On Grid Graph Reachability and Puzzle Games

Miquel Bofill ✉ ⌂ iD
Departament d'Informàtica, Matemàtica Aplicada i Estadística, Universitat de Girona, Spain

Cristina Borralleras ✉ iD
Departament d'Enginyeries, Universitat de Vic - Universitat Central de Catalunya, Spain

Joan Espasa ✉ iD
School of Computer Science, University of St Andrews, UK

Mateu Villaret ✉ iD
Departament d'Informàtica, Matemàtica Aplicada i Estadística, Universitat de Girona, Spain

──── **Abstract** ────

Many puzzle video games, like Sokoban, involve moving some agent in a maze. The reachable locations are usually apparent for a human player, and the difficulty of the game is mainly related to performing actions on objects, such as pushing (reachable) boxes. For this reason, the difficulty of a particular level is often measured as the number of actions on objects, other than agent walking, needed to find a solution. In this paper we study CP and SAT approaches for solving these kind of problems. We review some reachability encodings and propose a new one. We empirically show that the new encoding is well-suited for solving puzzle problems in the planning as SAT paradigm, especially when considering the execution of several actions in parallel.

1 Introduction

The problem of reachability can be understood as whether a vertex can be reached from another in a graph. Given its generality and usefulness, reachability is used in many and varied settings, where it is normally not used alone but as part of a more complex problem. For example, in (single agent) puzzle video games like Sokoban [9, 29], the difficulty of a level is often measured with the number of times the agent needs to move a reachable object in a grid. There are two main reasons for this: first, moving the agent from one location to another is almost trivial for a human player, because it consists of determining if there exists a path from one location to another; second, pushing an object may close or open free paths.

Some works devise methods to focus the search efforts on certain parts of the problem. In [18], the authors show how a reachability derived predicate can be specified with axioms and agent moving 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 agent 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 [22]. This is done with success for Sokoban and reachability, as shown by using axiom supporting model-based planners with Integer Programming (IP) and Answer Set Programming (ASP) technologies.

In this paper we review some well-known reachability encodings and propose a new one. To demonstrate its effectiveness, we use two case studies based on hard (PSPACE) video games [9, 17]. The main contribution of the work is on the suitability of the presented encoding for computing connected components in undirected graphs. In particular, it can be used to compute the connected component of an agent in a grid-based puzzle game at a given time step, i.e., the set of reachable locations by the agent. We argue that the proposed encoding results in a reduced search space in this setting, compared to existing reachability encodings, making it more efficient.

The paper proceeds as follows. In Section 2 we revisit some existing graph reachability encodings and introduce a new one. Section 3 introduces the two case studies we consider for our experimental evaluation, namely the games *A good Snowman is hard to build* and *Sokoban*. In Section 4 we briefly recall the planning as SAT approach and provide different solutions for the Snowman game: without reachability, with reachability and with reachability and parallelism. We also summarize our MiniZinc solutions with and without reachability. Finally, in Section 5, we make an empirical evaluation of the distinct methods proposed for the Snowman game, devise a specific algorithm for seeking optimality while using parallelism, and show that the new reachability encoding is the best suited for parallelism in Snowman as well as in a large set of Sokoban instances.

2   Graph Reachability Constraints

Here we review some standard encodings for reachability constraints, and propose a new one.

2.1   st-Connectivity

The problem of st-connectivity consists in determining, for a pair of vertices s and t in a graph, whether t is reachable from s. This problem has been widely studied due to its practical interest. Its complexity, for the restricted case of planar graphs, has been studied in [1], showing that it is complete for nondeterministic logspace (NL). We are interested in declarative approaches to st-connectivity.

A thorough study on logic-based characterizations of acyclicity and reachability conditions, and their corresponding encodings in the language of answer set programming can be found in [13]. A generic encoding for reachability would look like the following:

```
reachable(T,S) :- S = T.
reachable(T,S) :- reachable(T,S1), adjacent(S1,S).
```

The way of characterising reachability is intrinsically different in SAT than in ASP. In particular, we cannot mimic the above rules, where a location t is reachable from s either if s equals t or there is some neighbour s' of s such that t is reachable from s'. The reason is that, whereas ASP adheres to the closed-world assumption (i.e., all unknown values are assumed to be false), this is not the case for SAT. In other words, the direct translation of ASP reachability rules to SAT would be satisfied by any model where all corresponding reachability variables are set to true, hence not encoding reachability at all.

Nevertheless, ASP programs can be translated into SAT and, in fact, some ASP solvers use a SAT solver as a backend. The standard way of eliminating the closed-world assumption for an atom p is by adding the rule $p \leftarrow not\, not\, p$ [21]. But, naturally, reachability can be directly encoded in SAT. In order to deal with the open-world assumption, 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 [12], acyclicity can be easily modelled with SMT, by imposing an ordering on locations based on numeric values associated to them. Considering a directed graph $G = (V, E)$ and a source vertex $s \in V$, the encoding goes as follows. For every $v \in V$, a Boolean variable r_v denotes if vertex v is reachable from s. For every edge $(v, v') \in E$, a Boolean variable $e_{vv'}$ to indicates if edge (v, v') is in the reachability path. Moreover, to avoid cycles in those paths, an integer variable $a_v \in 1..|V|$ is included for each vertex $v \in V$. Cycles are forbidden by enforcing a topological ordering between the vertices in the reachability path. The constraints are:

$$r_s \qquad \forall_{v \in V \setminus \{s\}} \left(r_v \to \bigvee_{(v',v) \in E} e_{v'v} \right) \qquad \forall_{(v,v') \in E} \left(e_{vv'} \to r_v \wedge a_v < a_{v'} \right)$$

Additionally, a unit clause r_t must be imposed to require reachability from s to some desired vertex t. Note that the last are SMT constraints because of the presence of difference logic atoms $a_v < a_{v'}$. MiniZinc provides the global constraint `path` for *st*-connectivity, whose implementation resembles very much this one. Related work on global constraints and propagators for reachability can be found in [4, 10, 24].

Translation of Ordering Constraints to SAT

The previous ordering constraints on the values of the numeric variables a associated to vertices 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 $\neg t_{vv}$ (irreflexivity) and $t_{vv'} \wedge t_{v'v''} \to t_{vv''}$ (transitivity) where $t_{vv'}$ are Boolean variables, for vertices v, v'. Notice that transitivity constraints $t_{vv'} \wedge t_{v'v''} \to t_{vv''}$ are only needed for neighbours v' of v, since transitivity follows by induction. The encoding is then $\mathcal{O}(NM)$ size (for N vertices and M edges), with $\mathcal{O}(N^2)$ variables.

A similar encoding is given in [12], based on the transitive closure of the relation corresponding to the underlying graph: variables $t_{vv'}$ indicate that (v, v') is in the transitive closure, variables $e_{vv'}$ for edges (v, v') imply $t_{vv'}$, transitivity is expressed by $e_{vv'} \wedge t_{v'v''} \to t_{vv''}$, and cycles are forbidden by $e_{vv'} \to \neg t_{v'v}$. This encoding is also $\mathcal{O}(NM)$ size, with $\mathcal{O}(N^2)$ variables. Therefore, no numeric variables are needed at all. By replacing the third group of constraints by $\forall_{(v,v') \in E} \left(e_{vv'} \to r_v \wedge t_{vv'} \right)$, we obtain a full SAT encoding which is $\mathcal{O}(NM)$ size, with $\mathcal{O}(N^2)$ variables.

Note that this encoding computes a directed acyclic graph covering some subset of the reachable vertices (including the desired target). For this reason, in the following we will refer to it as *DAG encoding*.

2.2 A Simple Encoding for Grids

Grids are a particular case of graphs, and a simpler encoding for reachability in them is possible. The idea is to build a 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 (i) the source and target are in the path, (ii) if the source and target are different then each of them has exactly one neighbour in the path, and (iii) any location in the path different from the source and the target has exactly two neighbours in the path.

This encoding is $\mathcal{O}(N)$ size, with $\mathcal{O}(N)$ variables, given that the number of neighbours of each location is at most 4. Since it computes a path from the source to the target, we will refer to it as *path encoding*. It is worth noting that a path cannot cycle back through a

neighbour, e.g., $(1,1) - (1,2) - (1,3) - (2,3) - (2,2)$ would not be allowed, since $(1,2)$ has 3 of its neighbours in the path. But this is correct if all we are interested in is reachability. Moreover, the proposed constraints allow for unrelated cycles in addition to the path. Note that the trick of restricting the source and destination to have only one neighbour in the path is precisely what avoids a cycle, but then paths which are disconnected from the source and the destination are possible, in form of a cycle. These disconnected cycles cannot be easily avoided with additional constraints, since this is again a matter of connectedness.

2.3   A New SAT Encoding for st-Connectivity in Undirected Graphs

As said, the standard encoding for st-connectivity given in Section 2.1 computes a directed acyclic graph covering some subset of the reachable vertices (including the desired target). In this section we present a stronger encoding, which computes a tree rooted at the source vertex that covers all reachable vertices, i.e., a spanning tree of a graph covering all reachable vertices. This makes sense especially for undirected disconnected graphs, where we may want to know the connected component of a vertex.

The encoding is given for undirected graphs without self-loops. In the following constraints, the variables r_v denote if a vertex v is reachable from the source s, and the $t_{vv'}$ variables denote the existence of a path from vertex v to v' in the tree rooted at s.

$$r_s \tag{1}$$

$$\forall_{(v,v')\in E} \quad r_v \to r_{v'} \tag{2}$$

$$\forall_{v\in V:(s,v)\in E} \quad t_{sv} \tag{3}$$

$$\forall_{v\in V\setminus\{s\}} \quad r_v \to \bigvee_{(v,v')\in E} t_{v'v} \tag{4}$$

$$\forall_{(v,v')\in E,\, (v,v'')\in E,\, v'\neq v''} \quad \neg t_{v'v} \vee \neg t_{v''v} \tag{5}$$

$$\forall_{(v,v')\in E,\, v''\in V,\, v'\neq v''} \quad t_{vv'} \wedge t_{v'v''} \to t_{vv''} \wedge \neg t_{v''v} \tag{6}$$

$$\forall_{(v,v')\in E} \quad t_{vv'} \vee t_{v'v} \to r_v \tag{7}$$

Equation (1) sets the source vertex s as reachable, and Equation (2) propagates reachability to neighbours. Equation (3) sets outgoing paths from s to its neighbours. Equation (4) forces at least one path into each reachable vertex, except for s, while Equation (5) forces at most one path into each vertex. Therefore, Equations (4) and (5) force exactly one path into each reachable vertex, except for s. Moreover, Equation (6) defines transitivity of paths and forbids cycles. Therefore, Equations (3)–(6) define a tree rooted at s of vertices reachable from s. Moreover, thanks to Equations (1) and (2), that tree will span to all reachable vertices. Finally, Equation (7) sets to reachable any vertex in a path.

It is worth noting that the tree of paths defined by the $t_{vv'}$ variables is essential for setting to reachable exactly the vertices that are reachable from the source. The key idea is the following. Since in Equation (4) we are forcing some path into every reachable vertex different from the source, and cycles are forbidden by Equation (6), at least one vertex must be set to unreachable in areas disconnected from the source. Then, in setting that vertex to unreachable, unreachability spreads out to its neighbours by Equation (2) (if reachability spreads out, so does non-reachability).

Note also that Equation (7) is not needed to correctly set the value of the r_v variables, but it forces the $t_{vv'}$ variables to false in disconnected components, thus reducing the search space.
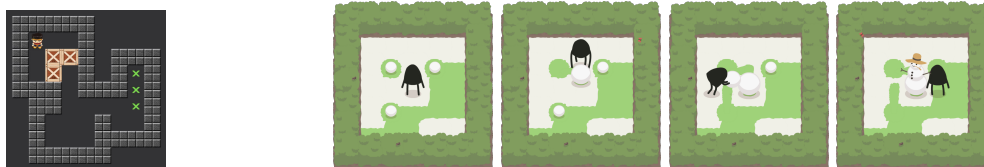
Figure 1 Left: a Sokoban problem instance. Right: *Andy* level of the Snowman game, showing the execution of the optimal solution *lluRurDllddddrUluRuurrrdLulD*. Letters represent the direction of movement. Uppercase letters indicate snowball movements.

This encoding is again $\mathcal{O}(NM)$ size, with $\mathcal{O}(N^2)$ variables, for N vertices and M edges. We will refer to it as *spanning tree encoding*.

3   Puzzle Games

We are interested in analyzing the suitability of the presented approaches and encodings for reachability in solving puzzle problems. Here we describe the puzzle games that we consider.

3.1   A Good Snowman is Hard to Build

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 snowballs of decreasing size. Snowman was released in 2015, and proved PSPACE-complete in 2017 [17].

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

The agent is not allowed to pull snowballs. The goal is to build snowmen composed by a pile of three snowballs of decreasing size. The scenarios considered consist of three, six or nine snowballs, hence, one, two or three snowmen. Snowmen can be built anywhere. Figure 1 (right) depicts an example of how to solve one of the levels of the game.

3.2   Sokoban

Sokoban is a well-known PSPACE-complete [9] challenging puzzle game. Each puzzle consists of a maze formed by inaccessible wall squares and accessible floor squares. There is a single

agent (the Sokoban) which can walk on floor locations (unless occupied by some box), and push single boxes onto unoccupied floor locations. The goal is to push all boxes onto a set of designated storage locations. One of the sources for the difficulty of this game is that many pushes are irreversible, leading to dead-end states from which reaching the goal is impossible. An example of the maze is given in Figure 1 (left).

4   Reachability in Puzzle Games

4.1   Planning as SAT

The problem of planning, in its most basic form, consists in finding a sequence of actions (a plan) that allows to transform an initial state into a goal state [16]. In the classical planning setting, finding out if there is a plan is PSPACE complete [8], while deterministic planning with numerical variables is undecidable in general [7].

A planning problem can be defined as a tuple (V, A, I, G), where V is a finite set of state variables, A is a finite set of action templates, I is the initial state and G is the goal. A state is a valuation over V, i.e., a function mapping each variable $v \in V$ to a value in its domain ($\{true, false\}$ in the Boolean case). The goal is a set of states (usually defined as a set of propositions that a goal state must satisfy). Actions $a = \langle Pre, Eff \rangle \in A$ are defined as pairs of preconditions and effects. Preconditions describe which are the requirements on the state to execute the action, whilst effects describe how the state is changed after its execution. Both preconditions and effects are typically given as sets of literals.

Plans and ASP stable models (or answer sets) are connected, as described in [27]. Moreover, propositional satisfiability and answer set programming are two closely related research areas [21]. Some ASP solvers, like *clasp* [15], combine the high-level modeling capacities of ASP with state-of-the-art techniques from the area of Boolean constraint solving. In fact, the primary *clasp* algorithm relies on conflict-driven nogood learning. Moreover, as said, some ASP solvers use a SAT solver as a backend. It also often occurs that planning problems can be directly encoded into SAT with no major difficulty.

In the planning as SAT approach [20], a planning problem is encoded to a Boolean formula, with the property that any model of this formula corresponds to a valid plan. Since the length of a valid plan is not known a priory, 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 length 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 denotes 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 and effects: for every $a = \langle Pre, Eff \rangle \in A$ and $t \in 0..T - 1$, we have $a^t \to Pre^t$ and $a^t \to Eff^{t+1}$, where Pre^t and Eff^{t+1} denote the corresponding formulas (conjunctions of literals) on the time-indexed state variables. Moreover, a change in the value of a state variable v can occur only if an action that can change this value is executed: for every variable $v \in V$ and $t \in 0..T - 1$, we have the frame axiom $v^t \neq v^{t+1} \to \bigvee\{a^t \mid a = \langle Pre, Eff \rangle \in A, v \in Eff\}$. Finally, it is stated that exactly one action is executed at each time step t, and that the goal holds at time T.

4.1.1   Sequential Plans

Here we propose an encoding for solving the Snowman problem, following a planning as SAT approach. It can be straightforwardly adapted for the similar yet simpler Sokoban problem.

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. The encoding is valid for any number of snowmen.

We represent states with Boolean variables stating, for each location, whether (i) there is snow or not, (ii) there is a snowball of a particular size or not, and (iii) there is the agent or not. The actions considered are only four, corresponding to a snowball movement per possible direction. This will eventually result in rolling, pushing or popping a snowball, 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, and T the number of time steps considered. For all $l \in L$ and $t \in 0..T$, we have the following variables: 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 snowball at location l at time t), c_l^t (the character is at location l at time t). And for all $t \in 0..T-1$: n^t, s^t, e^t, w^t (direction of action is north, south, east or west). The goal consists in requiring no partial snowman at any location: $\forall l \in L \ (bs_l^T \leftrightarrow bm_l^T) \wedge (bm_l^T \leftrightarrow bl_l^T)$. For each time step t in $0..T-1$ we have the following constraints:

- Exactly one action (n^t, s^t, e^t, w^t) is executed.
- 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 agent 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 agent walks north and it has a wall two locations ahead, there cannot be any snowball in front of him, and at the next time step his location 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 agent walks north without having a wall two locations ahead, apart from moving, it can also *roll* a snowball north, *push* a snowball into a stack of snowballs or *pop* a snowball from a stack of snowballs. 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):

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

- 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 snowball occupies that location, etc.: $\forall l \in L \quad (\neg s_l^t \rightarrow \neg s_l^{t+1}) \wedge (s_l^t \wedge \neg s_l^{t+1} \rightarrow bm_l^{t+1} \vee bl_l^{t+1}) \wedge \cdots$. Note that in some cases we don't use the actions as the reason for change, but it is enough to consider some of their effects, such as a snowball appearing.

4.1.1.1   Reducing the Search Space

In order to keep the search space small, we can consider the agent walking to a snowball and rolling, pushing, popping it in one direction as an atomic action. As is usually done in Sokoban, we could also consider rolling a snowball a certain number of locations as a single action but, since the snowball can remove snow (and increase its size) when rolling, this should be restricted to locations with no snow. Collapsing actions is motivated by the fact that the agent will walk only to move some snowball, so considering walking an action on its own is superfluous. Since reachability is obvious for a human player, the walking actions shouldn't be considered when measuring the difficulty of a scenario.

The presented encoding can be easily adapted to collapsing actions. Removing *move* (i.e., walk) actions essentially reduces to require the agent being at most at one location, and this location being reachable from the previous one, before each snowball action taking place. Any of the reachability encodings presented in Section 2 could be used, considering walls as well as snowballs as obstacles, with the *path encoding* being the best suited.

Some simple invariants can also be considered. For instance, since snowballs cannot decrease their size, it can be imposed that the number of large snowballs never exceeds the number of snowmen, and there are at least as many small snowballs as snowmen.

4.1.2   Parallel Plans

Another way to further reduce the search space, as well as to break symmetries, is to consider the execution of several actions in parallel. This has been extensively studied in the AI planning community [25, 28, 2].

Here we consider solving the Snowman problem by adhering to the so-called ∀-step semantics of parallel plans.[1] This implies that any ordering of the actions in the parallel plan must result in a valid sequential plan. Therefore, interfering actions cannot be scheduled at the same time.

Thanks to reachability constraints, we are omitting *move* (walking) actions. Therefore, we only need to consider *roll*, *push*, and *pop* actions. What is more, reachability will play a crucial role in avoiding interference between such actions.

Direct Interference   In Snowman, by direct interference between a pair of actions we refer to the case where the affected locations (i.e., the source and destination of the involved snowballs) do intersect. Constraints avoiding this kind of interference are straightforward.

Indirect Interference   Let's consider a set of actions that do not directly interfere and, moreover, their locations are reachable from the current agent location. Indirect interference between these actions is possible if executing some of them makes some other action location unreachable, thus preventing its execution.

Avoiding this kind of interference is not straightforward. In particular, given a set of (candidate) parallel actions, it is not enough to require all action locations to be reachable before and after executing all of them. The following example shows a situation with two actions whose locations are reachable both before and after their parallel execution, but there is no way of sequencing them.

▶ **Example 1.** In the following grids, greyed squares denote walls, ⛄ denotes the current location of the agent, circles denote snowballs (of different sizes), and arrows denote both action locations and the direction of the action (for locations reachable by the agent).


---
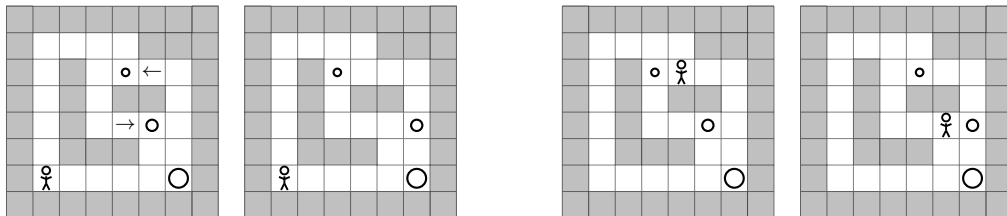
[1] The same ideas are valid for the simpler case of Sokoban.

As can be observed, two *roll* action locations are reachable by the agent both before and after executing them (left-left and left-right figures). But this does not imply the actions can be serialized. If we first move the small snowball, we get a situation where the other action location becomes unreachable (right-left figure). If we first move the medium snowball, an analogous situation occurs (right-right figure).

Therefore, we need to be more restrictive in parallel scenarios. A simple idea, in order to be able to serialize the execution of parallel actions, is to ask for reachability of all action locations *considering the present and future snowball locations as occupied.* This way, as we are only adding obstacles to the current situation, future reachability of action locations is guaranteed.[2] However, this limitation is way too restrictive: although it allows for serialization of parallel actions fulfilling it, it can prevent the execution of single actions under certain circumstances. The following example illustrates this situation.

▶ **Example 2.** In the situation below, if we consider the present and future location of the snowball as occupied, the action could not be performed.
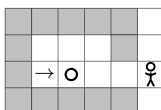
Therefore, when a single action is blocking itself the path used to reach its location, we cannot be so restrictive. The solution we propose is to add a new action to *jump* (walk) to a reachable location. This action will be used exclusively, while maintaining the aforementioned constraints on reachability (considering present and future snowball locations) only for *roll*, *push* and *pop* actions. This way, *roll*, *push* and *pop* actions can be executed safely in parallel, while jumping will be performed individually, when needed. It is worth noting that, thanks to the proposed reachability restrictions, the agent can stay put during any sequence of (combined) *roll*, *push* and *pop* actions. This contributes to reduce the search space.

It is not difficult to see that the proposed system is sound and complete with respect to finding a valid sequential plan. Concerns about optimality are discussed in the next subsection.

Upper Bounds and (Sub)optimality

From the serialization of a parallel plan we will obtain a valid plan. Note however that this plan will not only contain snowball actions (*roll*, *push* and *pop*), but possibly some walking (*jump*) actions. Moreover, the number of snowball actions may easily become suboptimal. A parallel plan of a minimal number of timesteps can nevertheless contain useless actions like, e.g., rolling some snowball back and forth while other necessary actions are being performed. However, this is not the only source of suboptimality. As the following example shows, two parallel plans of the same number of steps can result in two sequential plans of different length, even if none of the parallel plans contains any useless action (i.e., when removing some of the actions would result in an invalid plan).

▶ **Example 3.** The left group shows a parallel plan consisting of three steps, where two actions are performed in parallel at the first step. The right group shows an alternative

---

[2] Observe that, with this restriction, the parallel execution of the two actions of Example 1 would be forbidden, but they could be individually performed.

parallel plan of three steps, which is already a sequential plan. Both plans are valid and contain no useless actions. However, when serializing them, the first one results in a four steps plan.
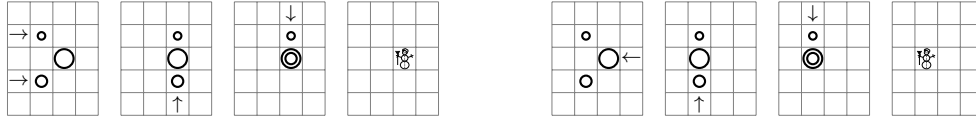
The previous example demonstrates that parallel plans are inherently suboptimal. In other words, the serialization of a minimal length parallel plan does not necessarily turn into a minimal length sequential plan, even when containing no useless actions. Therefore, a parallel plan will give us an upper bound on the number of necessary snowball actions.

4.2   Planning as CSP

Following a similar solving approach as the one described in previous section, here we consider the more expressive CSP framework using the MiniZinc [23] language. A state of the Snowman game can be naturally modelled with the coordinates of the agent and a matrix of integer variables. We can use a unique integer for each possible state of a cell. For example, 1 to denote a small snowball, 2 a medium one, 3 a small one on top of a medium one, . . . 7 a complete snowman, etc. Identifying the goal state consists in certifying the existence of as many cells of complete snowmen as required by the instance. This can easily be done by counting the amount of 7 with the `count` global constraint.

Recall that the only allowed action is moving the agent around the maze in a given direction. Again, the four directions are encoded using unique integers. A movement will translate into rolling, pushing or popping snowballs next to the agent, or simply walking into a free cell. To encode the movement of snowballs, we use three auxiliary variables per timestep: *action* indicates the position of the snowball to be moved, *next* indicates the resulting position of the snowball once the action is performed, and *prev* the position from where the agent pushes the snowball. These variables make the encoding of preconditions and effects easier. For example, when pushing a snowball, we disallow the *next* location to be a wall or to contain snowballs smaller than the one pushed.

We evaluate two MiniZinc models, one considering the movement of the character cell by cell, and another only enforcing that consecutive cells where actions take place are reachable. Respectively, we either enforce that the agent must be in the *prev* location, or that the *prev* location must be reachable from the current agent location. The reachability constraint can be stated using the `path` global constraint. This constraint implements *st*-connectivity similarly to the *DAG encoding* described in Section 2.1. It only differs by having a variable representing the distance from the source to each node, instead of an ordering variable. The distance to a node is defined by adding 1 to the distance from the source to its previous node in the path. Note that this approach forbids cycles. Alternatively, by using the global constraint `count` we also replicate the *path encoding* of Section 2.2 with MiniZinc.

Implied constraints similar to the invariants on the occurrences of snowballs of distinct sizes described in Section 4.1.1.1 can be imposed using the `global_cardinality_low_up` global constraint. Finally, we used a search strategy that sequentially decides on the ordered timesteps, i.e., by first looking for values of variables of the first timestep, then the ones of the second timestep, and so on.

5 Empirical Evaluation

In this section we evaluate the efficiency of the presented encodings on the Snowman and Sokoban problems. Experiments were run on a cluster of compute nodes equipped with Intel Xeon E-2234 CPU @ 3.60GHz processors and 16 GB of memory. As SAT solver, we used Kissat [5] in version 3.0.0. Kissat and its variations were the winners of various tracks of the 2021 and 2022 SAT competitions [11]. For MiniZinc we used version 2.7.2 with Chuffed 0.11 as a backend solver.

5.1 Snowman

For the Snowman problem, we consider the 30 levels of the base game. Table 1 compares the performance of Kissat and MiniZinc (with Chuffed) using the planning as SAT and planning as CSP approaches described in Sections 4.1.1 and 4.2, respectively. In both cases, we consider the setting with agent movements, and the setting where only snowball movements are considered (where *DAG* and *path* denote the used reachability encodings). For Kissat we only report on the *path encoding* described in Section 2.2, as the other reachability encodings performed very similarly on the same benchmarks. For MiniZinc we report on both the *DAG* encoding corresponding to the built-in `path` constraint, and the *path encoding*. As can be observed, both the number of steps and the time required to find a solution dramatically decrease when agent movements are ignored. The MiniZinc *path encoding* performs slightly better than the *DAG* one, perhaps due to its compactness. When comparing SAT with MiniZinc, clearly the SAT approach works better and solves more instances in all settings. From now on, we will focus on the SAT approach.

As explained in Section 4.1.2, the search space can be further reduced in the planning as SAT paradigm, by considering the execution of several actions at the same time. However, not all reachability encodings are equally well-suited for a parallel approach. The *spanning tree encoding* is probably the best-suited, since it sets to reachable exactly all reachable locations, i.e., it determines the connected component of the agent. This means that it needs no adaptation when moving from a sequential to a parallel setting and, moreover, it translates to a small number of models. The *DAG encoding* can be easily adapted to the parallel setting, by requiring reachability for all action locations. However, it can freely set other reachable locations either to reachable or not. This, in general, is going to translate into more (partial) models and, in case of a sequence of unsatisfiable instances (like the ones we will face in a planning as SAT approach), it could be a drawback. Finally, the *path encoding*, although probably being the best-suited for the sequential setting due to its small size, cannot be easily adapted for the parallel case unless it is replicated for every considered path. In case that many paths need to be considered simultaneously this would result in a considerable increase of the formula size and, moreover, the resulting formula would be highly symmetric.

In Table 2 we compare the performance of the different reachability encodings on the Snowman problem, in the planning as SAT approach. For the sequential setting, we consider the *path encoding* (labelled *path sequential*). For the parallel setting, we consider the three different encodings: the first one (labelled *path parallel*) consists in replicating the *path encoding* for each snowball, the second one (labelled *DAG parallel*) is adapted from the *DAG encoding* by setting as reachable each action location, and the third one (labelled *tree parallel*) is the original *spanning tree encoding*. We also increased the time limit from 1h to 8h. Therefore, the *path sequential* columns of Table 2 are almost the same as the SAT *path* columns of Table 1, the only difference being that "Ben & Alan" is solved in just over an

■ **Table 1** Lower bounds on the number of actions for the base instances of Snowman, considering all movements (left part), and only snowball movements (right part). Time in seconds (limit 1h, '-' for timeout). Best times, and best lower bound in case of timeout, are marked in bold.

	Mzn		SAT		Mzn *DAG*		Mzn *path*		SAT *path*	
	LB	time	LB	time	LB	time	LB	time	LB	time
Adam	52	-	67	**1767.39**	12	16.37	12	13.05	12	**11.94**
Alex	37	-	50	**1246.96**	13	34.25	13	26.95	13	**11.10**
Alice	28	-	**45**	**-**	14	-	14	-	19	**102.68**
Andy	19	390.41	19	**74.42**	6	5.19	6	**4.60**	6	7.07
Ben & Alan	27	-	**31**	**-**	10	-	10	-	**27**	**-**
Chris	31	114.52	31	**78.28**	7	5.70	7	**3.83**	7	6.62
Cynthia & Michael	32	-	**51**	**-**	10	-	10	-	30	1313.26
David	21	-	23	**294.15**	7	19.37	7	16.06	7	**9.62**
Freya	30	-	38	**212.82**	13	49.11	13	48.00	13	**13.53**
Helen	31	-	42	**1437.00**	11	24.38	11	21.98	11	**7.52**
Jack & Jill	26	-	**45**	**-**	13	-	13	-	16	**9.22**
Jessica & Amelia	27	-	**43**	**-**	10	-	10	-	16	**23.69**
Julian	30	-	**47**	**-**	13	641.80	13	610.05	13	**14.10**
Kate	36	571.20	36	556.82	10	12.15	10	9.83	10	**6.49**
Kevin	32	-	38	**402.10**	11	27.02	11	22.41	11	**7.02**
Lauren	40	-	42	**215.79**	11	14.56	11	9.02	11	**4.56**
Louise	33	**138.07**	33	144.10	13	22.31	13	17.2	13	**9.39**
Lucy	19	56.13	19	**41.19**	8	9.97	8	7.31	8	**3.12**
Lydia	27	**42.64**	27	55.33	7	7.36	7	4.33	7	**2.37**
Mary	41	**79.06**	41	142.78	10	7.74	10	5.40	10	**2.75**
Paul	34	-	**64**	**3430.39**	20	-	20	-	26	**67.73**
Rebecca	24	**18.83**	24	25.55	6	5.30	6	3.39	6	**1.82**
Rob, James & Matthew	19	-	**35**	**-**	8	-	8	-	**18**	**-**
Ryan	41	-	**52**	**2705.31**	15	51.74	15	40.64	15	**15.41**
Sally	48	-	**60**	**1175.77**	13	26.85	13	21.21	13	**8.55**
Sarah	26	219.26	**26**	**73.32**	8	7.41	8	5.52	8	**2.97**
Tanya	17	**16.05**	17	16.38	5	2.79	5	2.54	5	**1.54**
William	49	**263.01**	49	461.87	15	28.70	15	22.00	15	**10.09**
Willow	33	-	**52**	**-**	14	453.77	14	389.63	14	**15.05**
Zoe & Richard	24	-	**50**	**-**	10	-	10	-	17	**33.08**

hour, increasing its lower bound from 27 to 28, whereas "Rob, James & Matthew" appears to be a hard instance, keeping the same lower bound after 8 hours of computation. We also observe a clear gain in time on the hard instances in the parallel setting, being the tree based encoding the best performing on most of the hardest instances (marked in red). It is worth noting the discrepancies in the upper bounds found, which are inherent to parallel plans as explained in Section 4.1.2. Conclusions on performance of the encodings cannot be drawn from a such a small set of instances since, as is well-known, many aspects can cause the SAT solver to search differently, causing dramatic changes in the runtime on a given instance (see, e.g., the time required on "Rob, James & Matthew" when using the *DAG encoding*). It is apparent that the parallel approach performs better than the sequential one, especially on the hardest instances. Moreover, the upper bounds found are, in general, close to the optimum. Therefore, the question arises: which is the best strategy to solve the problem? Our proposal is to first find an upper bound on the number of snowball movements by following a planning as SAT strategy in parallel, then serializing the plan and seeking for shorter plans sequentially, until we get a negative answer. This strategy forces to include *noop* actions (i.e., null actions) since, sometimes, given a valid plan with n actions, a plan with exactly $n-1$ actions is not possible, but plans with fewer actions are. Interestingly,

■ **Table 2** Lower bounds (found sequentially) and first upper bounds (found in parallel) on the number of snowball actions for the base instances of the Snowman problem, considering different reachability encodings. Time in seconds (limit 8h). Best times in bold.

	path sequential		path parallel		DAG parallel		tree parallel	
	LB	time	UB	time	UB	time	UB	time
Adam	12	11.94	15	7.66	14	**3.38**	14	7.32
Alex	13	11.10	13	9.78	13	**4.98**	13	9.74
Alice	19	102.68	19	89.60	19	**67.07**	19	82.12
Andy	6	7.07	6	6.99	6	**2.25**	9	6.59
Ben & Alan	28	4101.25	36	1062.17	38	271.69	34	**201.32**
Chris	7	6.62	7	7.14	7	**2.42**	7	6.42
Cynthia & Michael	30	1313.26	30	314.69	32	146.87	30	**105.15**
David	7	9.62	8	8.21	7	**4.32**	8	5.39
Freya	13	13.53	17	11.46	15	**5.68**	17	7.32
Helen	11	7.52	12	**3.29**	11	3.86	11	4.42
Jack & Jill	16	9.22	16	8.99	19	**8.19**	16	9.47
Jessica & Amelia	16	23.69	20	14.70	20	**13.83**	22	16.10
Julian	13	14.10	13	**13.74**	13	15.84	13	21.14
Kate	10	6.49	11	**3.08**	11	3.42	11	4.11
Kevin	11	7.02	12	**6.44**	14	7.04	13	8.33
Lauren	11	4.56	17	**4.37**	13	4.56	16	5.85
Louise	13	9.39	17	**5.70**	15	5.89	15	7.51
Lucy	8	3.12	9	**2.09**	11	2.29	9	2.96
Lydia	7	2.37	7	**2.29**	7	2.33	7	2.90
Mary	10	**2.75**	10	3.30	10	3.13	10	4.48
Paul	26	**67.73**	29	332.66	31	148.94	31	197.57
Rebecca	6	**1.82**	6	2.23	6	2.41	6	3.15
Rob, James & Matthew	18	-	43	5501.42	38	26509.42	44	**4619.90**
Ryan	15	**15.41**	15	25.64	15	22.22	15	26.84
Sally	13	8.55	18	**4.90**	16	6.32	18	7.86
Sarah	8	2.97	8	**2.60**	8	2.97	8	3.43
Tanya	5	1.54	7	**1.34**	7	1.51	7	1.55
William	15	10.09	15	**8.72**	15	9.51	15	11.55
Willow	14	15.05	16	**9.01**	16	11.85	18	14.65
Zoe & Richard	17	33.08	19	**10.78**	19	11.99	21	14.66

when this happens it allows to decrease more than one step at a time in the descending process towards unsatisfiability.

Table 3 shows the results of this algorithm on the hardest instances. We give the total times, including the bottom-up parallel process and the top-down sequential process (in other words, the difference between the times in Table 3 and the times in Table 2 corresponds to the time required by the descending process). It is worth noting that in all cases we used the *path encoding* for descending, as it was the best performing one for the sequential case. In conclusion, the recipe would be to use the *spanning tree encoding* for reachability when ascending in parallel, and the *path encoding* for descending sequentially. Note that, this way, we are able to solve "Ben & Alan" in 1561.04 seconds (compared to 4101.25 in the sequential ascending approach; see Table 2), and "Cynthia & Michael" in 539.62 seconds (compared to 1313.26 seconds). In the latter, most of the time is devoted to certify the optimal upper bound already found when ascending in parallel. As for the hardest level, "Rob, James & Matthew", we are able to obtain an upper bound of 44 in 4619.90 seconds (about an hour and a quarter), and to drop it to 32 after 8 hours. This is the only open instance left, with an optimum between 18 and 32.

A comparison with state-of-the-art planners for the sequential case can be found in [6].

▮ **Table 3** Best upper bounds of hard Snowman instances, found by first finding an upper bound in parallel (seeking for increasingly long plans; see Table 2), then serializing the plan and sequentially seeking for shorter plans. Total time in seconds (limit 8h). Best results in bold.

	LB	*path parallel*		*DAG parallel*		*tree parallel*	
		UB	time	UB	time	UB	time
Ben & Alan	28	28	2673.75	28	2044.74	**28**	**1561.04**
Cynthia & Michael	30	30	753.61	30	638.02	**30**	**539.62**
Rob, James & Matthew	18	**32**	-	36	-	**32**	-

▮ **Table 4** PAR-2 scores and number of timed out instances out of 795 Sokoban instances (time limit 1h), for each of the reachability encodings. Best results in bold.

	path parallel	*DAG parallel*	*tree parallel*
PAR-2	3,737,743	3,345,659	**3,052,851**
timeouts	497	442	**398**

5.2 Sokoban

In order to observe how the efficiency of (the replication of) the *path encoding* degenerates with the number of target locations, in this section we consider the Sokoban game. In Sokoban, the number of objects to move can be much larger than the number of snowballs in Snowman. Moreover, running a statistically significant amount of instances will allow us to better evaluate the performance of the different reachability encodings. To this purpose, we have selected 10 instance sets from a large Sokoban repository [26]. The selection was done pseudo-randomly, by discarding sets with only a small number of objects. The selected sets were *bagatelle*, *cantrip*, *cantrip2*, *chessboards*, *dh1*, *GRIGoRusha 2001*, *Sasquatch IX*, *Sharpen*, *SokoStation*, and *Tian Lang*, resulting in a total of 795 instances. Table 4 summarizes the results for the parallel approach (i.e., finding a first upper bound by ascending in parallel). The different encodings have been ranked using the PAR-2 scheme: the score of a encoding is defined as the sum of all runtimes for solved instances $+ 2 \times timeout$ for unsolved instances. Similarly to the case of Snowman, we observe that the *spanning tree encoding* is the best performing one, followed by the *DAG encoding*, and ending with the *path encoding*. We additionally checked that the *spanning tree encoding* performs better than the *DAG encoding* if we restrict to the commonly solved instances, with a total time of $105,553.14$ vs $161,063.81$ seconds, respectively.

Finally, for completeness, we also compare the performance of the planning as SAT approach with that of well-known ASP solvers on Sokoban instances of the ASP competition. We restrict to the Sokoban instances from the ASP Competition 2011, where "*the sokoban walking to a box and pushing it a certain number of locations in one direction*" was an atomic action. This was lifted in later editions and, lately, the Sokoban problem has not been included in the competition anymore. We consider *clasp* [15] (a conflict-driven answer set solving system, which took the first place in the ASP Competition 2011), *BPSolver* [29] (based on a dynamic programming approach using mode-directed tabling to store subproblems and their answers, which took the second place), and *lp2sat* [19] (based on translating an ASP program to a set of clauses such that any SAT solver can be used to calculate its answer sets). Both *clasp* and *lp2sat* are part of the Potsdam Answer Set Solving Collection [14]. For *BPSolver* we consider the same version submitted to the competition, as few changes

seem to have been made subsequently. For *clasp* we consider its current version 3.3.9. As for *lp2sat*, we use version 1.25 combined with Kissat 3.0.0 (instead of the old MiniSAT version used in the ASP Competition 2011) for a fair comparison with our approach.

We consider both decision and optimization instances from the competition. It is not easy to make a fair comparison since, in spite of ignoring walking actions, we do not consider pushing a box a certain number of locations in one direction as an atomic action, which is a disadvantage to our system. We have removed all unsatisfiable decision instances since, moreover, we seek for the shortest plan step by step, whereas solvers from the competition only had to check the instances for satisfiability, given a number of steps. Finally, we restrict to the instances solved by all the solvers, since some instances raised an error in *BPSolver*. This way, although unfair, the comparison is as fair as possible, without modifying any of the solving methods. The first row of Table 5 shows the results of this *unfair* comparison, where it can be observed that our system is competitive with *BPSolver* in spite of the disadvantage.

Table 5 PAR-2 scores (time limit 1h) for Sokoban benchmarks from the ASP Competition 2011. Kissat stands for the sequential planning as SAT approach using the *path encoding* for reachability.

	Kissat	*BPSolver*	*clasp*	*lp2sat* + Kissat
unfair	**314.34**	327.90	548.55	1216.49
fair	**3506.65**	-	39336.81	19982.19

For a more fair comparison, we modified the ASP model of the Sokoban problem from the competition so that pushing a box a certain number of locations in one direction is not an atomic action anymore, but keeping reachability, i.e., ignoring walking actions as before. Then, we ran all the ASP solvers on this modified model, by asking them for satisfiability given the number of steps found by our system. The results are summarized in the *fair* row. In this case, the superiority of kissat alone is even clearer, even though we include the total time for finding the solution step by step. We do not include the results for *BPSolver* since the version submitted to the competition had the translator for the ASP models built in, making it impossible to execute it with other models. We must also mention that we removed 6 instances for which, surprisingly, *clasp* reported an "unsatisfiable" answer, whereas all other solvers reported "satisfiable".

6   Conclusions and Further Work

We have reviewed some standard reachability encodings for graphs and presented a new one, suitable for determining connected components in undirected graphs. We have compared the performance of the different encodings on two puzzle problems using SAT and CP based approaches. Specifically, we have devised an algorithm for solving hard instances of video game puzzle problems, based on the planning as SAT paradigm. Considering the simultaneous (i.e., parallel) execution of (non-interfering) actions at locations reachable from the avatar, has turned out to be crucial to quickly find an upper bound on the number of needed actions, and to consequently solve the problem in a reasonable time. The proposed new encoding for reachability has shown to be the best performing on the hardest instances.

As seen, acyclicity is a property closely related to reachability in the context of SAT. Some works have introduced SAT and SMT solvers with support for detecting acyclicity and reachability in graphs [12, 3]. Therefore, an alternative approach could be to use a SAT or SMT solver with built-in support for reachability, such as MonoSAT [3].

―――― **References** ――――

1 Eric Allender, David A Mix Barrington, Tanmoy Chakraborty, Samir Datta, and Sambuddha Roy. Planar and grid graph reachability problems. *Theory of Computing Systems*, 45(4):675–723, 2009.

2 Tomás Balyo. Relaxing the relaxed exist-step parallel planning semantics. In *25th IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2013, Herndon, VA, USA, November 4-6, 2013*, pages 865–871. IEEE Computer Society, 2013. `doi:10.1109/ICTAI.2013.131`.

3 Sam Bayless, Noah Bayless, Holger H. Hoos, and Alan J. Hu. SAT Modulo Monotonic Theories. In Blai Bonet and Sven Koenig, editors, *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, January 25-30, 2015, Austin, Texas, USA*, pages 3702–3709. AAAI Press, 2015. URL: `http://www.aaai.org/ocs/index.php/AAAI/AAAI15/paper/view/9951`.

4 Christian Bessiere, Emmanuel Hebrard, George Katsirelos, and Toby Walsh. Reasoning about connectivity constraints. In Qiang Yang and Michael J. Wooldridge, editors, *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*, pages 2568–2574. AAAI Press, 2015. URL: `http://ijcai.org/Abstract/15/364`.

5 Armin Biere, Katalin Fazekas, Mathias Fleury, and Maximillian Heisinger. CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020. In Tomas Balyo, Nils Froleyks, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda, editors, *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions*, volume B-2020-1 of *Department of Computer Science Report Series B*, pages 51–53. University of Helsinki, 2020.

6 Miquel Bofill, Cristina Borralleras, Joan Espasa Arxer, Gerard Martin Teixidor, Gustavo A. Patow, and Mateu Villaret. A good snowman is hard to plan. In *ICAPS Workshop on Knowledge Engineering for Planning and Scheduling (KEPS 2023), Prague, Czech Republic, July 9th*, 2023.

7 Blai Bonet and Hector Geffner. Qualitative numeric planning: Reductions and complexity. *J. Artif. Intell. Res.*, 69:923–961, 2020. `doi:10.1613/jair.1.11865`.

8 Tom Bylander. Complexity results for planning. In *Proceedings of the 12th International Joint Conference on Artificial Intelligence (IJCAI)*, pages 274–279, 1991.

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

10 Diego De Uña. *Discrete optimization over graph problems*. PhD thesis, University of Melbourne, Parkville, Victoria, Australia, 2018. URL: `http://hdl.handle.net/11343/217321`.

11 Nils Froleyks, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda. SAT competition 2020. *Artificial Intelligence*, 301:103572, 2021. `doi:https://doi.org/10.1016/j.artint.2021.103572`.

12 Martin Gebser, Tomi Janhunen, and Jussi Rintanen. SAT Modulo Graphs: Acyclicity. In *Logics in Artificial Intelligence - 14th European Conference, JELIA 2014, Funchal, Madeira, Portugal, September 24-26, 2014. Proceedings*, pages 137–151, 2014.

13 Martin Gebser, Tomi Janhunen, and Jussi Rintanen. Declarative encodings of acyclicity properties. *Journal of Logic and Computation*, 30(4):923–952, 2020.

14 Martin Gebser, Benjamin Kaufmann, Roland Kaminski, Max Ostrowski, Torsten Schaub, and Marius Schneider. Potassco: The Potsdam Answer Set Solving Collection. *AI Commun.*, 24(2):107–124, 2011. `doi:10.3233/AIC-2011-0491`.

15 Martin Gebser, Benjamin Kaufmann, and Torsten Schaub. Conflict-driven answer set solving: From theory to practice. *Artif. Intell.*, 187:52–89, 2012.

16 Malik Ghallab, Dana S. Nau, and Paolo Traverso. *Automated Planning and Acting*. Cambridge University Press, 2016.

17 Weihua He, Ziwen Liu, and Chao Yang. Snowman is PSPACE-complete. *Theoretical Computer Science*, 677:31 − 40, 2017. URL: `http://www.sciencedirect.com/science/article/pii/S0304397517302050`, `doi:https://doi.org/10.1016/j.tcs.2017.03.011`.

18 Franc Ivankovic and Patrik Haslum. Optimal planning with axioms. In Qiang Yang and Michael J. Wooldridge, editors, *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*, pages 1580–1586. AAAI Press, 2015. URL: `http://ijcai.org/Abstract/15/226`.

19 Tomi Janhunen. lp2sat 1.10 — A tool for translating normal logic programs into SAT. http://www.tcs.hut.fi/Software/lp2sat/, 2006. Computer Program.

20 Henry A. Kautz and Bart Selman. Planning as satisfiability. In *10th European Conference on Artificial Intelligence, ECAI 92, Vienna, Austria, August 3-7, 1992. Proceedings*, pages 359–363, 1992.

21 Yuliya Lierler. What is answer set programming to propositional satisfiability. *Constraints An Int. J.*, 22(3):307–337, 2017. `doi:10.1007/s10601-016-9257-7`.

22 Shuwa Miura and Alex Fukunaga. 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.*, pages 218–227, 2017.

23 Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. Minizinc: Towards a standard CP modelling language. In Christian Bessiere, editor, *Principles and Practice of Constraint Programming - CP 2007, 13th International Conference, CP 2007, Providence, RI, USA, September 23-27, 2007, Proceedings*, volume 4741 of *Lecture Notes in Computer Science*, pages 529–543. Springer, 2007. `doi:10.1007/978-3-540-74970-7\_38`.

24 Luis Quesada. *Solving constrained graph problems using reachability constraints based on transitive closure and dominators.* PhD thesis, Catholic University of Louvain, Louvain-la-Neuve, Belgium, 2006.

25 Jussi Rintanen, Keijo Heljanko, and Ilkka Niemelä. Planning as Satisfiability: Parallel Plans and Algorithms for Plan Search. *Artificial Intelligence*, 170(12-13):1031–1080, 2006.

26 Sokoban. Sokoban Repository. `http://sokobano.de/en/levels.php`, 2023. [Online; accessed 25-April-2023].

27 V. S. Subrahmanian and Carlo Zaniolo. Relating stable models and AI planning domains. In Leon Sterling, editor, *Logic Programming, Proceedings of the Twelfth International Conference on Logic Programming, Tokyo, Japan, June 13-16, 1995*, pages 233–247. MIT Press, 1995.

28 Martin Wehrle and Jussi Rintanen. Planning as Satisfiability with Relaxed Exists-Step Plans. In *AI 2007: Advances in Artificial Intelligence, 20th Australian Joint Conference on Artificial Intelligence, Gold Coast, Australia, December 2-6, 2007, Proceedings*, pages 244–253, 2007. `doi:{10.1007/978-3-540-76928-6\_26}`.

29 Neng-Fa Zhou and Agostino Dovier. A Tabled Prolog Program for Solving Sokoban. *Fundam. Informaticae*, 124(4):561–575, 2013. `doi:10.3233/FI-2013-849`.