Software Verification – assignment 4: Sōkoban

In this lab assignment, you'll model Sōkoban using binary decision diagrams (BDDs) and use this model to solve the game.

Sōkoban (meaning "warehouseman" in Japanese) is a computer game developed by Imabayashi Hiroyuki in 1982. A few years later, the company Spectrum HoloByte released ports for home computers like the IBM PC and Commodore 64, and the game became popular worldwide.

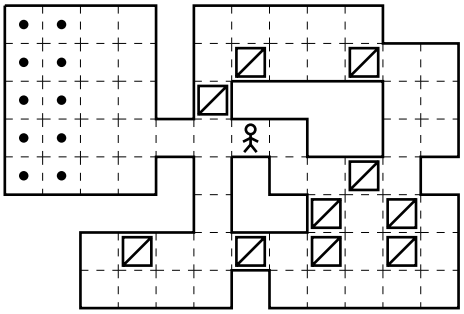
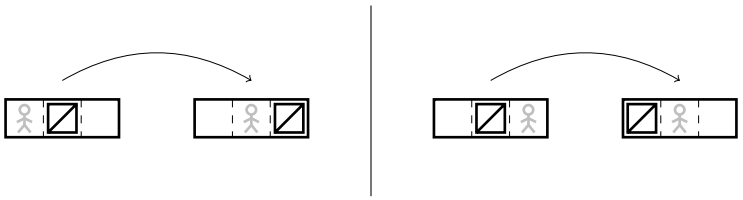


Figure 1: A level of Sōkoban. Goals are marked with a dot.

A level of Sōkoban consists of a tiled playfield containing a maze. The player controls a character that can move to adjacent tiles within this maze.

The maze also contains boxes: when the player is behind a box, the box can be pushed one tile forward, provided that space is empty. The goal of Sōkoban is to push all the boxes to designated goal tiles. The level is completed if all boxes are on a goal.

In this assignment, we consider a simplified version of Sōkoban, where the position of the player is not considered. A box can be moved if there is an empty tile both in front of and behind the box. The possible horizontal moves are pictured below; vertical moves follow similarly. Note that the move relation is local: only a subset of all tiles in involved in each move.



1 The BDD package

BDD operations are provided by the library Sylvan [1]. A Python binding is included with the assignment in the package `bdd`. To use this package, Sylvan version 1.5.0 needs to be installed.

> If you are working at LIACS or remotely through SSH, the tools and libraries needed for this course have already been installed. To access them, type:
>
> ```
> $ source /vol/share/groups/liacs/scratch/softveri/init.sh
> ```
>
> Your shell prompt will now display `(softveri)` to show the tools are accessible.

The BDD package provides the object `BDD`. BDDs are accessed by an `Edge` to its root node; nodes themself are represented by the object `Node`. The object `BDD` contains edges to the terminal nodes, `true` and `false`. It also provides the function `var(i)`, which returns an edge to a node with variable `i`, that has its high edge leading to True and its low edge to False (Figure 2).

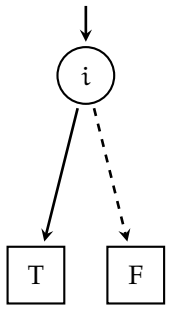


Figure 2: `dd.var(i)`.

An `Edge` provides several operations. These operations work on the BDD rooted at the given edge. The standard operators for negation (`~`), AND (`&`), OR (`|`), and XOR (`^`) can be used on them. Two additional important operations are `evaluations`, which is a generator that returns all evaluations contained in the BDD; and `image`, which provides the image function.

Using these operations, you can construct a BDD representing any arbitrary boolean expression. Remember you can use `help(bdd.Edge)` in an interactive session to explore all available functionality.

2 Solving Sōkoban with BDDs

The file `sokoban.py` contains a class `Sokoban` that initializes a Sōkoban model and can run reachability algorithms on it.

A particular configuration (board) of the level is represented as follows: each tile of the level has a variable. If the variable is false, that tile is empty; if it is true, there is a box on that tile. During initialization two variables (one unprimed and one primed) are created for each tile and put into the sets `varsSrc` (unprimed) and `varsDst` (primed).

Furthermore, the BDD `initial` is created: this BDD contains exactly one evaluation, namely that of the initial board position.

2.1 Task 1

Because we want to determine in which boards the level has been completed, the class has a BDD `winning` representing the set of all winning (completed) boards. In the case of simplified Sōkoban, there is only one such board. You will need to **construct the BDD that contains the winning board**.

2.2 Task 2

In order to find the reachable states, we need to be able to apply the move relation. Since the moves are local, the model can keep partial move relations (`movePartial`) instead of a single

complete one (`move`). In order to compare the performance of the two approaches, this model will construct both.

Construct the BDDs for the partial move relations. The complete move relation is constructed from the partial ones. In order to construct the partial relations, you can make use of the list of "triples" that has already been created. This list contains sets of 3 variables each, that represent three adjacent tiles (horizontal or vertical) that are involved in a move.

2.3 Task 3

To find all reachable states, there are several reachability algorithms. You'll implement a number of them and compare their performance. The algorithms are as follows. Here, R is the complete move relation; R_i is a partial move relation; and $*$ denotes repetition until convergence.

algorithm	mode	definition
BFS, complete relation	`bfs`	R^*
BFS, partial relations	`bfspart`	$(R_1 \cup \cdots \cup R_k)^*$
Chaining	`chaining`	$(R_1 \circ \cdots \circ R_k)^*$
Saturation-like	`satlike`	$(R_1^* \circ \cdots \circ R_k^*)^*$
Saturation [2]	`sat`	$(((R_1^* \circ R_2)^* \cdots \circ R_i)^* \cdots \circ R_k)^*$

Running `python sokoban.py --mode M --level N` (where `M` is one of the mode names given above, and `N` is a number between 1 and 3) will run a reachability algorithm, showing the node count along the way. At the end, the total run time will be displayed.

Implement the various reachability algorithms. Display of statistics is done by the `Stats` object; remember to call `stats.update` on every loop iteration of your algorithm.

Compare the performance (run time and node usage) of the algorithms you've implemented. Give results for all three levels. Try to give an explanation for the differences you're seeing.

3 Results

Write a short report answering the above questions and make a compressed archive of the report together with your source code.

- The archive must be in `.zip` or `.tar.gz` format.
- Be sure to include your name and student number in the report.

References

[1] Van Dijk, T. *Sylvan 1.2.0 documentation* (2017). Retrieved from `https://trolando.github.io/sylvan/`

[2] Van Dijk, T.; Meijer, J.; Van de Pol, J. *Multi-Core On-The-Fly Saturation*. Lecture Notes in Computer Science, vol. 11428 (2019): 58–75. Retrieved from `https://www.tvandijk.nl/pdf/2019parsat.pdf`