Proceedings of the 6th Annual North American Conference on AI and Simulation in Games (GAMEON-NA), pp. 5-12, 2011.

PROCEDURAL GENERATION OF SOKOBAN LEVELS

Joshua Taylor and Ian Parberry
Dept. of Computer Science & Engineering
University of North Texas
Denton, TX, USA
Email: ian@unt.edu, JoshuaTaylor@my.unt.edu

KEYWORDS
Procedural generation, Sokoban, puzzle.

ABSTRACT

We describe an algorithm for the procedural generation of levels for the popular Japanese puzzle game Sokoban. The algorithm takes a few parameters and builds a random instance of the puzzle that is guaranteed to be solvable. Although our algorithm and its implementation runs in exponential time, we present experimental evidence that it is sufficiently fast for offline use on a current generation PC when used to generate levels of size and complexity similar to those human-designed levels currently available online.

INTRODUCTION

In puzzle games the level design can make the difference between a game that is trivially easy or completely impossible. It is difficult to find the balance between the two, where the levels are challenging but still solvable. Here we present an algorithm that automates the generation of Sokoban puzzles of a given difficulty.

Sokoban is a puzzle game played on a rectangular grid. The goal is for the player's avatar to push boxes onto marked goal squares. The challenge comes from the placement of the walls, goals and boxes and the restriction that the avatar is only strong enough to push one block at a time and cannot pull blocks at all. The simplest way of explaining it is to show a picture, for example Figure 1, which shows a level with a single box and a single goal. This figure and the other screenshots in this paper are from JSoko (Damgaard et al. 2010).

Culberson has shown (Culberson 1998) that Sokoban is PSPACE-complete, meaning that it is in a sense at least as difficult as almost any one-player game. (Most games that are hard in this sense are for two or more players.) This, together with its simple rules, makes Sokoban a challenging candidate for procedural generation of puzzle instances. Completely random Sokoban levels are extremely likely to be unsolvable, or if they are solvable, then they are likely to be very easy. Even hand-made levels suffer from this problem unless the person making the level is an experienced Sokoban level designer.

Most other research done on Sokoban has been geared towards solving existing Sokoban puzzles (Junghanns and Schaeffer 1997, Botea et al. 2003). Some work has also been done on estimating the difficulty of a given Sokoban problem (Jarušek and Pelánek 2010a, Ashlock and Schonfeld 2010). Relatively little research has been done on generating new Sokoban levels (Murase et al. 1996, Masaru et al. 2003), although there are several existing generator programs (Mühendisi Accessed 2011). Additionally, there has been some research on generating levels for other PSPACE-complete puzzle games (Servais 2005).

The interested reader is invited to visit our Sokoban Generator webpage (Taylor and Parberry 2011) for supplementary information. This includes some more detailed instructions for the novice on how to play Sokoban, several hundred procedurally generated Sokoban levels, a link to an open source Java implementation called JSoko on which to play-test those levels, a short video showing JSoko's solution to some of our levels, some larger color images from this paper, and the archived data from the experiments performed to generate the performance data for the tables and figures that will appear later in this paper.

OBJECTIVES

Any procedural generation system should satisfy several criteria (Doran and Parberry 2010): *novelty*, *structure*, *interest*, *controllability* and *speed*. Our Sokoban level generator possesses these qualities as follows: Novelty: The generator produces a new and different puzzle on each run. Structure: The puzzles are nontrivial yet not impossible to solve, without requiring verification of this by use of an automated solver. Interest: Players should find the prospect of solving the puzzles attractive. This is left for future work; Development is currently underway. Controllability: Designers have control over the size and difficulty of the generated levels. Speed: The generator can run offline on a modest computer and generate at least one challenging puzzle, or hundreds of nontrivial puzzles per day.

Our primary aim is to generate reasonably difficult, but not impossible, Sokoban levels. There are two reasons for this. Firstly, these levels are the kind that are hard for a human to make, at least without a lot of experience. Secondly, we believe that in puzzle games, difficulty is related to interest.

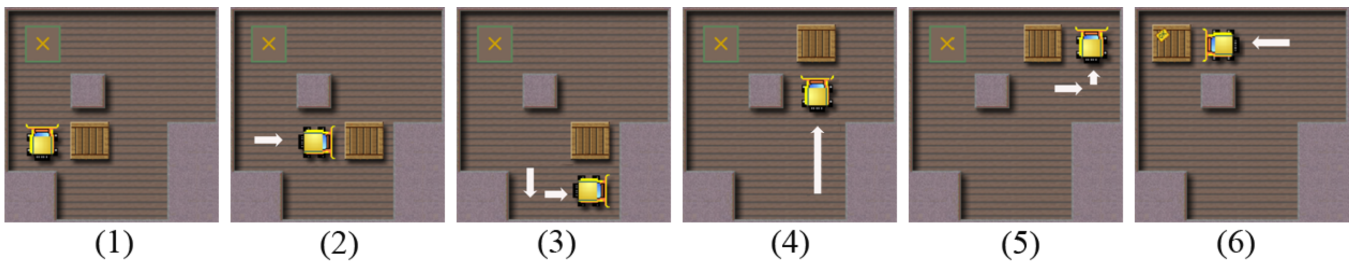


Figure 1: Solving a simple Sokoban level. The aim is to push the box to the square marked with the "X" at top left using the yellow bulldozer. The white arrows indicate player actions. The six images show, from left to right, (1) the start configuration, (2) push the box one place right, (3) reposition the player below the box, (4) push the box two places up, (5) reposition the player to the right of the box, and (6) push the box two places to the left into the final configuration.

Interesting puzzles are neither too difficult nor too easy (at least for most players), and yet it is these puzzles that are the most difficult to generate.

METHOD

The idea of working backwards from the goal towards the start is not new (Takes 2007), but previously it has only been used to *solve* existing levels. Here we use that idea to *generate* new levels. Our algorithm consists of three high-level steps, each of which will be described in more detail in its own subsection below.

1. Build an empty room.
2. Place goals in the room.
3. Find the state farthest from the goal state.

EMPTY ROOMS

To build an empty room, we use a method somewhat similar to that of (Murase et al. 1996). We begin by choosing a width and height for the level. This is done by simply picking a random number within a user-specified range. The level is then partitioned into a grid of 3 $\times$ 3 blocks. Each block is then filled in using a randomly chosen and randomly rotated or flipped template. The templates consist of a 3 $\times$ 3 pattern of walls and floors surrounded by a border of blanks, walls and floors (see Figure 2). The borders cause neighboring templates to overlap. A non-blank tile must match any pattern it overlaps, whether it is placed before or afterwards.

This overlap helps to create interesting levels by preventing some bad configurations from being generated. For example, the pattern consisting of a single wall in the middle surrounded by a ring of floor will become a large dead-end unless there are at least two floor tiles adjacent to each other and that pattern. Since the templates are randomly rotated and flipped before being placed, this is very easy to enforce by simply placing two adjacent floor tiles in the border of that template and leaving the rest blank.

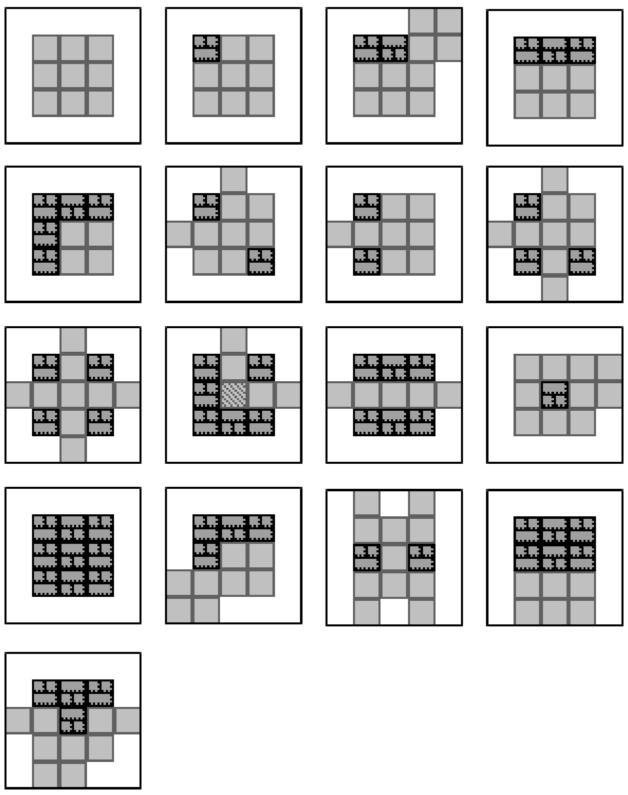


Figure 2: Templates used to design an empty room.

If the generator places blocks in such a way that it cannot fill in one of the cells with any of the available templates, it will discard that attempt and start over. The run time of this step is very small compared to the rest of the algorithm, so even throwing away several partial room shapes does not create any noticeable loss of speed.

Finally, there are some post-processing checks to make sure the level will work well with the remaining steps. Any level that fails one or more of these tests is discarded.

- The level is checked for connectivity. There should be one contiguous section of floor. There is one special case here. The templates that allow the player to pass through, but will not allow a box to pass, are checked as if there was a wall tile separating the two sides. This only affects this check, and that tile is counted as a floor tile in all other cases.

- Any level that has a 4×3 or 3×4 (or larger) section of open floor is discarded. By observation, such levels tend to make levels with very bushy, but not very deep state spaces. This makes it very hard to generate the level, but not much harder to solve it.
- The level must have enough floor space for the planned number of boxes, plus the player and at least one empty space.
- If the level contains any floor tiles surrounded on three sides by walls, it is discarded. This is a somewhat aesthetic choice, but such tiles are either obviously dead space if there is no goal there, or an easy place to get boxes out of the way otherwise, so we think it improves the quality of the resulting levels somewhat.

PlACING GOALS

Goal placing is done by brute force, trying every possible combination of goal positions. This is admittedly very inefficient. Many human made levels place the goals in certain patterns, such as a rectangle of contiguous goals, but by doing a brute force search for the best places to put the goals, some obvious patterns emerge.

One pattern that seems to hold for most, but not quite all, of the levels generated so far is that the goals are touching either a wall or another goal. Whether forcing this would provide a significant speed-up, or a significant drop in the quality of the resulting levels has not yet been investigated.

Our generator uses a timer that checks it has exceeded its allotted time. If it has, it will terminate and return the best result so far. To help ensure that that result is something interesting, even if not the best, the positions for the goal crates are checked in random order. This is done by creating a shuffled list of the empty spaces on the board.

FARTHEST STATE

For each placement of the goals the system finds the farthest state from that goal state, that is, the state with the longest shortest path from itself back to the goal. Over all goal states, the farthest farthest state is returned as the output of the generator. Thus, the distance from the goal state to the start state is the metric by which we judge the resulting levels, as well as influencing the algorithm used to search the state space. The definition of *distance* is crucial. In Sokoban there are four common distance metrics. The simplest is just the move count, incremented every time the avatar moves. As a measure of the difference between states, this does not work very well. Just making a large labyrinth with only one obvious solution will still give a high distance, but will be fairly trivial in the end.

The number of box pushes, incremented each time the avatar moves into a square containing a box, is not much different than the move count. A level that required the player to push boxes down long hallways would give a high score, but again would not be difficult, just tedious.

The box lines metric is more interesting. It counts how many times the player pushes a box, but any number of pushes of the same box in the same direction only count as a single box line. From our observations, the number of box lines corresponds fairly well with the difficulty of the resulting level. We are currently using the box line metric in our generator.

The last metric is box changes. It counts how many times the player stopped pushing one box, in any direction, and began pushing another. This may be an even better measure of difficulty, and may improve the overall speed of the generator, but it is more difficult to implement.

Any metric except for the move count allows us to abstract out the avatar position. Instead of keeping up with which square the avatar is in, we keep up with which group of contiguous floor squares it could reach. This abstraction provides a significant decrease in the time it takes to generate the set of further states.

All of this is done in reverse compared to how Sokoban is played. The reason for this is to prevent the generator from having to consider invalid moves. Any state reachable when moving in reverse will be solvable when played normally.

Unfortunately, none of the usual search algorithms are suitable for this problem. The most obvious way to find the farthest state is to use a breadth-first search, returning the last state found, but since moves in Sokoban are not reversible, the only way to prevent repetitions is to store a list of all visited nodes. For Sokoban, or any other PSPACE-complete problem, this will quickly fill up the available memory. Iterative deepening is unsuitable for similar reasons. Informed searches, like A* or IDA*, are unsuitable because the target is very vaguely defined, meaning we have no clear indication when to stop the search. To get around these problems, we use a form of iterative deepening twice, trading off the high memory requirements for a somewhat slower algorithm.

```
proc Go(goal) $\equiv$
  startSet := MakeStartSet(goal);
  resultSet := startSet;
  depth := 1;
  do
     prevSet := resultSet;
     resultSet := Try(startSet, resultSet, depth);
     if resultSet = $\emptyset$ then exit fi;
     depth := depth + 1;
  od;
  Go := (prevSet, depth).
```

```
proc Try(startSet, prevResults, depth) $\equiv$
  resultSet := Expand(prevResults);
  tempSet := startSet;
  for $i := 1$ to depth do
    resultSet := resultSet − tempSet;
    tempSet := Expand(tempSet);
  od;
  Try := resultSet.
```

MakeStartSet takes the goal state and places the player into each available contiguous floor area. Expand takes a set of states and returns the set of states one step farther. What those states are depends on which metric is being used, which is why the choice of metric has such an impact on the running time. Go is almost a standard iterative deepening algorithm. It takes the goal, calls MakeStartSet to set things up and then calls Try one depth at a time. What is different here is the end condition. Go calls Try until it fails and then returns the previous set of results. Try takes the starting set, the previous results and a target depth. It then calls Expand on the previous results. Then it starts over expanding the start set, subtracting that from the results. What is left after the target depth has been reached is all of the nodes that can be reached in depth steps, but no sooner.

GENERATING LEVEL SETS

Our generator returns the set of all levels that are as far from the goal state as possible. This can be anywhere from one to a few hundred levels, and some are obviously better than others. We have an additional layer over the generator that attempts to select a good level from those generated and then collects the results of several runs into a level set. Additionally, it makes an attempt to reject levels that are much too easy, or are too similar to levels already in the level set. Finally, when the target number of levels has been reached, it attempts to sort them by some measure of difficulty.

The questions of what is *better* and what is *not good enough* both rely on the same, rather arbitrary, measure. We take the candidate level and give it a score based on a number of factors. To begin with, the score is $100 \cdot (\text{pushes} - \text{number of sibling levels} + 4 \cdot \text{lines} - 12 \cdot \text{boxes}) + \text{Random}(0, 300)$, where pushes is the number of box pushes in the solution, sibling levels is the number of other levels the generator found at the same depth, lines is the number of box lines in the solution, boxes is the number of boxes in the level and Random is just a random number between the given values. While this is, again, fairly arbitrary, the rationale is that both more pushes and more lines make the level more difficult, while levels with many sibling levels seem to be less interesting, just by observation. The number of boxes is subtracted not because more boxes makes the level less interesting, but because the number of lines needs to exceed the number of boxes by a certain factor for the level to have a better chance of being a good level. The random factor is mainly there to break ties.

Some other checks are made after getting the base score. Any trapped box is worth -100000 points, which is almost guaranteed to get the level rejected. Any box touching a wall is worth -150 points, a box touching the player is worth 50 points, and a box touching another box is worth 30 points. Finally, a goal area touching a goal area is worth 30 points. These constants can be adjusted by the individual designer to suit his or her intuition about features possessed by good Sokoban levels. Any level with a final score of 0 or less is rejected. The base score is quite a bit higher than the scores for most of the various other checks though, so not many levels are rejected at this point. This same score is then used to choose the best level from those generated, assuming any are left.

Once a level has passed all of the other tests, the program checks to see whether or not it is too similar to another level already in the set. Currently, this just removes the player and checks for exact matches for all of its rotations and reflections. This still generates a few levels that a human would consider too similar, so there is still more work to be done here.

Assuming no other level is too similar, the level is added to the set. When the set gets to target size, it is sorted by difficulty and written to a file. The measure of difficulty we currently use is $\text{lines} \cdot \log \text{lines} + \log \text{time} - \text{lines}/\text{pushes}$. This is based on our observations that the number of box lines is the most important factor. time is the time taken to generate that level, in seconds, and is mainly used as a tie breaker. The lines/pushes factor is small correction that favors levels with shorter box lines rather than long corridors.

EXPERIMENTAL RESULTS

We have implemented and tested our new algorithm for the automatic generation of Sokoban levels. Figure 6 contains some screenshots of sample levels generated. The reader is invited to visit our Sokoban Generator webpage (Taylor and Parberry 2011), where he or she can download some of our level sets and try them out.

Our algorithm is certainly suitable for offline use in a level-a-day style game. In practice it can generate several levels over the course of a day depending on how much CPU power it is given and how large the desired levels are. The theoretical run time for the generation of one level is roughly

$$b\binom{s}{b}^2,$$

where b is the number of boxes and s is the number of empty spaces. This is feasible for a fairly small number of boxes that might occur in practice. We have been able to generate levels with 4 boxes within a 2×3 level outline within a few hours.

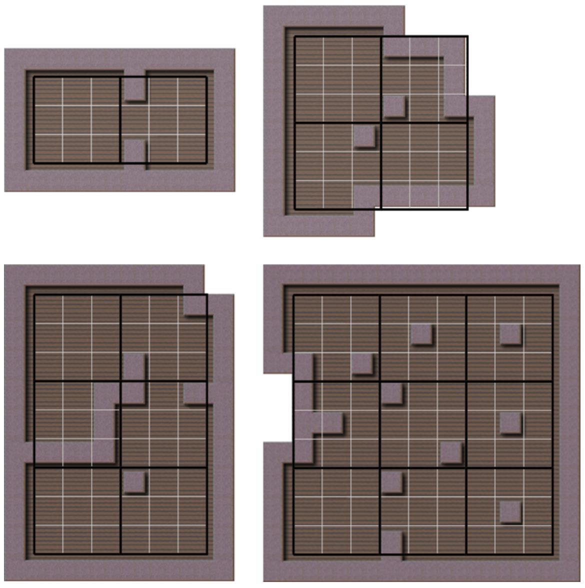


Figure 3: Examples showing the relative sizes of 1×2, 2×2, 2×3, and 3×3 Sokoban levels.

We ran experiments measuring the average run-time generating 10 random puzzles of each size 1×2, 2×2, 2×3, and 3×3. See Figure 3 for an indication of the relative sizes of these levels. All of these results are from runs on an Intel i7 3.2 GHz quad-core processor with hyperthreading. Our system is not written to make use of the extra cores, but we run several independent copies of the code simultaneously, relying on the operating system to place each copy on a separate processor core.

Tables 1 and 2 show in Column 3 the experimental running time required to generate 2-box and 3-box puzzles respectively, averaged over 10 samples for each entry. These data are depicted pictorially in Figure 4 (top) with an exponential trendline. Tables 1 and 2 also show in Column 2 the average number of moves required to solve the generated puzzles using the autosolver in JSoko, set to "move optimal with best pushes". These data are depicted pictorially in Figure 5 (bottom).

Table 3 shows in Column 3 the average experimental running time for 2×2 puzzles (which have 36 cells). See the level at top right of Figure 3 to get some idea of the size of the puzzle. These data are depicted pictorially in Figure 5 (top) with an exponential trendline. Table 3 also shows in Column 2 the average number of moves required to solve the generated puzzles using the autosolver in JSoko, set to "move optimal with best pushes". These data are depicted pictorially in Figure 4 (bottom).

Levels with a single box are generally uninteresting. Levels with 2 boxes can be generated very quickly, usually within a few seconds, but tend to be very easy. At 3 boxes the levels start to get slightly more interesting, and can still be generated within a few minutes. Levels with 4 or more boxes can

Size	**Moves**	**Time**
1×2	26	$<$ 1 sec
2×2	48	1.9 sec
2×3	60	16 sec
3×3	73	128 sec

Table 1: Average runtime for the generation of 2-box puzzles with the corresponding average number of moves needed to solve them (averaged over 10 random samples each).

Size	**Moves**	**Time**
1×2	38	58 sec
2×2	69	2.7 min
2×3	98	1.1 hr
3×3	115	24.5 hr

Table 2: Average runtime for the generation of 3-box puzzles with the corresponding average number of moves needed to solve them (averaged over 10 random samples each).

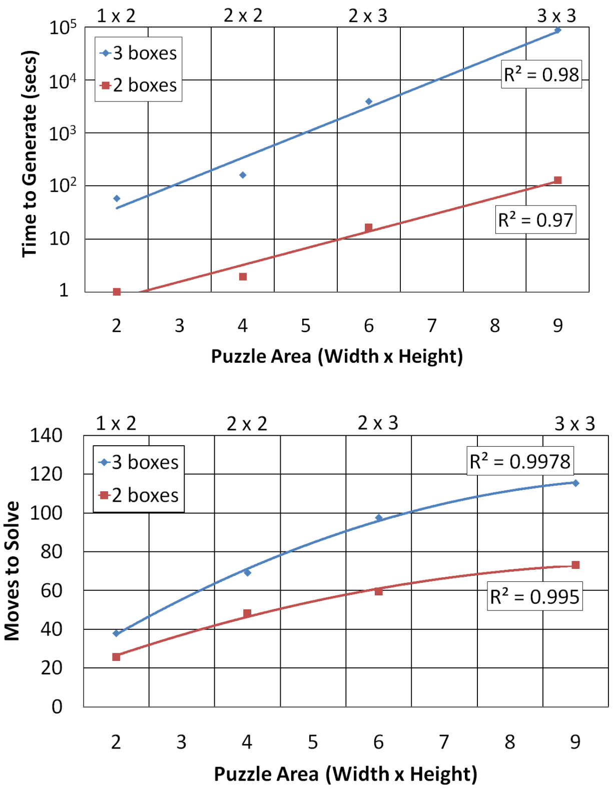
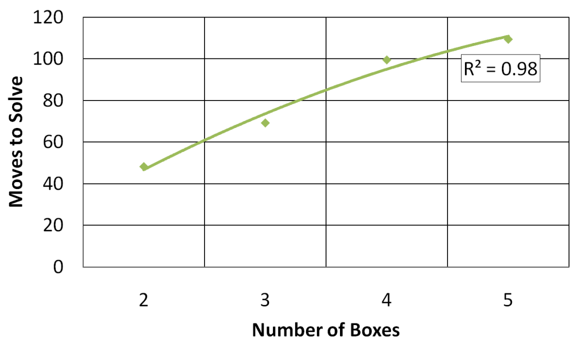


Figure 4: The average generation time in seconds for puzzles with 2 and 3 boxes versus puzzle area (top) and the number of moves needed to solve them (bottom).

Boxes	Moves	Time
2	48	1.9 sec
3	69	2.7 min
4	100	3.4 hr
5	109	26 hr

Table 3: The average experimental running time for the generation of 2×2 puzzles, with the corresponding average number of moves needed to solve them. The averages were computed over 10 random samples each.



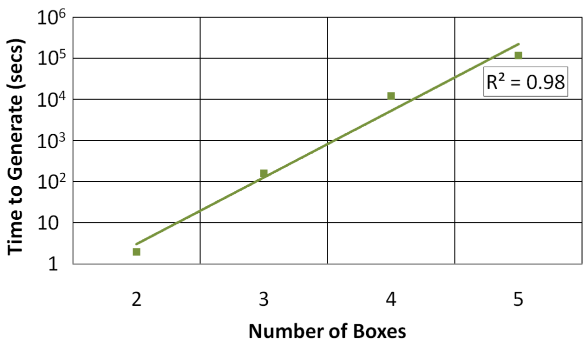


Figure 5: The average number of moves (top) and the average generation time in seconds (bottom) versus number of boxes for the generation of 2×2 puzzles.

be very interesting, and difficult, but take substantially more time to generate. Using the timer feature, we can force the generator to return the best result after a given time period. Using a time limit of 4 hours, we have generated levels with 5 and 6 boxes that appear interesting and difficult (for example Figure 6 includes some 5-box puzzles).

Using just iterative deepening, the algorithm runs several times faster, but uses much more memory. On some levels, the program crashed after consuming over 1.5GB of memory. Using our algorithm, the same levels never exceeded 40MB of memory.

CONCLUSION AND FUTURE WORK

While we found the puzzles that we generated "interesting", we provide no justification for this claim in this paper, although we do invite the reader to try for themselves by visiting our Sokoban Generator webpage (Taylor and Parberry 2011). We plan to gather data from play-testing in the next phase of this research, and we will report the results in a later paper. Some research into what makes a level interesting and what makes it difficult is needed, though some research on these questions has already been done (Ashlock and Schonfeld 2010, Jarušek and Pelánek 2010b).

REFERENCES

Ashlock D. and Schonfeld J., 2010. *Evolution for automatic assessment of the difficulty of Sokoban boards.* In *Proc. IEEE Congress on Evolutionary Computation*. 1–8.

Botea A.; Müller M.; and Schaeffer J., 2003. *Using Abstraction for Planning in Sokoban.* In *Computers and Games, Springer Lecture Notes in Computer Science*, vol. 2883. 360–375.

Culberson J., 1998. *Sokoban is PSPACE-complete.* In *Proc. International Conference on Fun with Algorithms*. 65–76.

Damgaard B.; Marxen H.; and Meger M., 2010. *JSoko.* URL `http://sourceforge.net/projects/jsokoapplet/`.

Doran J. and Parberry I., 2010. *Controlled Procedural Terrain Generation Using Software Agents. IEEE Transactions on Computational Intelligence and AI in Games*, 2, no. 2, 111–119.

Jarušek P. and Pelánek R., 2010a. *Difficulty Rating of Sokoban Puzzle.* In *Proc. Fifth Starting AI Researchers' Symposium*.

Jarušek P. and Pelánek R., 2010b. *Human Problem Solving: Sokoban Case Study.* Tech. Rep. FIMU–RS–2010–01, Faculty of Informatics, Masaryk Univ. Brno.

Junghanns A. and Schaeffer J., 1997. *Sokoban: A Challenging Single-Agent Search Problem.* In *Proc. IJCAI Workshop on Using Games as an Experimental Testbed for AI Research*. 27–36.

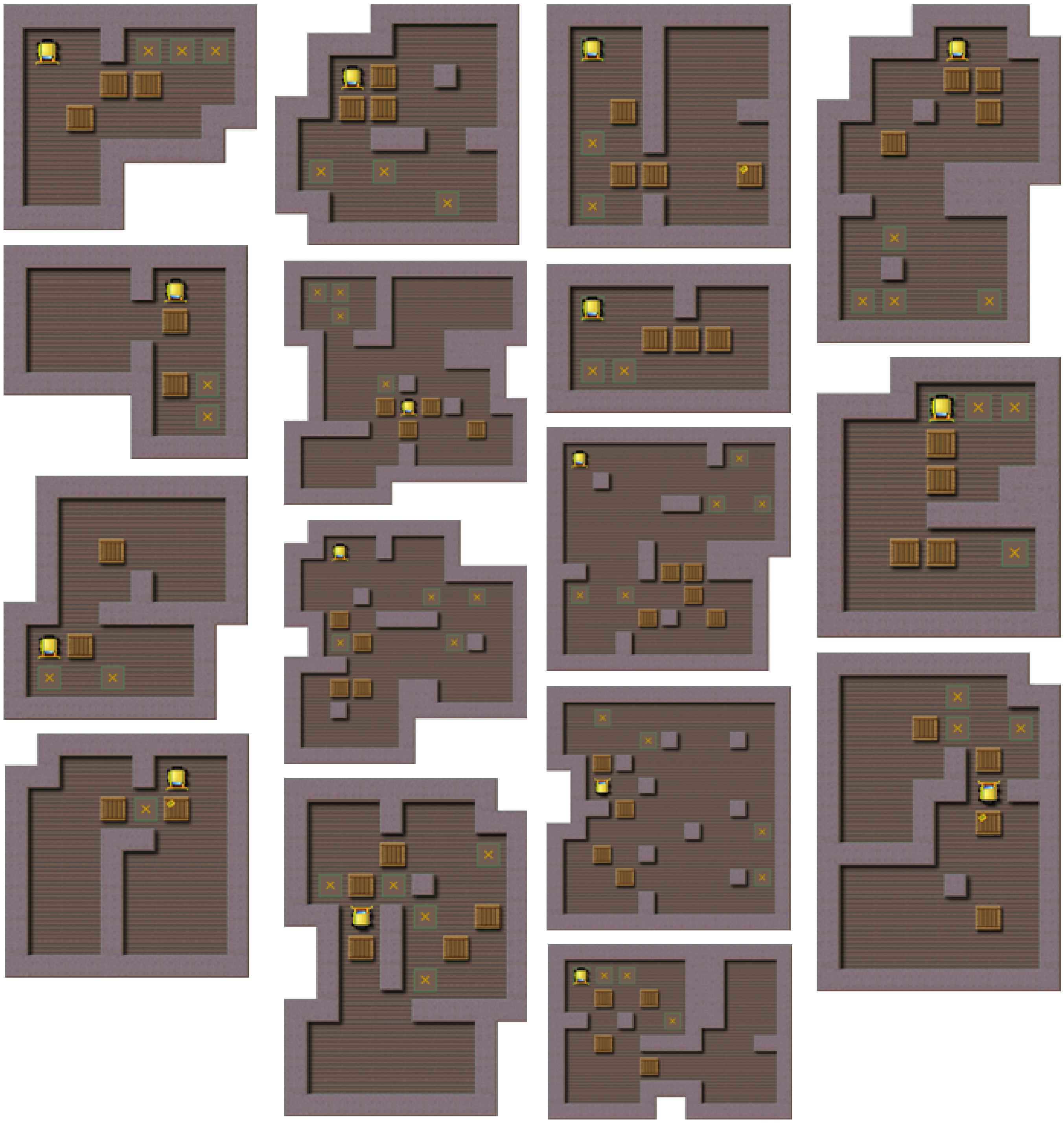


Figure 6: Some levels of varying difficulty created by our generator.

Masaru O.; Tomoyumi K.; and Satoru K., 2003. *A Method of Automatic Creation of Goal-Area in Sokoban Maps*. *Joho Shori Gakkai Shinpojiumu Ronbunshu*, 67–74.

Mühendisi M., Accessed 2011. *Sokoban Level Generators (A to Z)*. URL `http://www.erimsever.com/sokoban7.htm`.

Murase Y.; Matsubara H.; and Hiraga Y., 1996. *Automatic making of Sokoban problems*. In *PRICAI'96: Topics in Artificial Intelligence*, *Springer Lecture Notes in Computer Science*, vol. 1114. 592–600.

Servais F., 2005. *Finding Hard Initial Configurations of Rush Hour with Binary Decision Diagrams*. M.Sc. Thesis, Univ. libre de Bruxelles, Faculté des Sciences.

Takes F., 2007. *Sokoban: Reversed Solving*. Bachelor's Thesis, Leiden University.

Taylor J. and Parberry I., 2011. *Sokoban Generator*. URL `http://larc.unt.edu/ian/research/sokoban/`.

BIOGRAPHY

JOSHUA TAYLOR is a PhD student in the Department of Computer Science and Engineering at the University of North Texas. His research interests include procedural content generation.

IAN PARBERRY was born in London, England and emigrated as a child with his parents to Brisbane, Australia. After obtaining his undergraduate degree there from the University of Queensland he returned to England for a PhD from the University of Warwick. He has worked in academia in the US ever since. He is currently a full Professor in the Department of Computer Science and Engineering at the University of North Texas where he recently stepped down from a 2-year term as Interim Department Chair. A pioneer of the academic study of game development since 1993, his undergraduate game development program was ranked in the top 50 out of 500 in North America by The Princeton Review in 2010. He is on the Editorial Boards of the *Journal of Game Design and Development Education*, *IEEE Transactions on Computational Intelligence and AI in Games*, and *Entertainment Computing*, and he serves as the Secretary of the Society for the Advancement of the Science of Digital Games, which organizes the Annual Foundations of Digital Games conference. He is the author of 6 books and over 80 articles over 30 years' experience in academic research and education. His *h*-index is 18 and his Erdös number is 3. He can be contacted at `ian@unt.edu` or on Facebook. His home page is `http://larc.unt.edu/ian`.