Analysis of Stochastic Behaviour in Sokoban

BRAM HAGENS, University of Twente, The Netherlands

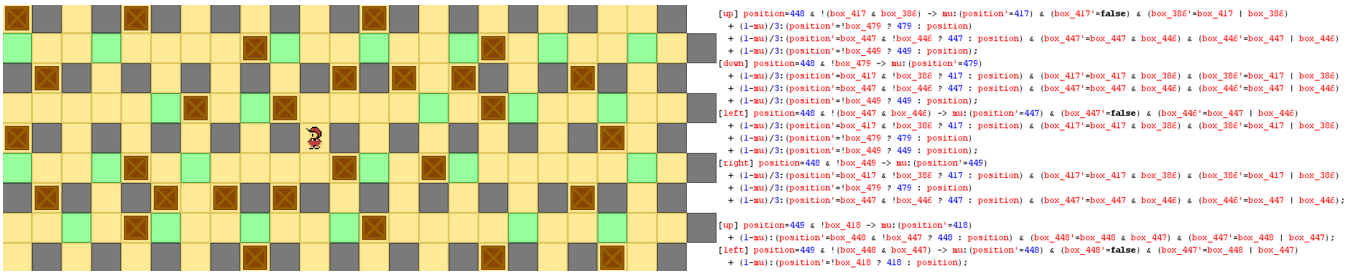


Fig. 1. A Sokoban level next to a part of its probabilistic model representation.

Sokoban is a challenging puzzle game for both humans and computers. By modifying the game to include stochastic elements, it can be used to benchmark different probabilistic model checkers. Probabilistic model checkers are used to analyse systems that exhibit probabilistic behaviour. To improve the real-world performance of these tools, being able to benchmark them with a large variety of models is essential. This research introduces a novel tool that can generate probabilistic models from Sokoban levels in the PRISM and JANI formats. With tens of thousands of Sokoban levels readily available online, many models can easily be generated and can be used as a part of a more extensive benchmarking suite, which in turn can be used to determine and compare various performance aspects of probabilistic model checkers during probabilistic model checker competitions. Additionally, the paper includes preliminary benchmarks for the Storm, Modest and PRISM model checkers, as well as an analysis of their performance and the properties of the models.

Additional Key Words and Phrases: Sokoban, probabilistic model checking, stochastic behaviour, model generation, JANI, PRISM, Modest, analysis.

1 INTRODUCTION

Systems that exhibit probabilistic behaviour exist everywhere: computer networks, security protocols, and traffic flows, among others. Using probabilistic model checking, a technique used to verify the correctness of a model of a stochastic system, one can compute bounds on the likelihood that a state will be reached. This can help prevent undesirable behaviour, such as a system crash or a deadlock. As these systems get more complex, it is important to use tools that can verify these models efficiently.

Sokoban is a puzzle game where a player pushes boxes into predefined locations. Solving Sokoban levels has been proven to be an NP-hard problem [4]. Introducing stochastic behaviour into this game will allow for the game to be used to benchmark probabilistic model checking tools. This is done by defining a probability that determines how often the model checker deviates from the shortest solution.

TScIT 37, July 8, 2022, Enschede, The Netherlands

The goal of this research is to generate probabilistic models from Sokoban levels in various formats so that these models can be used as part of a more extensive test suite to benchmark existing probabilistic model checking tools in competitions such as QComp[1].

2 BACKGROUND

2.1 Sokoban

Sokoban is a single-player transport puzzle game where a player pushes boxes around into predetermined locations. A level is completed when all boxes are pushed into the correct spots. The player can walk around and push boxes in 4 directions: up, down, left and right. The player is contained by walls that surround the level, which also prevent boxes from being able to be pushed into specific locations. Figure 2 shows an example of a Sokoban level that can be solved in 100 moves. It is possible to make a level unsolvable by pushing a box into a corner, which further adds to the difficulty of the game. Not only is solving Sokoban puzzles NP-hard, but it is also proven to be PSPACE-complete [3], making it significantly more challenging to solve than many other NP-hard problems. As a result, many levels exist that state-of-the-art solvers cannot solve.

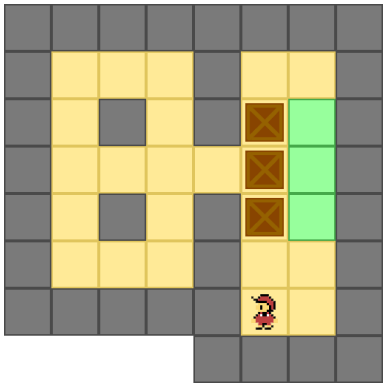


Fig. 2. An example Sokoban level. The grey tiles represent walls, the brown tiles represent boxes, and the green tiles are the targets.

[1] https://qcomp.org/

2.2 Probabilistic model checking

Model checking is a formal verification technique for analysing and verifying systems. Using model checking tools, one can verify the correctness of certain aspects of the model. For example, a model of a Sokoban level would include the player and box positions, and the solution can be found by querying the state where all boxes are in the solved position.

Probabilistic model checking is a similar technique used to analyse and verify systems that exhibit probabilistic behaviour. A model of a system is often described using a high-level modelling language that defines a set of rules and how these rules modify the state. From this high-level description, the probabilistic model checker will convert it into a probabilistic model by computing the complete set of reachable states.

The user can supply the model checker with properties, often defined in a property language based on temporal logic. In the case of probabilistic model checking, these are usually quantitative properties, for example, the probability of an event occurring. The model checker calculates these properties by analysing the complete state space generated previously. The full state space analysis yields accurate results, but this comes at the cost of increased computation costs.

PRISM [8], Storm [6], and Modest [5] are the three probabilistic model checkers chosen to be used in this research, mainly because they have all competed before in QComp.

2.3 Modelling languages

The models in this research will be described in both the PRISM language [8] and the JANI specification [2]. The PRISM language is used throughout the paper because of its simplicity and brevity.

The PRISM language allows the user to specify variables, which can be bounded, unbounded, and have an initial value:

```
x: [0..2];
y: int;
z: bool init false;
```

Transitions can be defined and will only be used if a guard on the left-hand side is satisfied:

```
[] x=1 & y=2 -> (z'=true);
```

The assignments on the right-hand side can be made based on probabilities as well:

```
[] x=2 -> 0.5:(z'=false) + 0.5:(z'=true);
```

Additionally, ternary statements are also supported and can be used to conditionally assign a value:

```
[] y=3 -> (x'=z ? 1 : 2);
```

Furthermore, PRISM implements a property specification language that allows the user to proof or analyse certain properties. It can be used to verify quantitive properties such as the minimum or maximum probability of reaching a state:

```
Pmin=? [F state]
Pmax=? [F state]
```

Time bounds can be introduced into these properties, which can be used to check if a state is reached within t time steps.

```
Pmax=? [F<=t state]
```

Rewards can be attached to transitions. The expected reward upon reaching a state can also be computed:

```
Rmin=? [F state]
Rmax=? [F state]
```

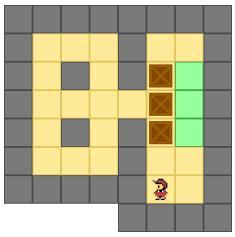
The full documentation for the PRISM language and property specification language can be found in the PRISM manual[2].

2.4 Sok format

.sok files[3] are files that contain Sokoban levels. There are currently tens of thousands of user-created levels in this format freely available online. Apart from a few inconsistencies, the format is easy to parse, and the large selection of levels makes it a good choice for this research. The contents of a .sok file are depicted in Figure 3.

```
# # # # # # # #
#         #     #
#   #     # b . #
#           b . #
#   #     # b . #
#         #     #
# # # # # p     #
          # # # #
```

(a) ASCII representation of a .sok-file. 'p' represents the player, 'b' represents a box, '.' represents a target location, and '#' represents a wall.



(b) Graphical representation of a .sok-file. The brown rectangles represent boxes, the green tiles represent target locations, and the gray tiles represent walls.

Fig. 3. A .sok-file in both its ASCII and graphical representation.

3 RESEARCH QUESTIONS

RQ1. To what extent do the stochastic additions influence the solutions generated by the model checker?

RQ2. On average, which of the three model checkers, Storm, Modest, and PRISM, is the best suited for solving probabilistic Sokoban models?

4 RELATED WORK

There exist many Sokoban solvers, such as Festival[4] and Sokolution[5]. The goal of this research however is not to create a solver for (a stochastic version of) Sokoban, but to create a model that a probabilistic model checker will solve.

A project similar to this research has been implemented that generates Sokoban models, which are then used to benchmark three different model checking tools: LTSmin, DiVinE, and nuXmv [10]. This research solely focused on the standard implementation of Sokoban and therefore benchmarked non-stochastic models with non-probabilistic model checkers. In contrast, this research only considers stochastic models and probabilistic model checking tools.

[2]PRISM manual - http://www.prismmodelchecker.org/manual/Main/Welcome
[3]http://www.sokobano.de/wiki/index.php?title=Sok_format
[4]https://festival-solver.site/
[5]http://codeanalysis.fr/sokoban/

Similarly, another project focuses on generating models for various logic puzzles, including Sokoban, for LTSmin [7]. This research is different from the mentioned project for the same reasons mentioned above.

5 MODELLING AND GENERATION

5.1 Rules

The models must abide by the following simple Sokoban rules:

(1) A player can walk a single step at a time in any of the four directions (up, down, left, right) if there is not a box or a wall at the destination.
(2) A player can push a box in any of the four directions (up, down, left, right) if there is a box at the destination, and the tile behind the destination is not a box nor a wall.

5.2 Encoding

The positions of the player and all boxes have to be encoded in a way that allows for a state to only be represented in one way, so that the state space will not be unnecessarily large. The position is stored as a simple integer bounded by the first and last reachable position on the board. For each reachable position, an additional flag is maintained that signals whether there is a box at that position. The encoding of the level depicted in Figure 4 is as follows:

```
position: [6..18] init 17;

box_6: bool init false;
box_7: bool init false;
box_8: bool init false;
box_11: bool init false;
box_12: bool init true;
box_13: bool init false;
box_16: bool init false;
box_17: bool init false;
box_18: bool init false;
```

There are nine box flags, one for each reachable position. As the box is in position 12 initially, the flag corresponding to this position one is set to true. The player can move freely between the reachable tiles, given that there is no box or wall in the way, and their position is bounded by the first and last reachable tile index. According to the bounds, the player can reach invalid positions, such as a wall. The transitions of the model prevent this from happening.

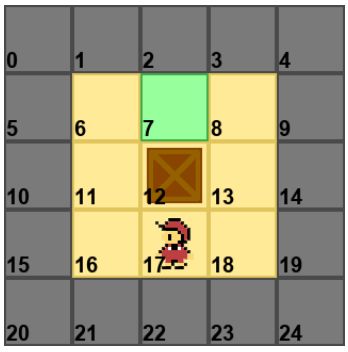


Fig. 4. A simple Sokoban level with a position index on each tile.

5.3 Non-deterministic transitions

The probabilistic model is a Markov decision process (MDP). This type of model allows for both probabilistic and non-deterministic transitions. The moves in the model description are specified in such a way that they will be chosen non-deterministically. The moves can be in any direction, as long as the puzzle's layout allows for this. If the player is positioned at a tile T, they can walk to any adjacent tile T', given that this tile is not a wall nor contains a box. If T' does contain a box, the player will have to push the box forwards to the adjacent tile T''. This is only possible if T'' is not a wall nor contains a box. This means that for any combination of (T, T'), a walk transition exists, while a push transition between (T, T') only exists if T'' is not a wall.

From position 12 in Figure 4, only walk transitions exist, as the push condition cannot be satisfied. The transitions look as follows:

```
[up]    position=12 & !box_7  -> (position'=7);
[down]  position=12 & !box_17 -> (position'=17);
[left]  position=12 & !box_11 -> (position'=11);
[right] position=12 & !box_13 -> (position'=13);
```

From all other reachable positions, the player can always push in one direction. Since the existence of a push transition into a direction also implies the existence of a walk transition into that same direction, the two moves are modelled as a single transition. This transition looks as follows from position 17:

```
[up] position=17 & !(box_12 & box_7) -> (position'=12)
 & (box_12'=false)
 & (box_7'=box_12 | box_7);
```

The other two transitions, left and right, are again walk-only transitions. There is no down transition, as there is a wall at position 22.

Using these two types of transitions, a model checker can determine the shortest solution to this non-deterministic model.

5.4 Probabilistic mistakes

Stochastic behaviour is added to the transitions, which forces the model checker to choose a suboptimal move with a predetermined probability. In this case, a suboptimal move is one that in most cases will increase the number of steps taken to solve the level. This behaviour is added by modifying the transitions so that with a probability of μ, the player makes the optimal move. The remaining probability, $1 - \mu$, is divided into equal parts depending on the possible amount of alternative moves the player can make from that position. If the player cannot make the specified alternative move due to the current state of the board (for example because there are boxes in the way), the player does not move to prevent the model from deadlocking due to an invalid move.

The probabilistic up transition from position 12 in Figure 4 is modelled as follows:

```
[up] position=12 & !box_7 -> mu:(position'=7)
 + (1-mu)/3:(position'=!box_17 ? 17 : position)
 + (1-mu)/3:(position'=!box_11 ? 11 : position)
 + (1-mu)/3:(position'=!box_13 ? 13 : position);
```

From position 12, there are four possible walk moves: up, down, left, and right. The primary move, up in the example above, is chosen with a probability of μ. The three alternative moves are chosen

with a probability of $\frac{1-\mu}{3}$. As only walk moves are permitted from this position, it is possible that a move cannot be executed because of a box being in the way. As mentioned above, the model must not deadlock due to invalid moves, thus the alternative moves are modelled using a ternary statement so that they are only executed if they are valid moves.

The concept remains the same if push moves are also permitted, however, due to the need to check and reassign more variables, these are more complex. The assignments for the walk/push move from position 12 to position 7 in Figure 4 are modelled as follows; for the sake of brevity, the guards and other irrelevant assignments are left out:

```
(position'=box_12 & !box_7 ? 12 : position)
 & (box_12'=box_12 & box_7)
 & (box_7'=box_12 | box_7)
```

The probabilistic transitions have a significant consequence on the solvability of the levels. The model now forces the model checker to make a suboptimal move with the probability of μ. While invalid moves will not be executed, it is now possible to make the level unsolvable by pushing a box into a position from which it can no longer be moved.

5.5 Rewards

PRISM, Modest and Storm all support MDPs with transition rewards. This means that a certain cost can be attached to a transition, which can be used to calculate the total amount of moves taken to reach the required state. However, if the probability of reaching this state is less than 1, the rewards converge to infinity. Support for conditional reward properties for MDPs is missing in all three model checkers, meaning with the model described above, rewards cannot be used due to the fact that a mistake could make the level unsolvable. However, to still be able to calculate the expected amount of moves required to solve the level, the generation tool can also output models where a mistake can only be a walk move, not a push move. This means a level will never be unsolvable, as a walk move can always be undone; thus, the rewards work as expected.

5.6 Generation

The model generation tool generates probabilistic models as described above from Sokoban levels supplied by the user as .sok files. The generated models are described by the PRISM language and the JANI specification so that a wide range of probabilistic model checkers can be targeted. The generated models contain an undefined constant, μ, that defines the probability that the player makes the best move. A high-level overview of the workings of the tool is as follows:

(1) Parse a .sok file into an intermediate representation that better suits the generation process.
(2) Run a depth-first search starting from the player's position to find all reachable tiles.
(3) Generate the variables used to encode the player and box positions.
(4) Iterate over all possible moves for each reachable tile and generate the transitions between the states.

6 METHOD

6.1 Benchmarks

The PRISM, Modest and Storm model checkers will be benchmarked using the Microban level set[6] by David W. Skinner. This set is part of the Large Test Suite and Open Test Suite[7], both often used to benchmark Sokoban solvers. The set contains 155 levels that can be solved quickly by specialised solvers. However, solving the stochastic variants using probabilistic model checkers is a much more computationally expensive task, as the entire state space has to be explored.

For each level and model checker, the same property is benchmarked: the maximum probability that the target state is reached. During the benchmark, the peak RAM usage is recorded, and the total time until the answer is calculated is measured. All levels are benchmarked twice: The first run is done with a μ value of 0.3, the second run is done with a μ value of 0.9. This is done to further identify differences in performance between the model checkers.

The PRISM model checker only supports models in the PRISM language. Therefore, it will be benchmarked using the generated models in the PRISM language. The Modest and Storm benchmarks are run on the generated models defined according to the JANI specification.

The benchmarks are run inside a virtual machine (VM) to ensure consistent results. This allows for an equal amount of resources to be allocated to the model checkers. The VM runs on Ubuntu 64-bit 20.04.4 LTS, and has access to four cores running at 4.0 GHz. Each model checker has an allocated 8 GB of RAM. This memory limit is assigned to the process itself, not the VM.

To prevent the benchmarks from running for an extensive amount of time, a 5-minute timeout is used for each level. Failing to find the maximum probability of reaching the target state within these 5 minutes means the benchmark has failed.

6.2 Analysis

To better understand the implications that the stochastic additions have on the solutions generated by the model checkers, the following properties are analysed:

(1) What is the maximum probability of the target state being reached?
(2) What is the maximum probability of the optimal solution being reached?
(3) What is the expected number of moves in which the target state will be reached?

The probability that a player makes the best move, μ, is varied so that the effect of the stochastic additions can be made clear. The analysis is done on the entirety of the Microban level set, however, a model is skipped if the computation time for the analysis exceeds ten minutes. The probabilities are computed per level and then averaged to create results that are representative of the entire level set.

[6] Puzzles by David W. Skinner - http://www.abelmartin.com/rj/sokobanJS/Skinner/David%20W.%20Skinner%20-%20Sokoban.htm
[7] Test suite overview - http://sokobano.de/wiki/index.php?title=Solver_Statistics

		All		Shared	
Checker (engine)	Num. solved	Avg. runtime solved (s)	Avg. peak memory use solved (MB)	Avg. runtime solved (s)	Avg. peak memory use solved (MB)
PRISM (hybrid)	127	27	296	16	260
Storm (hybrid)	130	35	2792	18	2831
Modest (mcsta)	121	26	267	23	255

Table 1. Result of the benchmarks on the Microban level set for $\mu = 0.3$. The *All* column takes all solved levels into account, while the *Shared* column only counts the levels that are solved by all model checkers.

		All		Shared	
Checker (engine)	Num. solved	Avg. runtime solved (s)	Avg. peak memory use solved (MB)	Avg. runtime solved (s)	Avg. peak memory use solved (MB)
PRISM (hybrid)	128	15	299	13	278
Storm (hybrid)	138	37	2811	26	2853
Modest (mcsta)	131	22	518	22	524

Table 2. Result of the benchmarks on the Microban level set for $\mu = 0.9$. The *All* column takes all solved levels into account, while the *Shared* column only counts the levels that are solved by all model checkers.

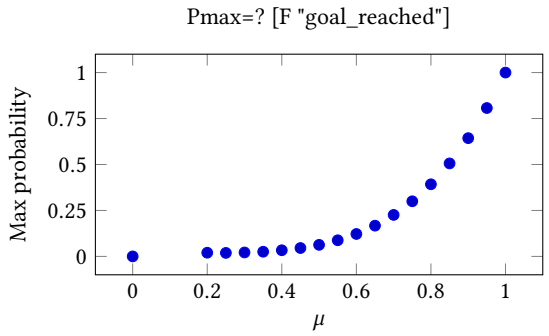


Fig. 5. Average maximum probability of the goal state being reached based on the value of μ for the Microban level set (n=92).

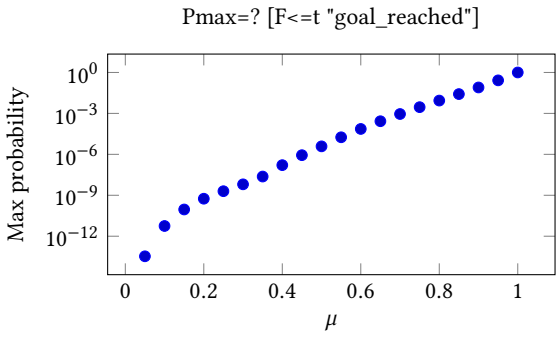


Fig. 6. Average maximum probability of the goal state being reached in t moves based on the value of μ for the Microban level set, where t is the minimum amount of moves required to solve the level (n=103).

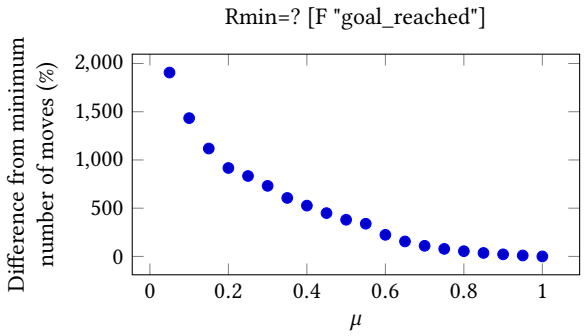


Fig. 7. Average expected number of moves required to solve the level based on the value of μ for the Microban level set, expressed relative to the number of moves to reach the optimal solution (n=84).

7 RESULTS

The results for RQ1 are presented in Figure 5, Figure 6, and Figure 7. The results for RQ2 can be found in Table 1 and Table 2. All unsolved models were due to the time constraints imposed on the experiment, as none of the model checkers came close to the memory limit of 8 GB.

8 DISCUSSION

8.1 Analysis of stochastic additions to Sokoban

The stochastic additions greatly influence the generated solutions by the model checkers. As evident from Figure 5, making mistakes will significantly decrease the probability of being able to complete the level. This is expected, as a single push in the wrong direction will often cause a level to become unsolvable. The probabilities for $\mu < 0.2$ have not been calculated due to the high level of iterations required to calculate the probability of solving the level with small μ-values.

Figure 6 shows that the player often cannot recover and find an optimal solution if a mistake was made. However, in very few cases, there are two or more optimal solutions and a mistake will put the player on the path from one optimal solution to another optimal solution.

As expected, a strong correlation exists between μ and the expected number of moves to solve a level, as shown in Figure 7. The fewer mistakes the player makes, the fewer moves are required to solve the level. The irregularities in this figure can be attributed to the smaller sample size for this experiment. Increasing the sample size should smoothen out the curve in the plot. The generated models are modified for this specific property, as explained in Subsection 5.5. This means that in most cases, a mistake is simply corrected by undoing the last move and carrying on with the optimal solution. If conditional properties with rewards were supported, the expected number of moves would likely be higher, as in most cases a wrong push move will need multiple moves to correct.

8.2 Benchmarks of PRISM, Storm, and Modest

For both $\mu = 0.3$ and $\mu = 0.9$, Storm using the hybrid engine was able to solve the most Sokoban levels out of the test set within the 5-minute time limit. However, this comes at the cost of runtime performance and memory usage, as it performed the worst in these classes for both the *All* and *Shared* categories.

The most memory-efficient model checker for $\mu = 0.3$ is Modest using the mcsta engine in both the *All* and *Shared* categories. For $\mu = 0.9$, PRISM is the most memory-efficient in both categories. None of the model checkers exceeded the 8 GB memory limit before the timeout of 5 minutes was reached.

In general, PRISM with the hybrid engine is the fastest model checker. For the *All* category for $\mu = 0.3$, Modest was able to beat the average runtime by one second.

This proves that Storm is the most effective probabilistic model checker for this test set according to these measurements, as it was able to solve the greatest amount of models within the time and memory constraints out of the three model checkers. All model checkers were run with their default configuration, meaning there may have been configurations available that would yield more optimal results. This is outside the scope of this research.

As evident by the results analysed in Subsection 8.1, a lower value of μ reduces the probability of a valid solution being found. This can also be seen by comparing Table 1 and Table 2: On average, levels were solved quicker for $\mu = 0.9$ than for $\mu = 0.3$.

9 CONCLUSION

Out of the three model checkers tested, Storm is the most capable of solving the probabilistic Sokoban levels. However, this comes at the cost of increased memory usage and a higher average time to solve levels. None of the model checkers ran out of memory within the time constraints. However, Storm used more than five times the amount of memory than the other tested model checkers.

The stochastic additions to the Sokoban game greatly influence the solutions generated by the model checkers. On average, the parameter that controls how often a player makes a mistake, μ, determines how difficult the model will be to solve for the model checker. The higher the probability of making a mistake, the longer it takes for the model checker to solve certain properties, such as finding the maximum probability of completing the level.

10 FUTURE WORK

The generated can be optimised further by reducing the bounds of the player position. Currently, it is bound between the lowest and highest reachable position index, but it is possible that many of these values in between the bounds are walls and thus not valid player positions. A solution to this would be to map all the positions on the board to a unique integer in the range $[0..n)$, with n being the number of reachable tiles on the level. That way, all invalid positions are no longer considered, and the player's position will be bounded between 0 and $n - 1$.

In some cases, transitions are generated with guards that can never be satisfied. This can be a walk move towards an immovable box, or a push move where it is impossible for the player and box to be in those specific positions. Additionally, immovable boxes on a goal position are still considered when determining if a level is solved or not; these can also be optimised away.

As described in Subsection 5.5, the model is modified to determine the expected number of moves. This influences the resulting amount of moves and thus is not optimal. This can be solved by using conditional reward properties to find the expected amount of moves for all solutions that reach the goal state. Baier et al. give an intricate solution for this [1]. However, this method is not implemented in PRISM, Storm or Modest. An extension for PRISM is available that supports conditional properties, but rewards are not supported [9].

REFERENCES

[1] Christel Baier, Joachim Klein, Sascha Klüppelholz, and Sascha Wunderlich. 2017. Maximizing the Conditional Expected Reward for Reaching the Goal. In *Tools and Algorithms for the Construction and Analysis of Systems*. Springer Berlin Heidelberg, 269–285. https://doi.org/10.1007/978-3-662-54580-5_16

[2] Carlos E. Budde, Christian Dehnert, Ernst Moritz Hahn, Arnd Hartmanns, Sebastian Junges, and Andrea Turrini. 2017. JANI: Quantitative Model and Tool Interaction. In *Tools and Algorithms for the Construction and Analysis of Systems*. Springer Berlin Heidelberg, 151–168. https://doi.org/10.1007/978-3-662-54580-5_9

[3] Joseph Culberson. 1997. *Sokoban is PSPACE-complete.* https://doi.org/10.7939/R3JM23K33

[4] Dorit Dor and Uri Zwick. 1999. SOKOBAN and other motion planning problems. *Computational Geometry* 13, 4 (1999), 215–228. https://doi.org/10.1016/S0925-7721(99)00017-6

[5] Arnd Hartmanns and Holger Hermanns. 2014. The Modest Toolset: An Integrated Environment for Quantitative Modelling and Verification. In *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014. Proceedings (Lecture Notes in Computer Science, Vol. 8413)*, Erika Ábrahám and Klaus Havelund (Eds.). Springer, 593–598. https://doi.org/10.1007/978-3-642-54862-8_51

[6] Christian Hensel, Sebastian Junges, Joost-Pieter Katoen, Tim Quatmann, and Matthias Volk. 2020. The Probabilistic Model Checker Storm. https://doi.org/10.48550/ARXIV.2002.07080

[7] Bram Kamies. 2016. Solving Logic Puzzles using Model Checking in LTSmin. In *24th Twente Student Conference on IT, Enschede, The Netherlands.*

[8] M. Kwiatkowska, G. Norman, and D. Parker. 2011. PRISM 4.0: Verification of Probabilistic Real-time Systems. In *Proc. 23rd International Conference on Computer Aided Verification (CAV'11) (LNCS, Vol. 6806)*, G. Gopalakrishnan and S. Qadeer (Eds.). Springer, 585–591.

[9] Steffen Märcker, Christel Baier, Joachim Klein, and Sascha Klüppelholz. 2017. Computing Conditional Probabilities: Implementation and Evaluation. In *Software Engineering and Formal Methods*. Springer International Publishing, 349–366. https://doi.org/10.1007/978-3-319-66197-1_22

[10] Kasper Wijburg. 2016. Comparing Model Checkers Through Sokoban. In *24th Twente Student Conference on IT, Enschede, The Netherlands.*