. More formally, we will check the following formula:

$$\mathsf{EG}(\neg\mathsf{error} \cup \mathsf{goal}) \tag{1}$$

To check if Equation 1 is solvable we calculate the least fixed point of it using the method shown in the lectures, that is we calculate the result of Equation 2. We then check of this result satisfies the initial property. If it does then the Sokoban screen is solvable.

$$Lfp(Z \mapsto goal \vee (error \wedge \mathbf{EX}Z)) \tag{2}$$

Screens We interpret a screen as a grid of x columns and y rows (position (0,0) is considered the position in the top left). Each block and the man has a position in the screen identified by which row and column it is in. Goal positions and walls are not explicitly encoded in the screen but are instead encoded in goal and error properties.

Positions For the man and for each block variables are defined which indicate in which row and in which column the block/man is. For a screen with x columns, y rows and n blocks we define $(n + 1) * (x + y)$ variables.

For each block we define

$bx_{i,x}$ which indicates if block i is in the column numbered x.

$by_{i,y}$ which indicates if block i in in the row numbered y.

For the man we define similar variables, but named just mx_x for the columns and my_y for the rows as a screen only has one man.

The actual implementation defines just a sequence of BDDvars, one for each of these variables.

Transition relation Before showing the different properties of the system, we will first show the transition relation which describes how the man moves around the board and how this affects blocks around him.

The main idea behind the transition relation is rather simple. Given that the man is in a certain position (x, y) he can move either up, down, left or right, and if there is a block in that position that block moves in the same direction. If the man is currently at an edge then he can not move in the direction which would make him fall of the board, and if the man is separated from the edge of the board by only a block, then he also can not move in that direction as that would push the block outside of the board.

Any transitions which result in either the man or a block overlapping with a wall, or which result in two blocks overlapping are considered erroneous and their resulting states will satisfy the error property.

Equation 3 shows the formal transition relation. blocks is considered the set of block(numbers) in this screen, *rows* the set of row numbers and *cols* the set of column numbers. max_r will be the maximum column number and max_c the maximum column number. In this relation the primed version of each variable denotes that variable in the next state, i.e. the state after the transition has happened. We will only fully show the formula for moving right, the formulas for moving left, up and down are defined similar.

$$
\begin{aligned}
&\bigwedge_{x\in cols, y\in rows} (mx_x \wedge my_y) \implies \Bigg(\\
&\qquad \Big(x \neq max_c \wedge \big(x \neq (max_c - 1)(\wedge_{b\in blocks}\neg(bx_{i,x+1}) \wedge by_{b,y})\big) \\
&\qquad\qquad \implies \big(\neg mx'_x \wedge mx'_{x+1} \wedge my'_y \\
&\qquad (\wedge_{b\in blocks} bx_{b,x+1} \wedge by_{b,y} \implies bx'_{b,x+1} \wedge bx'_{b,x+2} \wedge by'_{b,y}))\Big) \\
&\qquad\quad \vee \Big(\text{similar for left}\Big) \vee \Big(\text{similar for up}\Big) \vee \Big(\text{similar for down}\Big)\Bigg)
\end{aligned} \tag{3}
$$

For standard screens this encoding is actually more complex than necessary. Since all screens are bordered by a layer of walls it is not necessary to check if a block would be pushed of screen and if the man would walk of screen, since states in which a transition could result in this would already satisfy the error property, since a block or the man would overlap with the outside walls. However defining the transition like this allows for an optimization in which all layers of outside wall have been removed from a board, reducing the state space.

Error property A state is considered to be an error state if either the man or one of the blocks is overlapping with a wall, or when two blocks are overlapping. Equations 4 shows this relation formally. In this *walls* is the set of walls on the screen, and w^x denotes the column wall w is in and w^y the row it is in.

$$
\begin{aligned}
&\bigvee_{w\in walls, b\in blocks} (bx_{b,w^x} \wedge by_{b,w^y}) \\
&\qquad \vee \left(\bigvee_{x\in cols, y\in rows} \bigvee_{b\in blocks} \bigvee_{b'\in blocks\setminus\{b\}} (bx_{b,x} \wedge bx_{b',x} \wedge by_{b,y} \wedge by_{b',y}) \right)
\end{aligned} \tag{4}
$$

Goal property A state is a goal state if all blocks are on a goal position and no two blocks occupy the same (goal) position. Equation 5 shows this formally. In this *goals* is the set of goal positions and g^x denotes the column goal g is in, and g^y the row it is in.

$$
\begin{aligned}
&\bigwedge_{g\in goals} \bigvee_{b\in blocks} bx_{b,g^x} \wedge by_{b,g^y} \\
&\qquad \wedge \left(\bigvee_{x\in cols, y\in rows} \bigvee_{b\in blocks} \bigvee_{b'\in blocks\setminus\{b\}} \neg(bx_{b,x} \wedge bx_{b',x} \wedge by_{b,y} \wedge by_{b',y}) \right)
\end{aligned} \tag{5}
$$

Initial property The initial property describes a state in which all blocks and the man are in their initial position on the board (and nowhere else). Equation 6 shows this. In this $init^x_b$ denotes the the initial column of block b, $init^y_b$ denotes the initial column of block b. $init^x_m$ denotes the initial column of the man and $init^y_m$ the initial row of the man.

$$\begin{aligned}
&\bigwedge_{x \in cols, y \in rows} \bigwedge_{b \in blocks} \Big(\\
&\qquad ((x = init^x_b \implies bx_{b,x}) \wedge (x \neq init^x_b \implies \neg bx_{b,x})) \\
&\qquad \wedge ((y = init^y_b \implies by_{b,y}) \wedge (y \neq init^y_b \implies \neg by_{b,y}))\Big) \\
&\wedge \bigwedge_{x \in cols, y \in rows} \Big(((x = init^x_m \implies mx_x) \wedge (x \neq init^x_m \implies \neg mx_x)) \\
&\qquad \wedge ((y = init^y_m \implies my_y) \wedge (y \neq init^y_m \implies \neg my_y))\Big) \qquad (6)
\end{aligned}$$

2.2 Implementation

As stated in the introduction we have implemented our solver using the Sylvan ROBDD library. The different properties and the transition relation are all expressed as ROBDDS. The algorithms presented in the lecture are used to asses whether or not a screen satisfies Equation 1.

Parts of the formulas presented in Section 2.1 need not be encoded in the ROBDD since they are static. For example when considering Equation 3 the generated ROBDD does not contain a check for $x \neq max_c$, instead if $x = max_c$ the transition is not generated.

Not complete Sadly not all of the above is actually available in our implementation. Due to time constraints and trouble setting up Sylvan our implementation is not a finished product. In Section 5 we will elaborate a bit more on this trouble and why these have resulted in an unfinished implementation.

Our current implementation dynamically creates all needed variables for the input screen and dynamically creates an (incomplete) transition relation. The goal, error and initial properties are not yet dynamically created and are instead given as constant formulas. Our implementation is then capable of calculating the result of Equation 1, however because the transition formula is incomplete this result is not yet meaningful.

Counter examples or Witnesses Our implementation in its current form does not provide a counter example or witness, nor is the transition set up to keep track of such information. This means that with the current design the implementation is not able to generate a LURD string. This capability could be added by annotating the transition relation with variables which track what the last direction the man traveled in was, and then keeping track of these variables in the function which computes the least fixed point.

3 Results

Because the actual implementation is not finished we are sadly unable to discuss much results. We can not comment on either the capabilities or performance of our solver as we can not actually test the solver using any of the provided example screens.

However we can comment on the features of our implementation if it actually completely implemented the formulas and methods mentioned in Section 2. If this were the case then our implementation would be able to read an arbitrary (valid) Sokoban screen and be able to determine whether or not the screen is solvable.

4 Usage

To build our solver it is sufficient to run the included Makefile using the `make` command. This assumes that Sylvan and all its dependencies are correctly installed and on the PATH. In particular it assumes that the file `libsylvan.a` is on the PATH, it will also search for `libsylvan.a` in `./sylvan/src/` if it is not on the path.

After building our solver it is possible to run the solver using `./main <Sokoban sreen>`. It is possible to either give a path to a Sokoban screen or to input a screen directly. Example 7 will run our solver for the file `/screens/screen.2000`.

```
$ ./main /screens/screen.2000
```
(7)

When running the implementation it will output some diagnostic information to `std::cerr` and will state that it is not completely implemented on `std::cout`.

Some properties in our implementation are not yet based on the inputed screen, but our instead based on a minimal version, one with all the outside walls removed and an extra wall in the top left corner, of screen 2000. This file is also included as `screen.2000_minimal`.

5 Discussion

As stated in Section 2.2 our implementation is not complete due to trouble setting up Sylvan and other technical difficulties.

The technical troubles with Sylvan mostly had to do with the very limited documentation on Sylvan and trouble installing Sylvan. The latter is because even though Sylvan comes with CMAKE files and a claim that Sylvan will be added to the path, actually running CMAKE and `make install` will not correctly add Sylvan to the path on Mac OSX. Attempting to fix this and then working around the problem took quite a lot of time.

Something else which sadly took a lot of time was working with C++. Neither of us had a lot of experience in this language and familiarizing ourself with the language and its oddities took quite a lot of time as well, which made implementation of our solver considerably slower.