AI Assisted Design of Sokoban Puzzles using Automated Planning

Tomáš Balyo[1] and Nils Froleyks[2]

[1] CAS Software AG, Karlsruhe, Germany
tomas.balyo@cas.de
[2] Johannes Kepler University, Linz, Austria
nils.froleyks@jku.at

Abstract. Designing interesting and challenging levels for a puzzle game is a very difficult and time consuming task. It is often possible to develop random puzzle generators that can produce solvable levels. However, in order to obtain appealing levels, usually a human designer needs to be involved. In this paper we propose a new generic method for assisting human designers to create solvable levels for a puzzle game by using Automated Planning. We will demonstrate our method on the well-known Japanese puzzle game Sokoban.

Keywords: Sokoban · Automated Planning · Puzzle Generation

1 Introduction

Sokoban is a puzzle game that originated in Japan. It was invented by Hiroyuki Imabayashi, and published in 1982 by Thinking Rabbit [11]. The word Sokoban is Japanese for warehouse keeper. Each puzzle represents a warehouse, where boxes are randomly placed. A warehouse keeper (the player) has to push the boxes around the warehouse so that all boxes end up in designated goal positions.

The game of Sokoban is a complicated computational problem. It was first proven to be NP-hard [6] and later PSPACE-complete [3]. While the rules are simple, even small levels can require a lot of computation to be solved. Designing interesting solvable levels is also challenging and a subject of academic research [15,18,14,5].

Automated planning [8] is one of the central techniques in artificial intelligence. The task of planning is to find a sequence of actions, i.e., a plan, that transforms the world from a given initial state to a goal state, i.e., a state that satisfies the given goal conditions. Planning is a very competitive research area and there exist multiple high performance planning tools that are constantly being developed and improved [16].

In this paper we will demonstrate how the power of automated planning tools can be utilized to design a system that can generate challenging solvable puzzles and intelligently assist a human puzzle designer. To our best knowledge, this is the first time, that automated planning has been used in this context.

We will demonstrate our technique on the example of Sokoban puzzles, since the game is widely known and well studied. Nevertheless, the technique can be used for any puzzle game that satisfies the following conditions:

– *Single player.* The game is played by a single player. There may exist helpful or adversary agents in the game as long as their behavior is fully deterministic and specified by simple rules.
– *Finite and discrete game world.* Each game state can be fully described with finitely many finite domain variables.
– *Deterministic gameplay.* Random events or random outcomes of player actions are not allowed.
– *Full observability.* There are no hidden or unknown elements that influence the gameplay.

The rest of the paper is organized as follows. In the next section we will provide the preliminary definitions of automated planning and the rules of Sokoban. Then we will review the related work in the area of procedural generation of Sokoban levels. Following that we will describe our new method and our new tool that implements it. Finally, we will present an evaluation of our tool.

2 Preliminaries

2.1 Automated Planning

As we already briefly stated in the introduction, planning is the task of finding a plan (a sequence of actions) that transforms the world from a given initial state to a goal state that satisfies the goal conditions. How to represent the world states, goal conditions, and describe the set of possible actions is defined in this Subsection.

Planning problems are modeled using the Planning Domain Definition Language (PDDL) [9], which is based on the programming language LISP [21]. PDDL is a very rich language with many features, however, we will only require a small subset of it which we will describe below.

The basic building blocks of PDDL are *Objects* and *Types*. Each object is of a certain type. For example if we define a type "city" then we can define the objects "Paris", "London", and "Madrid" of the type "city". Another type could be "person" and the objects of this type are for example "Alice" and "John". In PDDL we would express this using the following lines:

```
(:types city person - object)
(:objects
  Paris London Madrid - city
  Alice John - person
)
```

In PDDL we can refer to objects using variables. Variable names always start with a question mark "?" and each variable has a type. For example a variable "c" of the type "city" would be declared as: `(?c - city)`.

Variables appear in *Predicates*, which are atomic statements that are used to express certain conditions. For example, a predicate called ”livesIn” could have two parameters, one of the type ”person” and one of the type ”city”. In PDDL we would declare this predicate as `(livesIn ?p - person ?c - city)` and it would mean that an object of type ”person” lives in an object of type ”city”. Using the predicate we can now declare facts about our objects by substituting variables with objects of the proper type, for example:
`(livesIn Alice Madrid), (livesIn John London)`.

The last building block of PDDL that we need are operators, which can be intuitively understood as templates for actions. Actions change the world state by modifying the truth values of predicates. An action a consists of a name $name(a)$, a set of preconditions $pre(a)$ and a set of effects $eff(a)$. Both preconditions and effects are sets of grounded predicates (predicates where all variables are substituted by objects).

1. Preconditions represent the predicates that must be true in the given world state in order to execute the action. We say that an action a is applicable in a given world state s if and only if all predicates in $pre(a)$ hold true in s.
2. Effects are used to update the world state after the action is executed. Positive effects are predicates that will become true (unless they are already true) after the action is executed. Negative effects are negated predicates (wrapped in *not*) and they become false. All other predicates that are not involved in the effects of the executed actions remain unchanged.

The following is an example of an action representing moving Alice form Madrid to Paris:

```
(:action move-Alice-Madrid-Paris
  :precondition (and
    (livesIn Alice Madrid)
  )
  :effect (and
    (not (livesIn Alice Madrid))
    (livesIn Alice Paris)
  )
)
```

The precondition is that Alice lives in Madrid and the effects are that Alice does not live in Madrid anymore and she lives in Paris. If we wish to model all possible movements for both Alice and John and the three cities, we would need to write down 12 actions that are very similar to each other. A better solution is to use the already mentioned operators, i.e., action templates. Operators look like actions with the difference that they may have parameters and use predicates with variables in the preconditions and effects. An operator for the move actions would be declared as follows:

```
(:action move
  :parameters(?p - person ?from ?to - city)
  :precondition (and
    (livesIn ?p ?from)
  )
  :effect (and
    (not (livesIn ?p ?from))
    (livesIn ?p ?to)
  )
)
```

A planner would then generate all the possible actions from this template by substituting all the possible combinations of objects for the three parameters. This process is referred to as grounding.

Now we have everything we need to fully describe a planning problem in PDDL, which consists of the following elements:

1. set of used types
2. set of predicates
3. set of operators
4. list of all the objects in the problem together with their types
5. the initial state of the world in the form of grounded predicates (predicates with objects substituted for all variables)
6. the goal conditions in the form of grounded predicates

When describing a planning problem in PDDL we split the description into two files: domain.pddl and problem.pddl. The first file, domain.pddl, contains the types, predicates, and operators. The rest is written in the problem.pddl file. For our moving example the domain.pddl would be:

```
(define (domain moving)
  (:requirements :strips :typing)
  (:types city person - object)
  (:predicates
    (livesIn ?p - person ?c - city)
  )
  (:action move
    :parameters(?p - person ?from ?to - city)
    :precondition (and
      (livesIn ?p ?from)
    )
    :effect (and
      (not (livesIn ?p ?from))
      (livesIn ?p ?to)
    )
  )
)
```

and the problem.pddl would contain:

```
(define (problem moving-1)
  (:domain moving)
  (:requirements :strips :typing)
  (:objects
    Paris London Madrid - city
    Alice John - person
  )
  (:init
    (livesIn Alice Madrid)
    (livesIn John London)
  )
  (:goal (and
    (livesIn Alice Paris)
    (livesIn John Paris)
  ))
)
```

The domain file describes the general planning problem of moving people between cities, while the problem file describes the concrete problem instance of moving John and Alice from London and Madrid to Paris. An automated planner would now take these two files and find a plan, which in this case would consist of two actions: `move-alice-madrid-paris` and `move-john-london-paris`.

Since automated planning is a very competitive research field, it is easy to find well performing planning tools that are freely available on the internet. One way to choose a good planner is to look at the International Planning Competition website [16], where state-of-the-art planners are evaluated and compared in regular time intervals.

2.2 Sokoban

Each Sokoban level consists of a two dimensional rectangular grid of squares (see Figure 2 for an example). If a square contains nothing it is called a floor. Otherwise it is occupied by one of the following entities (see Figure 1):

- *Wall.* Walls make up the basic outline of each level. They cannot be moved and nothing else can be on a square occupied by a wall. A legal level is always surrounded by walls.
- *Box.* A box can either occupy a goal or an otherwise empty square. It can be moved in the four cardinal directions by *pushing* (see below).
- *Goal.* Goals are treated like floors for the most part. Only when each goal is occupied by a box the game is completed. In a legal level the number of goals matches the number of boxes. For the sake of simplicity, we will call a square that is either a goal or a floor square *free* since the worker and boxes can enter both.
- *Worker.* There must be exactly one worker in each level. It is the only element that is directly controlled by the player.

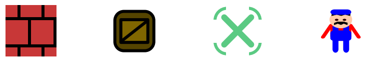


Fig. 1: The four kinds of tiles that make up a Sokoban warehouse: Wall, Box, Goal, and Worker (from left to right).

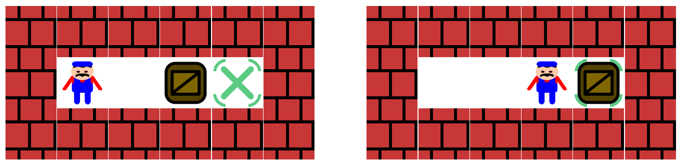


Fig. 2: A simple Sokoban level in its initial (left) and solved (right) state. The solution to this level consists of two steps: MOVE-RIGHT and PUSH-RIGHT.

There are two kind of moves in Sokoban:

1. *Move the worker.* The worker can be moved in the four cardinal directions (up, down, left, right) by one square in each step. This movement is directly controlled by the player. The worker may be moved onto an adjacent free square.
2. *Push a box.* The worker can push a box in a certain direction if the square behind the box is free. To be precise, there are always three squares (A,B,C) involved in a push move. The first (A) contains the worker, the second (B) contains a box and the third one (C) is a free (empty or goal) square. These three squares must form a single line of adjacent squares. After the push is performed, the box occupies the free square (C) and the worker occupies the square formerly occupied by the box (B).

The goal of the game is to find a *solution*, which is a sequence of moves and pushes. Executing a solution leads to every box ending up on a goal. It does not matter which box ends up on which goal. A level may have no solution. Such a level is undesirable and should not be presented to a human player for obvious reasons.

3 Related Work

Most academic work on Sokoban focused on developing efficient solvers that find short, but not necessarily optimal solutions. Most are based on performing a heuristic search on the state space of the Sokoban puzzle. To make the search efficient multiple domain specific enhancements are used. Most notably, heuristics that recognize if the current state is already unsolvable and abort the branch of the search accordingly. The first solver to implement these techniques was *Rolling Stone* [13] followed by *JSoko*[3], *YASS*[4], *Takaken*[5] and *GroupEffort*[7].

[3] https://www.sokoban-online.de/
[4] https://sourceforge.net/projects/sokobanyasc/
[5] http://www.ic-net.or.jp/home/takaken/e/soko/index.html

Botea et al. [2] used automatic planning but instead of a simple encoding to planning (see Section 4.1) they decomposed the warehouse into a set of different rooms connected by tunnels. A plan that successfully moves the boxes between the rooms is translated to actual box pushes and player movements afterwards.

Another topic related to our work was investigated more recently. Assessing the difficulty of a given level is important for designing new ones. Humans enjoy problem solving, but only if the problem is of adequate difficulty. Jarušek et al. [12] conducted an empirical study on how easily humans solve a set of Sokoban levels. They collected over 700 hours of test data from different participants to establish a ground truth and presented a set of nontrivial metrics trying to predict the collected data. Ashlock et al. [1] observed artificial agents that were the result of an evolutionary learning process on randomly generated Sokoban levels. Due to the limited capabilities of the agents the metrics they present are not useful to predict the difficulty of harder levels. Van Kreveld et al. [20] developed a metric that is not specific to Sokoban but supposed to be generic enough to capture the difficulty of different grid-based puzzle games.

The first published Sokoban level generator algorithm is by Murase et al. [15]. Their approach has three phases.

1. *Generate random levels.* In this phase predefined templates of rooms are placed randomly over a prototype level consisting of only walls. The templates are placed such that they are connected by passages. Then boxes and goal tiles are placed randomly.
2. *Filter out unsolvable levels.* Phase one may generate levels that have no solution. According to the authors this happens in around half of the cases. In this phase they use a Sokoban solver to try to find a solution and filter out unsolvable levels.
3. *Evaluation.* In this phase the levels are automatically evaluated to determine whether they are interesting. The evaluation is based on simple metrics such as the length of the solution, the number of changes in directions when pushing a box and the number of detours.

The complexity of this approach is dominated by phase two – filtering out unsolvable levels. This step requires solving Sokoban problems, which is a PSPACE-complete problem [4].

The approach of Taylor and Parberry [18] is similar to Murase et al. in that they first generate a random level based on placing templates of walls. Then they randomly place goals with boxes on them in the rooms. At this point they actually have a solved Sokoban puzzle. In the following stage they “unsolve” the level by doing reverse Sokoban moves, i.e., pulling boxes away from the goals. The aim of this stage is to reach a state that is far as possible from the solved state. They do this by running an iterative deepening search of the state space.

The complex part of this algorithm is the search for the starting state in the second stage. The process is very memory intensive, since all the visited states have to be kept in memory in order to avoid looping. On the other hand, the algorithm has the anytime property, i.e., it can be stopped at any time to return a valid solution, however, letting it run longer will yield a better solution.

In [19] an auditory Stroop test was performed to compare the engagement of players while playing hand-crafted Sokoban levels against levels generated by the approach of Taylor and Parberry [18]. The experiment showed that players found procedurally generated levels equally interesting to hand-crafted levels. This demonstrates that there is entertainment value in procedurally generated puzzles.

Kartal et al. [14] propose a Monte Carlo tree search (MCTS) based Sokoban level generator. They formulate puzzle generation as an MCTS optimization problem such that the puzzles are generated through simulated gameplay. The search process starts with a level full of walls except for one tile, which contains the player in its start position. The following actions are possible at each node of the search tree:

1. *Remove a Wall.* Choose a wall that is adjacent to an empty tile and remove it. By only removing walls adjacent to empty tiles they can ensure that no unreachable rooms are generated.
2. *Place a Box.* Choose an empty tile and put a box there.
3. *Freeze the Level.* With this action the search is changed to play mode. Removing walls and placing boxes is not allowed after this action. The current positions of walls, boxes and the player constitute the starting state of the level (without any goal positions, they will be defined later).
4. *Move the Player.* Simulate play by executing random legal moves of the player, i.e., walking around and pushing boxes.
5. *Evaluate the Level.* This is the final action of each search path. The current positions of the boxes are declared to be the goal locations and the quality of the generated level is estimated based on data driven evaluation functions.

Similarly to the previously presented method, this generator also has the anytime property. It is capable of producing a wide variety of levels thanks to its stochastic nature. Nevertheless, like all the presented approaches, it has its limitations and the generation of large puzzles remains a bottleneck as the number of possible level designs grows exponentially.

An up-to-date survey on procedural puzzle generation [5] gives an overview of the methods for generating puzzles for many games similar to Sokoban.

4 Puzzle Generation as Planning

Our proposed approach is based on the idea of using automated planners to generate solvable Sokoban levels. This means that our only task is to express the problem of level generation in PDDL and the rest is taken care of by the planning tool. We will formulate the problem of Sokoban level generation as an extension of Sokoban level solving.

4.1 Sokoban Solving as Planning

Using automated planning to solve Sokoban is not a new idea by any means. Actually, Sokoban is one of the standard benchmark problems used to evaluate new planning algorithms and tools in many academic papers and the international planning competition [16]. Nevertheless, in order to keep this paper self-contained, we will present a simple PDDL encoding of Sokoban in this Subsection.

To encode Sokoban solving we will only require one kind of objects – squares. We will have one object of type square for each location in the level that is not a wall. Additionally, we will need the following predicates:

1. `(above ?a ?b - square)` meaning that square "a" is above square "b"
2. `(left_of ?a ?b - square)` meaning that square "a" is on the left side of square "b"
3. `(box_at ?a - square)` meaning that there is a box at square "a"
4. `(worker_at ?a - square)` meaning that there is the worker at square "a"

To complete the domain description we need to specify the operators. We will need two kinds of operators – move and push and we will need 4 of each for the 4 cardinal directions (up, down, left, right). Fist, we will describe the move-up operator:

```
(:action move_up
  :parameters (?from ?to - square)
  :precondition(and
    (above ?to ?from)
    (worker_at ?from)
    (not (box_at ?to))
  )
  :effect (and
    (not (worker_at ?from))
    (worker_at ?to)
  )
)
```

The operators for moving down, left, and right are analogous, they only differ on the line with `(above ?to ?from)` where move-down has `(above ?from ?to)`, move-left has `(left_of ?to ?from)` and move-right has `(left_of ?from ?to)`. Next we describe the push-up operator:

```
(:action push_up
  :parameters (?from ?to ?box_to - square)
  :precondition(and
    (above ?to ?from)
    (above ?box_to ?to)
    (worker_at ?from)
    (box_at ?to)
    (not (box_at ?box_to))
  )
  :effect (and
    (not (worker_at ?from))
    (worker_at ?to)
    (not (box_at ?to))
    (box_at ?box_to)
  )
)
```

Like in the case of move operators, the other three push operators (push down, left, and right) only differ on the lines with the "above" predicates.

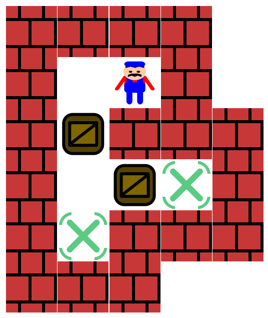
What remains is to specify the initial state and the goal conditions. For the initial state we need to declare the following predicates:

1. `(above a b)` for each pair of non-wall squares such that "a" is above "b".
2. `(left_of a b)` for each pair of non-wall squares such that "a" is on the left side of "b".
3. `(box_at a)` for each square "a" that contains a box.
4. `(worker_at a)` for the square "a" that contains the worker.

As for the goal conditions, we only need to specify that the goal squares must contain a box:

1. `(box_at a)` for each square "a" that contains a goal.

An example of a Sokoban level and its encoding in given on Figure 3.



```
(:objects
  s11 s12 s21 s31 s32 s33 s41 - square
)
(:init
  (above s11 s21) (above s21 s31)
  (above s31 s41) (left_of s11 s12)
  (left_of s31 s32) (left_of s32 s33)
  (box_at s21) (box_at s32) (worker_at s12)
)
(:goal (and
  (box_at s41) (box_at s33)
))
```

Fig. 3: A Sokoban level (left) and its encoding in PDDL (right)

4.2 Level Creation as Planning

As we already mentioned in the introduction, our generator is meant to assist a human designer and not just generate fully random levels (as is the case in the related work presented in Section 3). Therefore we allow the user to specify the size of the puzzle by defining the outer walls. The goal positions are also set by the designer. Additional walls, boxes and even the worker position may be defined as well. Lastly, the designer specifies which of the remaining free squares may contain a wall, a box, a worker, or some combination of the three. The last thing to define is the number of walls and boxes to be added to the puzzle. If no worker position has been specified in the input then the worker will be added automatically. We will refer to this input as a *level template*. Figure 4 contains an example of a level template, the corresponding starting puzzle and final puzzle.

```
p sokogen 1 2 0   // add 1 wall and 2 boxes, do at least 0 pushes
####              // symbols: # - wall              $ - box
#2.#              //          @ - worker            . - goal
#00###            //          + - worker on goal    * - box on goal
#.333#            // The numbers specify which tokens can be placed:
# 333#            //   0 - any      3 - box                6 - wall or box
#22###            //   1 - worker   4 - worker or wall
####              //   2 - wall     5 - worker or box
```

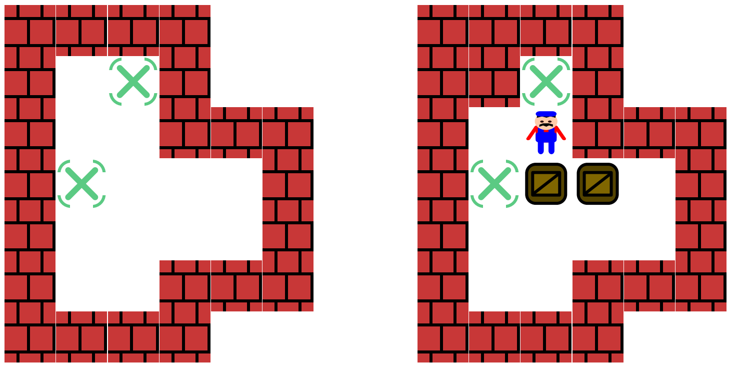


Fig. 4: A level template file for our Sokoban puzzle generator (top), the starting puzzle (down left) and the final solvable level (down right).

Transforming a level template into a solvable level will be task of the automated planner. In order to do this we must model the problem in PDDL. The PDDL model is an extension of the model used for solving Sokoban puzzles that we described in the previous Subsection. We will add four new operators:

1. *Add wall.* This operator adds a wall to one of the free squares that is allowed to contain a wall according to the level template.
2. *Add box.* Like the "add wall" operator, but for adding a box.
3. *Add worker.* Like the previous two but adds the worker.
4. *Start playing.* This operator means that we transition from the level creation phase to the playing phase of the planning problem. No more walls, boxes or workers can be added after this action is executed. Move and push actions are not allowed to happen before this action.

We also need to modify the goal conditions. For Sokoban solving we only required that all goal positions contain a box. Now we also require that the specified number of walls and boxes was placed. To model this we introduce two new types: wall and box. Then we declare as many objects of both types as we need to add according to the level template. For example, if we need to add 3 walls and 5 boxes, then 3 objects of type wall and 5 objects of type box are declared. Then with the help of two new predicates: `(wall_placed ?w - wall)` and `(box_placed ?b - box)` we can encode that all the additional walls and boxes have been placed.

In the initial state we must specify which squares may contain additional walls, boxes, or the player. For this purpose we introduce three new predicates: `(opt_wall ?s - square)` for walls, `(opt_box ?s - square)` for boxes, and `(opt_worker ?s - square)` for the worker.

To implement the *start playing* operator we will define two new predicates: `(making_level)` and `(playing)` to represent the current phase of the puzzle generation. The `(making_level)` is added to the initial state of problem definition, since we always start in this phase.

Now that we have defined all the new predicates we can model the four new operators in PDDL. We start with operators to place walls and boxes.

```
(:action place_wall
  :parameters(
    ?w - wall ?to - square
  )
  :precondition(and
    (making_level)
    (opt_wall ?to)
    (not (wall_placed ?w))
    (not (wall_at ?to))
    (not (box_at ?to))
  )
  :effect(and
    (wall_at ?to)
    (wall_placed ?w)
  )
)
```

```
(:action place_box
  :parameters(
    ?b - box ?to - square
  )
  :precondition(and
    (making_level)
    (opt_box ?to)
    (not (box_placed ?b))
    (not (wall_at ?to))
    (not (box_at ?to))
  )
  :effect(and
    (box_at ?to)
    (box_placed ?b)
  )
)
```

The ”place worker” and ”start playing” are defined next. Note, that ”place worker” also changes the phase to playing. This way we can ensure that the worker is added last and only once. Thanks to this property the operators to place the walls and the boxes do not need to check whether a worker has been placed on the square where they wish to place their item.

```
(:action place_player_and_start
  :parameters(?to - tile)
  :precondition (and
    (making_level)
    (opt_player ?to)
```

```
(:action start_play
  :parameters()
  :precondition(and
    (making_level)
  )
```

```
    (not (wall_at ?to))
    (not (box_at ?to))
  )
  :effect(and
    (player_at ?to)
    (not (making_level))
    (playing)
  )
)
```

```
  :effect(and
    (not (making_level))
    (playing)
  )
)
```

Lastly, the move and push operators from the Sokoban solving domain need to be slightly updated. The predicate `(playing)` must be added to preconditions. With this we have described a correct and complete encoding of the Sokoban puzzle generation into PDDL. However, there is one small issue we need to address.

Planners always try to find short plans. This has an unpleasant consequence for our problem. The planner is motivated to place the walls and boxes in such a way, that the generated puzzle can be solved with as few moves and pushes as possible. This means, that the generated levels tend to be very easy to solve. In order to address this issue, we modeled a mechanism, that enforces a certain minimum amount of pushes in the solve phase. This value can be specified by the puzzle designer as the third parameter on the ”p line” in the level template (see Figure 4). This is modeled by adding a counter to the push operators, that is increased with each push action. Then in the goal conditions we can require that the counter reaches the required value.

For more details refer to the complete domain PDDL file available in the project's repository[6]. The repository also contains the tool that generates the PDDL problem files from a given level template.

4.3 Comparison to related work

Our method bears the most similarity to the approach of Kartal et al. [14] (see Section 3). They formulate the puzzle generation as an MCTS optimization problem, while we model it as a planning problem. They start with a level that contains the worker and is otherwise full of walls and then remove some walls and add some boxes. We start with a partially built level that already contains all the goals and then we add additional walls, boxes and the worker. Then both approaches have a special action that transitions the search into the playing mode. Finally, in our approach we try to solve the level and backtrack to the level building phase if it is not solvable. In Kartal et al. [14] random moves are executed for some time and then the reached state is declared to be the goal state.

[6] https://github.com/biotomas/sokoplan/blob/master/SokoGen/domain.pddl

4.4 Generalization to other puzzles than Sokoban

It easy to use our puzzle generation concept for other puzzles as well. In order to do that one must only write the appropriate PDDL domain file and a generator for the PDLL problem files. The domain file is usually written by hand and problem files are generated by a script or a small program, however, both can be written by hand. In any case, the effort is very low in comparison to designing and implementing a dedicated puzzle generator for a specific puzzle as is done in related work. Another advantage of our concept is, that as the state-of-the-art planners evolve and their performance improves, our puzzle generator improves with it automatically.

5 Experimental Evaluation

Our Sokoban puzzle generation tool is available online at GitHub[7]. The repository contains everything you need to build and use our tool and also to replicate the experimental evaluation we present in this section.

5.1 Setup

As our tool is based on planning, we will obviously need a planner. Any planner that supports PDDL[8] would work, but based on some preliminary evaluations we settled on using the well established state-of-the-art planner FastDownward [10] with the LAMA 2011 [17] configuration.

We generated 300 level templates to use as benchmarks (more on how is described in the next Subsection) and we gave the planner a time limit of 1 minute to find a solvable puzzle. We run our experiments on a computer with an Intel(R) Core(TM) i7-7800X CPU @ 3.50GHz processor and 64 GB of main memory. The used operating system was Ubuntu version 5.8.0-26-generic.

5.2 Benchmark Instances

All of the 300 level templates are based on 10 base templates, which we designed by hand (see Figure 5). To create the benchmark templates of various complexities we reduced the number of goals in the base templates to a certain number. A benchmark template of complexity level x is defined as a template with x goal locations and the objective to add x walls and x boxes. We generated templates of complexity levels $1, 2, \ldots, 6$. We created 5 templates for each of the 6 complexity levels and each of the 10 base templates, hence the 300 total benchmark templates.

To create a template of complexity level x from a base template we first mark each floor square as a potential position for adding any (wall, box, or worker) object. Then we randomly remove goal squares until exactly x goal squares

[7] https://github.com/biotomas/sokoplan

[8] All available academic planners support PDDL

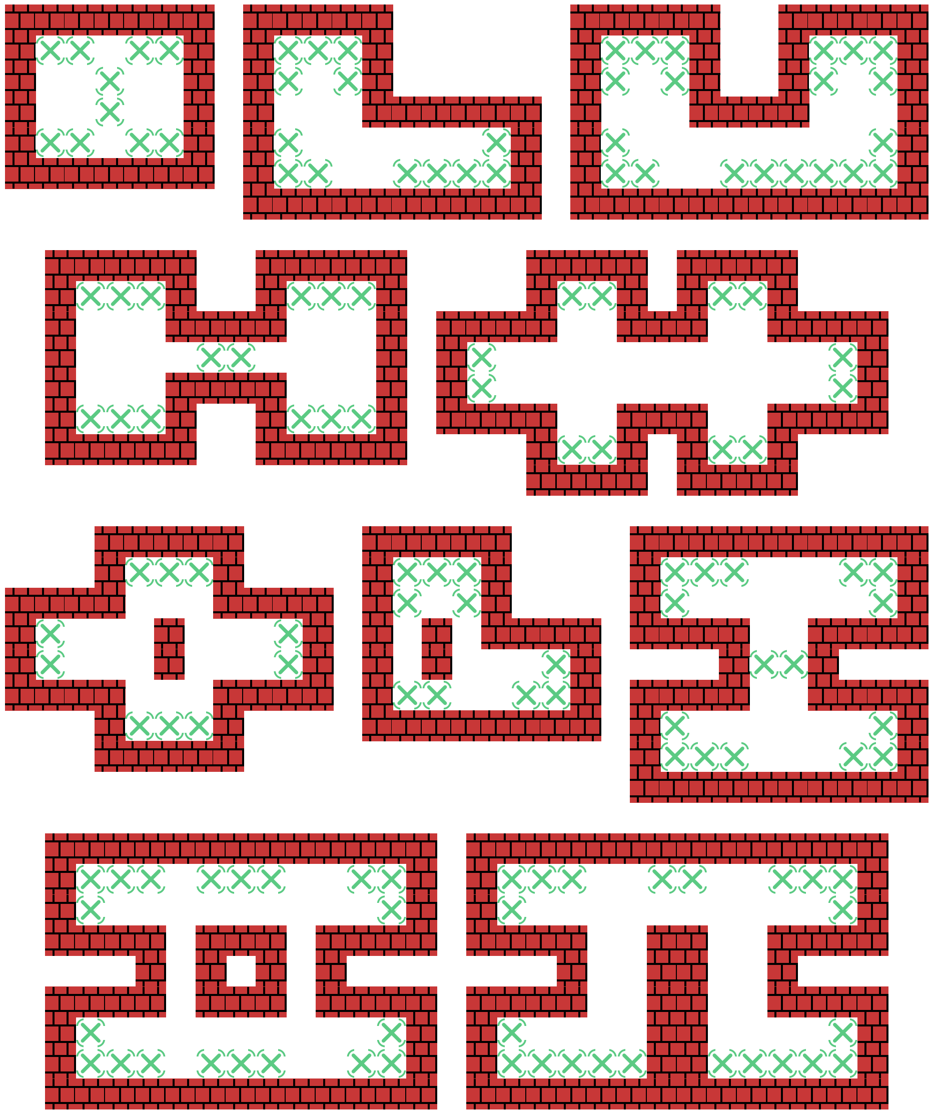


Fig. 5: The 10 base templates we used to generate our 300 benchmark level templates. The name of the base templates are (top left to bottom right): O, L, U, H, XX, X, B, I, II, and Pi.

remain. An example of generating a template of complexity level 4 from a base template follows:

```
base temp.    p sokogen 4 4 4  // objective:
#####         #####            //     add 4 walls and 4 boxes
#...#         #.  #            //     do 4 at least pushes
#. .#         # 0.#            //
# # ####  =>  #0#0####         // symbols:
# #   .#      #0#000 #         //     # - wall
#..  ..#      #. 00. #         //     . - goal
########      ########         //     0 - add any object
```

Admittedly, these benchmarks are not exactly like the level templates a human designed would use. A human designer would start with a level template and then modify it after seeing the level produced by the tool. They would perform several iterations of these steps until a satisfactory level has been found. Nevertheless, we used the approach described above since we needed to generate a large number of templates of various sizes and complexity levels. However, we believe the generated templates are still representative enough to perform a meaningful experimental evaluation of our tool.

5.3 Experimental Results

The results of the experimental evaluation are presented in Table 1. Not solving a level template can either mean that it is impossible to place the given amount of walls and boxes such that a solvable level is created or that the planner could not find a solution in the given time limit (of 1 minute). Unfortunately, in most cases, we cannot distinguish between these two scenarios, since planners are not very good at proving non-existence of plans.

For most of the base templates we could solve around 20 of the 30 level templates, except for X and B, which seem to be too tight to add more than 2 walls and 2 boxes in most of the cases. On the large base templates (XX, II, and Pi) we failed to solve most of the higher complexity templates. We believe that this is not due to the not existence of solutions, rather it is caused by the inability of the planner to find a solution within the given time limit. We could add 6 boxes and walls only for base templates U and I, which with 18 and 20 free squares represent middle sized levels. This seems to be the sweet spot between being to tight to place enough objects and too large to find a solution within the time limit.

Overall, the experimental evaluation showed that our approach works and we can rapidly generate levels of various shapes and complexities.

6 Conclusion

We presented a method to assist human level designers to generate solvable Sokoban puzzles using automated planners. Our method has several advantages.

Base	Free	Solved for Complexity Level						Total
Template	Squares	1	2	3	4	5	6	Solved
O	10	5	5	5	5	0	0	20
L	12	5	5	5	5	1	0	21
U	18	5	5	5	2	1	1	19
H	20	5	5	5	0	2	0	17
XX	30	5	5	5	1	1	0	17
X	18	3	3	1	1	0	0	8
B	9	5	5	0	0	0	0	10
I	20	5	5	4	3	2	1	20
II	28	5	5	3	4	0	0	17
Pi	35	5	5	5	1	0	0	16

Table 1: The table contains experimental results on our benchmarks grouped by base templates and complexity levels. The first column contains the names of the base templates, see Figure 5 for their definitions. The values in the second column are the number of free squares in the corresponding templates that can be used to place walls, boxes and the player. Columns 3 to 8 contain the number of solved instances within a time limit of 1 minute for each complexity level. The final column contains the total number of solved instances within 1 minute across all complexity levels.

Firstly, it based on a very generic principle (using planners) so it can be easily modified and used to generate puzzles other than Sokoban. Secondly, it is using a constantly evolving search technology (automated planning) so the generator will automatically improve with time as planners get more and more performant. Thirdly, it is very simple and easy to implement and customize.

6.1 Future Work

As for future work we would like to improve the performance of our tool by tuning the PDDL encoding and adjusting the configuration parameters of the used planner or evaluate other available planners.

We plan to develop a user friendly graphical user interface (GUI) for our generator to make it easy to use for less technical users.

Finally, we would like to test our general method on other puzzles than Sokoban. As described in the paper, this mostly only amounts to formulating new PDDL models for the given puzzles.

References

1. Ashlock, D., Schonfeld, J.: Evolution for automatic assessment of the difficulty of sokoban boards. In: IEEE Congress on Evolutionary Computation. pp. 1–8 (jul

2010). https://doi.org/10.1109/CEC.2010.5586239
2. Botea, A., Müller, M., Schaeffer, J.: Using abstraction for planning in sokoban. In: International Conference on Computers and Games. pp. 360–375. Springer (2002)
3. Culberson, J.: Sokoban is pspace-complete. In: Proceedings in Informatics. vol. 4, pp. 65–76. Citeseer (1997)
4. Culberson, J.: Sokoban is pspace-complete. Technical Reports (Computing Science) (1997)
5. De Kegel, B., Haahr, M.: Procedural puzzle generation: a survey. IEEE Transactions on Games **12**(1), 21–40 (2019)
6. Dor, D., Zwick, U.: Sokoban and other motion planning problems. Computational Geometry **13**(4), 215–228 (1996)
7. Froleyks, N., Balyo, T.: Using an Algorithm Portfolio to Solve Sokoban. In: Tenth Annual Symposium on Combinatorial Search (jun 2017)
8. Ghallab, M., Nau, D., Traverso, P.: Automated Planning and Acting. Cambridge University Press (2016)
9. Haslum, P., Lipovetzky, N., Magazzeni, D., Muise, C.: An Introduction to the Planning Domain Definition Language. Synthesis Lectures on Artificial Intelligence and Machine Learning **13**(2), 1–187 (apr 2019). https://doi.org/10.2200/S00900ED2V01Y201902AIM042
10. Helmert, M.: The fast downward planning system. Journal of Artificial Intelligence Research **26**, 191–246 (2006)
11. Imabayashi, H.: Sokoban Official. `https://sokoban.jp/title.html`
12. Jarušek, P., Pelánek, R.: Human problem solving: Sokoban case study. Technická zpráva, Fakulta informatiky, Masarykova univerzita, Brno (2010)
13. Junghanns, A., Schaeffer, J.: Sokoban: Evaluating standard single-agent search techniques in the presence of deadlock. In: Conference of the Canadian Society for Computational Studies of Intelligence. pp. 1–15. Springer (1998)
14. Kartal, B., Sohre, N., Guy, S.: Data driven sokoban puzzle generation with monte carlo tree search. In: Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment. vol. 12 (2016)
15. Murase, Y., Matsubara, H., Hiraga, Y.: Automatic making of sokoban problems. In: Pacific Rim International Conference on Artificial Intelligence. pp. 592–600. Springer (1996)
16. Pommerening, F., Torralba, A., Balyo, T., Vallati, M., Chrpa, L., McCluskey, L.: The international planning competition. `https://www.icaps-conference.org/competitions/` (1998–2018)
17. Richter, S., Westphal, M., Helmert, M.: Lama 2008 and 2011. In: International Planning Competition. pp. 117–124 (2011)
18. Taylor, J., Parberry, I.: Procedural generation of sokoban levels. In: Proceedings of the International North American Conference on Intelligent Games and Simulation. pp. 5–12 (2011)
19. Taylor, J., Parberry, I., Parsons, T.: Comparing player attention on procedurally generated vs. hand crafted sokoban levels with an auditory stroop test. In: Proceedings of the Foundations of Digital Games (2015)
20. van Kreveld, M., Löffler, M., Mutser, P.: Automated puzzle difficulty estimation. In: 2015 IEEE Conference on Computational Intelligence and Games (CIG). pp. 415–422 (aug 2015). https://doi.org/10.1109/CIG.2015.7317913
21. Winston, P.H., Horn, B.K.: LISP. Second Edition. Osti.gov, United States (jan 1986)