A Good Snowman is Hard to Plan

Miquel Bofill,[1] Cristina Borralleras, [2] Joan Espasa, [3] Gerard Martín,[1] Gustavo Patow,[1] Mateu Villaret[1]

[1] Departament d'Informàtica, Matemàtica Aplicada i Estadística, Universitat de Girona, Spain
[2] Departament d'Enginyeries, Universitat de Vic - Universitat Central de Catalunya, Spain
[3] School of Computer Science, University of St Andrews, UK
miquel.bofill@udg.edu, cristina.borralleras@uvic.cat, jea20@st-andrews.ac.uk, gerardmartinteixidor@gmail.com, gustavo.patow@udg.edu, mateu.villaret@udg.edu

Abstract

In this work we face a challenging puzzle video game: *A Good Snowman is Hard to Build*. The objective of the game is to build snowmen by moving and stacking snowballs on a discrete grid. For the sake of player engagement with the game, it is interesting to avoid that a player finds a much easier solution than the one the designer expected. Therefore, having tools that are able to certify the optimality of solutions is crucial.

Although the game can be stated as a planning problem and can be naturally modelled in PDDL, we show that a direct translation to SAT clearly outperforms off-the-shelf state-of-the-art planners. As we show, this is mainly due to the fact that reachability properties can be easily modelled in SAT, allowing for shorter plans, whereas using axioms to express a reachability derived predicate in PDDL does not result in any significant reduction of solving time with the considered planners. We deal with a set of 51 levels, both original and crafted, solving 43 and with 8 challenging instances still remaining to be solved.

Introduction

During the design of video games, one of the most important aspects to consider is the difficulty of each level. A common problem while designing the levels is the possibility of ignoring a possible solution to the level that is much easier than the ones the designer had initially in mind (Silli 2010). These unplanned solutions often are not desired because they break the difficulty slope, making the game uninteresting. To this end, it is of crucial interest to have a tool able to provide (certified) optimal solutions for given scenarios of the game.

In this work we focus on finding optimal solutions for a discrete time and space puzzle. In particular we consider the PSPACE-complete game *A Good Snowman is Hard to Build* (Davis and Hazelden 2015; He, Liu, and Yang 2017), where a character has to move snowballs to build snowmen. The game has some resemblance with the widely studied Sokoban (Culberson 1997) game, because the character pushes balls (instead of boxes) in a matrix-like maze with some obstacles. However, it generally gets more involved because not only scenario characteristics (positions of balls and snow) may change when moving snowballs, but the final positions of snowmen are not fixed, hence the goal is conceptually disjunctive, making the search space much bigger in comparison. Therefore, both the modelling and the increasing computational complexity makes the considered game challenging. In these kind of games, the difficulty of a level is sometimes measured using the number of times the character needs to move an object. There are two main reasons for this: first, moving the character from one location to another uses to be trivial because it consists on identifying if there is a path from one location to another; second, pushing an object may close free paths.

A Good Snowman is Hard to Build is naturally characterised as a planning problem, where we are asked to find a sequence of actions such as rolling balls on snow, and stacking or unstacking them, so that balls of the appropriate sizes are joined into snowmen. In the planning literature, most models for Sokoban have two actions: *move*, for changing the location of the character, and *push* for changing the location of a box by making the character push it. Some previous works devise methods to focus the search efforts on certain parts of the problem. In Ivankovic and Haslum (2015), the authors show how a reachability derived predicate can be specified with axioms and the *move* actions can be completely avoided in the model. The idea is to ensure with the reachability predicate that the pushing location is *reachable* from the current character location. This results in shorter plans and in a significant reduction of the search space and hence in the time required to solve the instances. Similarly, a proposal to automatise the inference of axioms and to reformulate the problem accordingly is given in Miura and Fukunaga (2017). This is done with success for Sokoban and reachability, as shown by using axiom supporting model-based planners with Integer programming and Answer Set Programming technologies.

In this paper, we start by providing a model for *A Good Snowman is Hard to Build* written in PDDL (Haslum et al. 2019), the de-facto standard language for AI planners. This model contains actions to both *move* the character and to *push* snowballs, and we evaluate the performance of some state-of-the-art planners on it. We also provide a PDDL model using axioms for the reachability predicate, hence, only considering *push* actions, which preconditions require that the location where the action is performed is reachable

from the location where the character was at the previous time step. Unfortunately, despite the detailed instructions for the planner from Ivankovic and Haslum (2015) we have not been able to install it, and the planner from Miura and Fukunaga (2017) was not available. Therefore, we use a state-of-the-art planner supporting derived predicates but without any specialized heuristics for them. To fully assess the overhead of the *move* actions, we also consider a "what if" scenario, where the character can do unsound teleports. That is, teleports that do not guarantee that there exists a valid path from source to destination (a sort of cheating). Interestingly, experiments show that this does not have a significant impact in solving time.

Next, we present an ad-hoc planning as SAT approach. Again, we then get rid of *move* actions by encoding reachability. This results in a significant reduction of the time horizon and, accordingly, of the solving times, allowing to solve many more instances to optimality. By using the proposed planning as SAT approach with reachability, we are able to certify the minimum number of ball movements needed to solve most of the original levels of *A Good Snowman is Hard to Build* and of the additional crafted instances that we also provide.

The main contribution of this paper is a new case study with a set of challenging PDDL instances (with and without derived predicates) and a detailed comparison of planning tools against planning as SAT methodology. For the planning as SAT approach we use the s-t-reachability encoding from Gebser, Janhunen, and Rintanen (2014); Rankooh and Rintanen (2021) and the folklore one based on neighbour counting. We adapt these encodings to deal with a priori unknown source and target locations in a dynamic environment.

For the sake of reproducibility, the presented PDDL models and instances and the SAT encoding generators can be found in the supplementary material at https://github.com/udg-lai/KEPS2023. In addition, the full detailed experimental results are also included.

The Game

A Good Snowman Is Hard To Build is a single-agent puzzle video game where the goal is to push snowballs in a maze to build some snowmen by stacking three balls of decreasing size.

The game elements are the agent (i.e., the black character controlled by the player), the playable cells, which may or not contain snow, and the snowballs, which are initially distributed on the playable cells. Snowballs have three possible sizes: small, medium and large. As in Sokoban, the only allowed action is moving the agent in one of four directions. The results of *moving* depend on the cells in front:

- *Move*: When the agent walks into a free cell, he simply moves to that cell.
- *Roll*: When the agent moves into a cell with a single ball, and there is a free cell in front of the ball, the ball gets pushed and the agent occupies the cell previously occupied by the ball. If a ball is pushed into a snow cell, the snow disappears and the ball increases in size, up to a maximum. Still, the snow is always removed.
- *Push*: A ball can be pushed on a stack of balls if the size of the ball in the top is bigger than the one being pushed. Then, the agent occupies the cell previously occupied by the pushed ball like when rolling.
- *Pop*: Trying to move into a stack of balls will pop the topmost ball but will not change the location of the agent. This action can only happen if the ball falls directly into a cell without any ball.

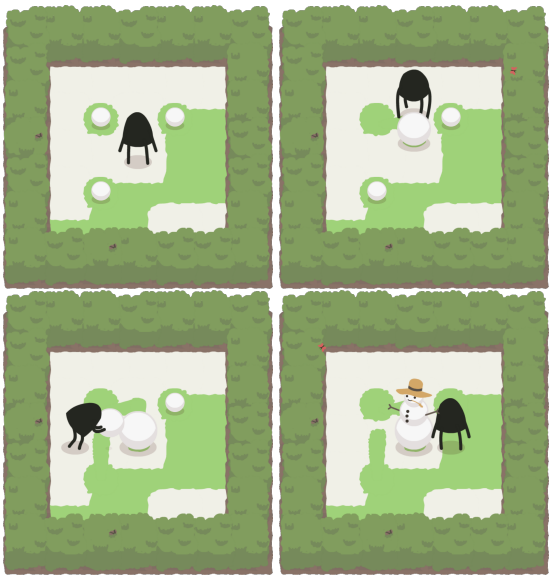


Figure 1: *Andy* level, showing the execution of the optimal solution *lluRurDllddrUluRuurrrdLulD*. Letters represent the direction of movement. Uppercase letters indicate ball movements.

Notice that, as in Sokoban, the agent is not allowed to pull balls.

The goal of the game is to build snowmen composed by a pile of three balls of decreasing size. The scenarios of the game considered consist of three, six or nine snowballs, hence, one, two or three snowmen. Snowmen can be built anywhere. Figure 1 depicts an example of how to solve one of the levels of the game.

PDDL Formulation

Since the problem is very similar to Sokoban, we consider a PDDL formulation based on the Sokoban domain used in the 2008 and 2011 International Planning Competitions. However, when encoding it in PDDL there are various key differences, as the usage of quantifiers and conditional effects is essential. This is due to elements such as: disappearing snow, ball stacking and the disjunctive goal. In this section we partially describe this PDDL formulation.[1]

[1]Full models and instances can be found at https://github.com/udg-lai/KEPS2023.

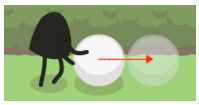


Figure 2: Location parameters of action `move_ball`.

Types, objects and fluents The objects considered have the following types: `loc`, `dir` and `ball`. The `loc` type represents a grid cell location, which may contain the character or a stack of balls. In addition to these, a cell may also contain snow. The `dir` type enumerates the four directions the character and balls can move (`up`, `down`, `left`, `right`) and the `ball` type defines all possible balls of the scenario, e.g., `ball1`, `ball2` and `ball3`. Note that when building two snowmen there will be six balls, whereas when building three snowmen there will be nine balls. In order to model the current state, the following predicates are defined:

- `(snow ?l - loc)`: location `l` is covered in snow.
- `(next ?from ?to - loc ?d - dir)`: locations `from` and `to` are are adjacent; location `to` is in direction `d` relative to `from`.
- `(occ ?l - loc)`: location `l` contains at least one ball.
- `(char_at ?l - loc)`: character is at location `l`.
- `(ball_at ?b - ball ?l - loc)`: ball `b` is at location `l`.
- `(ball_size_s ?b - ball)`: ball `b` has small size.
- `(ball_size_m ?b - ball)`: ball `b` has medium size.
- `(ball_size_l ?b - ball)`: ball `b` has large size.
- `(goal)`: the goal is satisfied.

Actions Three actions are defined: `move_character`, `move_ball` and `goal`. As its name indicates, action `move_character` moves the character from its current location to some of its next adjacent locations, according to a direction.

Action `move_ball` moves a single ball. This action has as parameters a ball to be moved (`b`) and three aligned and adjacent locations (`ppos`, `from` and `to`). Figure 2 illustrates the role of the three location parameters. With this action, ball `b` always is going to move, but the character may not.

To perform the action we first need to ensure that the given three locations are aligned and adjacent (line 5), ball `b` is located at the `from` location (line 6), and the character is located at the `ppos` location (line 7). As the game rules states, we can only move a ball if it does not have any other ball on top, i.e., if it is the smallest ball of the stack (lines 8–22). Also, if a ball is on top of a stack of balls, it cannot be moved to another stack of balls (lines 23–28). Finally, to maintain a correct game state, a ball can only be pushed to a stack of balls if it is smaller than all other balls in the stack (lines 29–33).

When the action is performed, since a ball is moved, it modifies the occupancy of some location (lines 35 and 37–43). Note that occupancy is conditional since the character

```
1  (:action move_ball
2   :parameters
3      (?b - ball ?ppos ?from ?to - loc ?d - dir)
4   :precondition (and
5      (next ?ppos ?from ?d) (next ?from ?to ?d)
6      (ball_at ?b ?from)
7      (char_at ?ppos)
8      (forall (?o - ball)
9          (or
10             (= ?o ?b) ; Ball != from ball action
11             (or ; Implication
12                 (not (ball_at ?o ?from))
13                 (or ; Ball ?b smaller than ?o
14                     (and
15                         (ball_size_small ?b)
16                         (ball_size_medium ?o))
17                     (and
18                         (ball_size_small ?b)
19                         (ball_size_large ?o))
20                     (and
21                         (ball_size_medium ?b)
22                         (ball_size_large ?o))))))
23     (or
24         (forall (?o - ball)
25             (or (= ?o ?b)
26                 (not (ball_at ?o ?from))))
27         (forall (?o - ball)
28             (not (ball_at ?o ?to))))
29     (forall (?o - ball)
30             (or ; Implication
31                 (not (ball_at ?o ?to))
32                 (or ... ; Ball ?b smaller than ?o
33                 ))
34   :effect (and
35      (occ ?to)
36      (not (ball_at ?b ?from)) (ball_at ?b ?to)
37      (when
38          (forall (?o - ball)
39              (or (= ?o ?b)
40                  (not (ball_at ?o ?from))))
41          (and (not (char_at ?ppos))
42               (char_at ?from)
43               (not (occ ?from))))
44      (not (snow ?to))
45      (when
46          (and (snow ?to)
47               (ball_size_s ?b))
48          (and (not (ball_size_s ?b))
49               (ball_size_m ?b)))
50      (when
51          (and (snow ?to)
52               (ball_size_m ?b))
53          (and (not (ball_size_m ?b))
54               (ball_size_l ?b)))
55      (increase (total-cost) 1)))
```

might be rolling, pushing or popping a ball. We also have to update the ball location (line 36). If the character rolls or pushes a ball, its location changes (lines 37–43). We remove the snow from the ball location unconditionally (line 44). If there is snow in the target location, we consider two cases: increasing ball size from small to medium (lines 45-49) or from medium to large (lines 50-54). At last, we declare that this action has cost 1 (line 55).

The `goal` action (in the single snowman scenarios) has three balls and a location as parameters. Via its precondition it enforces that the three balls are different objects located at the same place. Note that the ball sizes constraint is enforced in the possible state transitions of the `move_ball` action.

```
1  (:action goal
2   :parameters
3      (?b0 ?b1 ?b2 - ball ?p0 - loc)
4   :precondition (and
5      (not (= ?b0 ?b1)) (not (= ?b0 ?b2))
6      (not (= ?b1 ?b2))
7      (ball_at ?b0 ?p0) (ball_at ?b1 ?p0)
8      (ball_at ?b2 ?p0))
9   :effect (and  (goal)) )
```

As said in the introduction, in Sokoban-like games, sometimes the notion of optimal plans prioritizes box movements minimization. We propose to also consider this notion of optimality here, hence, correspondingly, only snow ball movements are considered. This is meaningful in Snowman because, for a human player, it is trivial to identify if there is a path from the current location of the character to the location of the next snow ball movement. Therefore, we have the statement `(:metric minimize (total-cost))` and only the action `move_ball` has a cost.

Cheating variant In order to evaluate how costly is having to find out all the movements of the character, we also evaluate a modified formulation, where the `move_character` action is removed and in the `move_ball` action it is assumed that the character can teleport (see the empirical evaluation section). Note that this relaxation of the formulation may lead to non-valid plans since sometimes balls will be blocking some paths.

Reachability variant PDDL offers logical axioms to specify derived predicates, a feature able to cleanly express reachability. Similarly as done in Ivankovic and Haslum (2015) with Sokoban, we declare a new predicate `reachable` with two parameters, the source and target location. Given a pair of `?s` (source) and `?t` (target) locations, the target will be `reachable` from the source if the target is not occupied and, it is either the same location as the source or there exists a middle location `?m` such that it is next to the source and it is able to reach the target.

We can now remove the action that moves the character. We need an additional parameter `?prevpos` in the `move_ball` action, to refer to the position where the character ended in previous state. We then require `(reachable ?prevpos ?ppos)` in `move_ball` action precondition, enforcing that the position where the ball push is going to be executed from is reachable from where

```
1  (:derived (reachable ?s ?t - location)
2     (and
3        (or ;; there is a path between ?s and ?t
4            (= ?s ?t) ;; either is the same location,
5            ;; or it is inductively reachable
6            (exists (?d - direction)
7               (exists (?m - location)
8                  (and
9                     (next ?s ?m ?d)
10                    (not (occupancy ?m))
11                    (reachable ?m ?t)))))
12       ;; and target location has no ball in it
13       (not (occupancy ?t))))
```

the character was. Finally, an additional effect will be required to update the character position, i.e., change the position of the character from `?prevpos` to `?ppos`.

Planning as SAT

Here we recall some basic ideas of Planning as SAT. In this approach (Kautz and Selman 1992), a planning problem is encoded to a Boolean formula, with the property that any model of this formula will correspond to a valid plan.

Since the length of a valid plan is not known a priori, the basic idea is to encode the existence of a plan of T steps with a formula $f(T)$. Then, the method for finding the shortest plan consists in iteratively checking the satisfiability of $f(T)$ for $T = 0, 1, 2, \ldots$ until a satisfiable formula is found.

Variables need to be replicated for each time step. E.g., a^t will denote if action a is executed at time t. Then, the general (standard) encoding goes as follows. First of all, it is stated that the execution of an action implies its preconditions, for each $t \in 0..T-1$:

$$a^t \rightarrow \mathit{Pre}^t \text{ for every action } a = \langle \mathit{Pre}, \mathit{Eff} \rangle$$

Also, if an action is executed, its effects must take place at the next time step:

$$a^t \rightarrow \mathit{Eff}^{t+1} \text{ for every action } a = \langle \mathit{Pre}, \mathit{Eff} \rangle$$

Preconditions and effects are typically given as sets of literals. Here, by Pre^t and Eff^{t+1} we denote the corresponding formulas (usually, conjunctions of literals) on the time-indexed state variables.

Moreover, a change in the value of every state variable v can occur only if an action that can change this value has been executed:

$$(v^t \neq v^{t+1}) \rightarrow \bigvee \{a^t \mid a = \langle \mathit{Pre}, \mathit{Eff} \rangle, v \in \mathit{Eff}\}$$

These are called frame axioms. Finally, it is stated that exactly one action is executed at each time step t, and that the goal holds at time T.

Snowman SAT encoding

In this section we propose an encoding for solving the problem at hand, following the planning as SAT approach.

For clarity and space limitations we do not provide the whole encoding, but the viewpoint (state variables) and an

excerpt of the formulas including the goal and relevant transition constraints and frame axioms. Also, for simplicity reasons, we only describe the case of a single snowman.

We represent states with Boolean variables stating, for each location, whether (i) there is snow or not, (ii) there is a ball of a particular size or not, and (iii) there is the character or not. The actions considered are only four, corresponding to a ball movement per possible direction. This will eventually result in rolling, pushing or popping a ball, depending on the current state. In other words, we only need to know the direction of the action.

We consider L to be the set of valid (non-wall) locations, S the set of locations with snow, and T the number of time steps considered. Below we detail the main variables and constraints.

Variables For all l in L, t in $0..T$:

- s_l^t: there is snow at location l at time t
- bs_l^t, bm_l^t, bl_l^t: there is a small, medium or large ball at location l at time t
- c_l^t: character is at location l at time t

For all t in $0..T-1$:

- n^t, s^t, e^t, w^t: direction of action at time t is north, south, east or west

Constraints The goal consists in requiring all balls to be at the same location at the end of the plan:

$$\forall l \in L \quad (bs_l^T \leftrightarrow bm_l^T \wedge bm_l^T \leftrightarrow bl_l^T)$$

For each time step t in $0..T-1$ we have the following constraints.

- Exactly one action is executed per time step: we impose an exactly-one constraint over n^t, s^t, e^t, w^t variables.
- Action preconditions and effects (excerpt of the action to move the character north):

  Let L_n be the set of valid locations with a wall at north and let L_{nn} be the set of valid locations with a wall two locations ahead at north.

  When the character is at any location in L_n, it cannot go north:

  $$\forall l \in L_n \quad c_l^t \rightarrow \neg n^t$$

  Otherwise, if the character moves north and it has a wall two locations ahead, there cannot be any ball in front of him, and at the next time step the location of the character has changed accordingly (here l_n denotes the location at north of l):

  $$\forall l \in L_{nn} \setminus L_n \quad c_l^t \wedge n^t \rightarrow (move : \neg c_l^{t+1} \wedge c_{l_n}^{t+1} \wedge \neg bs_{l_n}^t \wedge \neg bm_{l_n}^t \wedge \neg bl_{l_n}^t)$$

  Finally, if the character moves north without having a wall two locations ahead, apart from moving it can also *roll* a ball north, *push* a ball into a stack of balls or *pop* a ball from a stack of balls. For the sake of brevity we only describe the *push* north action (here l_{nn} denotes the location two steps ahead at north of l):

  $$\forall l \in L \setminus \{L_n \cup L_{nn}\} \quad c_l^t \wedge n^t \rightarrow (move : \ldots \vee push : (\neg c_l^{t+1} \wedge c_{l_n}^{t+1} \wedge ((\neg bs_{l_n}^{t+1} \wedge bs_{l_{nn}}^{t+1} \wedge bs_{l_n}^t \wedge \neg bm_{l_n}^t \wedge \neg bl_{l_n}^t \wedge \neg bs_{l_{nn}}^t \wedge (bm_{l_{nn}}^t \vee bl_{l_{nn}}^t)) \vee (\neg bm_{l_n}^{t+1} \wedge bm_{l_{nn}}^{t+1} \wedge \neg bs_{l_n}^t \wedge bm_{l_n}^t \wedge \neg bl_{l_n}^t \wedge \neg bs_{l_{nn}}^t \wedge \neg bm_{l_{nn}}^t \wedge bl_{l_{nn}}^t))) \vee roll : \cdots \vee pop : \ldots)$$

- Frame axioms impose that the state cannot change without a reason: snow cannot be created, or if it disappears from a location it must be because a ball occupies that location, etc.:

  $$\forall l \in L \quad (\neg s_l^t \rightarrow \neg s_l^{t+1}) \wedge (s_l^t \wedge \neg s_l^{t+1} \rightarrow bm_l^{t+1} \vee bl_l^{t+1}) \wedge \ldots$$

Optimal plans with respect to ball movements

As previously done with PDDL, now we also modify the basic encoding by considering only ball movement actions, asking in their preconditions that the location from where the action is performed is reachable from the last location of the character. Additionally, a cheating variant encoding is included in the experiments for performance comparison.

Standard Graph Reachability

We need to characterize reachability from the location of the character to the appropriate location next to the ball to be moved. Gebser, Janhunen, and Rintanen (2014) describe an SMT encoding for *source-target reachability* in graphs, where sources and targets are fixed beforehand. In this section we provide a similar encoding to SAT which is able to deal with unknown a priori sources and targets. We also incorporate the fact that accessible locations may change due to ball movements.

The basic idea of the encoding is to prove the existence of a *reachability path* from the source to the target location. We stress that the way of characterising reachability is intrinsically different in SAT or SMT than in PDDL. In particular, we cannot mimic the idea followed in PDDL, where a location t is reachable from s either if s equals t or there is some (unoccupied) neighbour s' of s such that t is reachable from s'. The reason is that, whereas planning adheres to the closed-world assumption (i.e., all unknown values are assumed to be false), this is not the case for SAT and SMT. In other words, the translation of the inductive reachability axioms of PDDL into SAT would be satisfied by any model where all corresponding reachability variables are set to true, hence not encoding reachability at all.

To deal with the open-world assumption of SAT (and SMT) when encoding reachability in graphs, we essentially need to break cyclic relations, i.e., paths from source to target must be encoded as a transitive and antisymmetric relation. As described in Gebser, Janhunen, and Rintanen (2014), acyclicity can be easily modelled with SMT, since

an ordering on locations can be imposed by assigning a numeric value to each of them. The details of the encoding are given below and the way we impose the ordering constraints in SAT is given in next subsection.

We consider a directed graph representation of the maze where the nodes correspond to the valid locations of the maze, and where there is a directed edge from l to l' if and only if l and l' are adjacent locations in the maze. In other words, we define a directed graph $G = (V, E)$ where $V = L$ and $E = \{(l, l'), (l', l) \mid l, l' \text{ are adjacent locations in } L\}$. The encoding goes as follows.

Variables We introduce for all l in L, t in $0..T-1$, the following Boolean variable:

- r_l^t: node l is reachable by the character at time t

For a node l to be reachable, either it must be the source or it must be reached through an edge ending in l and coming from a reachable node. Therefore, we need variables to state if an edge is building the reachability path or not. For all edges (l, l') in E and time t we introduce the following Boolean variable:

- $e_{l\,l'}^t$: edge (l, l') is in the reachability path at time t

Moreover, cycles in those paths must be avoided. Therefore the SMT encoding from Gebser, Janhunen, and Rintanen (2014) would also introduce, for each location l in L and time t the following integer variable:

- $a_l^t \in 1..|L|$: value associated to node l at time t; this is used to break cycles: a topological ordering will be enforced among variables a of the nodes in the reachability path to avoid cycles.

Constraints The reachability constraints are the following, where $next(l) = \{l' \mid (l, l') \in E\}$.

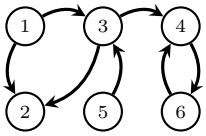
$$\forall_{l \in L} \quad r_l^t \rightarrow \big(c_l^t \bigvee_{l' : l \in next(l')} e_{l'\,l}^t\big)$$

$$\forall_{(l,l') \in E} \quad e_{l\,l'}^t \rightarrow (r_l^t \wedge a_l^t < a_{l'}^t)$$

Notice that this last ordering constraint is an SMT constraint because it is using the difference logic atom $a_l^t < a_{l'}^t$. We show in next subsection how to replace it in SAT.
If there is a ball in a location then it is not reachable:

$$\forall_{l \in L} \quad (bs_l^t \vee bm_l^t \vee bl_l^t) \rightarrow \neg r_l^t$$

Example 1 *Consider the graph below. To ease notation, we refer to locations as numbers.*



At each time-step, some location l will be asked to be reachable in some action's precondition. In other words, r_l will be asked to be true for some l in $1..6$.

The encoding for reachability would be the following (we use b_l as a shorthand for $bs_l \vee bm_l \vee bl_l$ and omit time superindexes).

$$\begin{array}{lll}
r_1 \rightarrow c_1 & e_{13} \rightarrow r_1 \wedge a_1 < a_3 & b_1 \rightarrow \neg r_1 \\
r_2 \rightarrow c_2 \vee e_{12} \vee e_{32} & e_{12} \rightarrow r_1 \wedge a_1 < a_2 & b_2 \rightarrow \neg r_2 \\
r_3 \rightarrow c_3 \vee e_{13} \vee e_{53} & e_{32} \rightarrow r_3 \wedge a_3 < a_2 & b_3 \rightarrow \neg r_3 \\
r_4 \rightarrow c_4 \vee e_{34} \vee e_{64} & e_{34} \rightarrow r_3 \wedge a_3 < a_4 & b_4 \rightarrow \neg r_4 \\
r_5 \rightarrow c_5 & e_{46} \rightarrow r_4 \wedge a_4 < a_6 & b_5 \rightarrow \neg r_5 \\
r_6 \rightarrow c_6 \vee e_{46} & e_{53} \rightarrow r_5 \wedge a_5 < a_3 & b_6 \rightarrow \neg r_6 \\
 & e_{64} \rightarrow r_6 \wedge a_6 < a_4 &
\end{array}$$

Translation of Ordering Constraints to SAT

The previous ordering constraints on the values of the numeric variables a associated to locations can be easily translated to SAT. We propose not to encode the numbers to binary form, but to encode the acyclicity relation directly to SAT.

A *strict partial order* is a relation $<$ that is irreflexive and transitive (which implies antisymmetry as well). This is all we need to ensure acyclicity. We can encode such a relation by adding the constraints

$$\begin{array}{rl}
\neg p_{ii}^t & \text{(irreflexivity)} \\
p_{ij}^t \wedge p_{jk}^t \rightarrow p_{ik}^t & \text{(transitivity)}
\end{array}$$

where p_{ij}^t are Boolean variables, for locations i, j and time t. Then, to get rid of SMT constraints and obtain a full SAT encoding, we can simply replace the constraints

$$\forall_{(l,l') \in E} \quad e_{l\,l'}^t \rightarrow (r_l^t \wedge a_l^t < a_{l'}^t)$$

by

$$\forall_{(l,l') \in E} \quad e_{l\,l'}^t \rightarrow (r_l^t \wedge p_{ll'}^t)$$

and no numeric variables a_l^t are needed at all.

Notice that transitivity constraints $p_{ij}^t \wedge p_{jk}^t \rightarrow p_{ik}^t$ are only needed for neighbours j of i, since transitivity follows by induction. This allows to reduce the number of such constraints from cubic to quadratic.

Grid Graph Reachability

Grids are a particular case of graphs, and a simpler encoding for reachability in them is possible. The idea is to build a single path from the source to the target as follows:

define a Boolean variable for each location denoting if it is included in the path, and impose that the source and destination are in the path and exactly one of their neighbours is in the path, and that any valid (non-obstacle) location different from the source and the destination has either zero or two neighbours in the path.

Not surprisingly, this simple approach is the best performing in our experiments.

Invariants

To help pruning the search space, some simple invariants can be considered. Since balls cannot decrease their size, the following constraints have to be fulfilled at any time: the number of large balls cannot exceed the number of snowmen, and there must be at least as many small balls as snowmen. Imposing these constraints has shown to be useful only in the harder instances.

Other invariants have been also considered, like forbidding to move balls to locations where they would get stuck without the possibility of building a snowmen, but this does not showed significant improvement.

Empirical Evaluation

For our experiments we used the planners Fast Downward (Helmert 2006) v22.12 and SymK (Speck et al. 2019) v3.0, and the SAT solver KISSAT (Biere et al. 2020) v3.0.0. These two planners were chosen due to their well-known performance and support for the required features present in the PDDL models. In particular, for Fast Downward we used three configurations: (i) the integrated Stone Soup portfolio (Seipp and Röger 2018) (ii) LAMA (Richter and Westphal 2010) and (iii) the blind heuristic. The Fast Downward Stone Soup portfolio won the satisficing and cost-bounded tracks of the 2018 International Planning Competition (IPC). When using Fast Downward in preliminary experiments, we observed that all configurations of Stone Soup always resorted to the blind heuristic in our instances, and therefore we only report results for LAMA and the blind heuristic. As SymK natively supports a variety of PDDL features that are rarely supported by other planners, such as conditional effects and derived predicates with axioms, we used its default bidirectional search. Finally, the KISSAT SAT solver and its variations were the winners of various tracks of the 2021 and 2022 SAT competitions (Froleyks et al. 2021).

To test the efficiency of the proposed approaches we considered all 30 instances of the main part of the game (the real world).[2] We also included a set of 9 *hidden* instances, that are present in the game files but not used in the actual game. Finally, we crafted 12 instances for some edge cases. More concretely, we considered much bigger scenarios,[3] instances with snow everywhere and some really easy to solve instances with two snowmen. In summary, we considered a total of 51 instances ranging from 1 to 3 snowmen and from 14 to 99 valid locations. We ran the experiments on a cluster of compute nodes equipped with Intel Xeon E-2234 CPU @ 3.60GHz processors, where each execution was given a time-limit of 1 hour and 16 GB of memory.

We considered three variations of the PDDL model: basic, cheating and reachability. *Basic* denotes the model with actions able to move the character as well as balls. *Cheating* is a model where we simply remove the action where the character moves, and the character can do "unsound" teleports. That is, it does not need to consider if there exists a path between a source and a target position before acting from the target position. Finally, *reachability* denotes a model where derived predicates are used to require reachability between subsequent action positions in the game. With these three models we aim to evaluate the effect of different encodings of reachability in PDDL.

Table 1 summarises all results.[4] The different SAT configurations are as follows: SAT refers to the basic SAT encoding without reachability constraints, SAT-cheating to a variation of the SAT encoding with "unsound" teleports, and the -R versions implement the reachability encodings with the ordering and counting variations, where + denotes the addition of implied constraints (invariants). The rest of the rows show the combined performance of each planner and PDDL variant considered. For example, basic-blind considers the basic model solved with Fast Downward using the

	Solved	PAR-2
SAT	27	196942
SAT-cheating	43(19)	62414
SAT-R-order	43	62818
SAT-R-count	43	60858
SAT-R-count+	43	60947
basic-blind	27	177082
basic-LAMA	14	275961
basic-SymK	29	166980
cheating-blind	25(16)	188086
cheating-LAMA	20(11)	234919
cheating-SymK	29(14)	162480
reachability-blind	26	183732
reachability-LAMA	16	263718
reachability-SymK	23	205981

Table 1: Number of solved instances (within the 1 hour timeout), and PAR-2 scores (the score of a solver is defined as the sum of all runtimes for solved instances + 2×timeout for unsolved instances) for each approach. For cheating models the number of valid solutions is given in parenthesis. The total number of instances considered is 51.

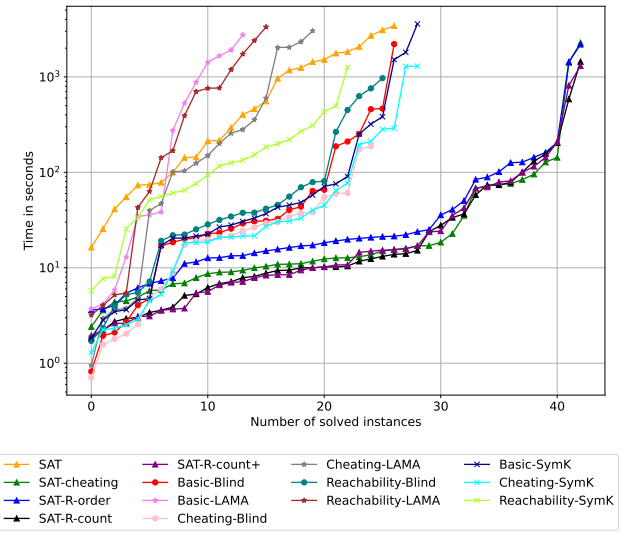


Figure 3: Cummulative number of solved instances for all the considered approaches. Time in seconds in logarithmic scale.

[2]The game has a second part (the dream world), governed by a different set of rules.

[3]The game scenarios contain at most 40 locations whilst the crafted ones have up to 99 locations.

[4]Detailed results are given in the supplementary material at https://github.com/udg-lai/KEPS2023.

blind heuristic.

Column Solved shows the number of instances solved by each system. For the cheating variants, it shows in parenthesis how many solutions are valid out of all the ones solved. Column PAR-2 shows the sum of all runtimes for solved instances + 2×3600 seconds per timed-out instances. The cheating and SAT-R variants are clearly the strongest approaches, solving 43 instances out of 51, and showing that the reachability encoding is critical to be able to obtain more (sound) solutions within the given time limit. The cheating variant is not faster than the reachability based ones. Notice that it has less variables but it also forbids less partial assignments, and therefore it may spend more time exploring possible moves. This effect may be amplified by the planning as satisfiability approach taken, as for finding a length-optimal plan it previously proves unfeasibility of all shorter plans.

When comparing the cheating and basic PDDL models, the cheating variant only results into notable gains with LAMA. When comparing the cheating model with the reachability model, there is a notable difference in both LAMA and SymK. Although not visible in the summarized results, amongst the PDDL approaches with two or three snowmen, only SymK was able to solve two of them within the given time limit. The rest of the planners were not able to preprocess any of them due to the increased complexity of the domain.

Following the PAR-2 scores, three main groups can be observed: the SAT-R variants as the best performers, SAT, blind search and SymK as the second group, and finally LAMA being notably slower. The implied constraints (invariants) did not seem to help in reducing the solving time, at least for the instances considered. Figure 3 depicts the cummulative number of solved instances over time. Note the time axis is in logarithmic scale. This figure allows to identify more clearly how fast are the three aforementioned groups.

Conclusions and Future Work

In this work we contributed new challenging planning benchmarks. In the framework of puzzle-like games, we believe that SAT models like the ones we have presented could be used to assist in the (semi)automatic design of new scenarios, for instance, by ensuring that randomly generated or crafted scenarios satisfy the desired constraints in terms of difficulty of the game levels.

We have shown how relatively simple encodings to SAT outperform by far state-of-the-art planners in the game *A Good Snowman is Hard to Build*, especially when considering reachability constraints.

Some works have introduced SAT and SMT solvers with support for detecting acyclicity and reachability in graphs (Gebser, Janhunen, and Rintanen 2014; Bayless et al. 2015). Therefore, an alternative approach could be to use a SAT or SMT solver with built-in support for reachability, such as MonoSAT (Bayless et al. 2015).

As future work, we plan to address the simultaneous (parallel) execution of actions as an additional way of reducing the time span, in order to overcome scalability issues and be able to solve the harder instances in a reasonable amount of time. This will require non-trivial checking of possible interferences between actions.

Acknowledgements

Work partially supported by grant PID2021-122274OB-I00 funded by MCIN/AEI/10.13039/501100011033 and by ERDF A way of making Europe.

References

Bayless, S.; Bayless, N.; Hoos, H. H.; and Hu, A. J. 2015. SAT Modulo Monotonic Theories. In Bonet, B.; and Koenig, S., eds., *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, January 25-30, 2015, Austin, Texas, USA*, 3702–3709. AAAI Press.

Biere, A.; Fazekas, K.; Fleury, M.; and Heisinger, M. 2020. CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling Entering the SAT Competition 2020. In Balyo, T.; Froleyks, N.; Heule, M.; Iser, M.; Järvisalo, M.; and Suda, M., eds., *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions*, volume B-2020-1 of *Department of Computer Science Report Series B*, 51–53. University of Helsinki.

Culberson, J. 1997. Sokoban is PSPACE-complete. Technical report, 97-02, Department of Computer Science, University of Alberta.

Davis, B.; and Hazelden, A. 2015. A Good Snowman Is Hard To Build. https://agoodsnowman.com/. [Online; accessed 23-March-2023].

Froleyks, N.; Heule, M.; Iser, M.; Järvisalo, M.; and Suda, M. 2021. SAT Competition 2020. *Artificial Intelligence*, 301: 103572.

Gebser, M.; Janhunen, T.; and Rintanen, J. 2014. SAT Modulo Graphs: Acyclicity. In *Logics in Artificial Intelligence - 14th European Conference, JELIA 2014, Funchal, Madeira, Portugal, September 24-26, 2014. Proceedings*, 137–151.

Haslum, P.; Lipovetzky, N.; Magazzeni, D.; and Muise, C. 2019. *An Introduction to the Planning Domain Definition Language*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers.

He, W.; Liu, Z.; and Yang, C. 2017. Snowman is PSPACE-complete. *Theoretical Computer Science*, 677: 31 – 40.

Helmert, M. 2006. The Fast Downward Planning System. *J. Artif. Intell. Res.*, 26: 191–246.

Ivankovic, F.; and Haslum, P. 2015. Optimal Planning with Axioms. In Yang, Q.; and Wooldridge, M. J., eds., *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*, 1580–1586. AAAI Press.

Kautz, H. A.; and Selman, B. 1992. Planning as Satisfiability. In *10th European Conference on Artificial Intelligence, ECAI 92, Vienna, Austria, August 3-7, 1992. Proceedings*, 359–363.

Miura, S.; and Fukunaga, A. 2017. Automatic Extraction of Axioms for Planning. In *Proceedings of the Twenty-Seventh International Conference on Automated Planning and Scheduling, ICAPS 2017, Pittsburgh, Pennsylvania, USA, June 18-23, 2017.*, 218–227.

Rankooh, M. F.; and Rintanen, J. 2021. Propositional Encodings of Acyclicity and Reachability by using Vertex Elimination. *CoRR*, abs/2105.12908.

Richter, S.; and Westphal, M. 2010. The LAMA Planner: Guiding Cost-Based Anytime Planning with Landmarks. *J. Artif. Intell. Res.*, 39: 127–177.

Seipp, J.; and Röger, G. 2018. Fast downward stone soup 2018. *IPC2018–Classical Tracks*, 72–74.

Silli, E. 2010. Mirror's Edge - Level Design Challenges & Solutions. Paper presented at GDC Europe 2010, Cologne, Germany.

Speck, D.; Geißer, F.; Mattmüller, R.; and Torralba, Á. 2019. Symbolic Planning with Axioms. In Lipovetzky, N.; Onaindia, E.; and Smith, D. E., eds., *Proceedings of the Twenty-Ninth International Conference on Automated Planning and Scheduling (ICAPS 2019)*, 464–472. AAAI Press.