A Numeric PDDL Representation of the Good Snowman Planning Problem

Matteo Paparo[1], Francesco Doria[1], Marco Maratea[1] and Mauro Vallati[2]

[1]DeMaCS, University of Calabria, Italy
[2]School of Computing and Engineering, University of Huddersfield, UK

Abstract

The single-agent puzzle game Good Snowman aims to navigate a confined, maze-like environment with the specific goal of constructing snowmen by stacking three snowballs in descending order of size, and can be naturally characterised as a planning problem, where it is 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. A solution to the planning problem has been provided via classical planning; however, the solution does not scale properly.

Motivated by the fact that parts of the problem can be conveniently represented using numeric predicates/fluents, in this paper we present a numeric PDDL representation of the Good Snowman planning problem. Results of running ENHSP as planning engine show that the numeric representation leads to computational advantages over the classical formulation. Further, we define, implement and test a domain heuristic tailored for this problem, that leads to further improvements.

Keywords

Automated Planning, Numeric PDDL2.1 Representation, Application

1. Introduction

In 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 solution that is much easier than the ones the designer had initially in mind [1]. These unplanned solutions 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 (optimal) solutions for given scenarios of the game, so that guarantees can be provided in terms of the expected complexity of a level.

In this work we focus on a discrete time and space puzzle. In particular, we consider the PSPACE-complete game *A Good Snowman is Hard to Build*[1] [2], where a character has to generate and move snowballs to build a required number of snowmen. The game has some resemblance with the widely studied Sokoban 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.

CILC 2026: The 41st Italian Conference on Computational Logic, June 23–25, 2026, Ferrara, Italy
*Corresponding author.
matteo.paparo@hotmail.com (M. Paparo); doria.francesco27.fd@gmail.com (F. Doria); marco.maratea@unical.it (M. Maratea); M.Vallati@hud.ac.uk (M. Vallati)
0000-0002-9034-2527 (M. Maratea); 0000-0002-8429-3570 (M. Vallati)

[1]https://agoodsnowman.com/.

A Good Snowman is Hard to Build (A Good Snowan, for short, hereafter) can be naturally characterised as a planning problem, where it 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. Ivankovic and Haslum [3] show how a reachability derived predicate can be specified with axioms, so that 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 [4]. This is done with success for Sokoban and reachability, as shown by using axioms supporting model-based planners with Integer Programming and Answer Set Programming technologies. Recently, a solution to the planning problem has been provided via classical planning [5]; however, while the encoding is elegant and compact, it shows some limitations in terms of the ability to solve larger problems.

In this paper, we present a numeric PDDL representation of the Good Snowman planning problem. The main idea is to leverage on conditional effects to model the range of modifications that actions can apply to the environment, while maintaining the encoding compact. The numeric aspects also support a more readable and concise encoding of the constraints. Results of the resulting PDDL2.1 model coupled with the ENHSP planning engine show that the numeric representation leads to computational advantages over the classical formulation running both ENHSP and the best planner in the analysis of [5], namely SymK . Then, with the goal of further improving scalability, we define, implement and test a domain heuristic tailored for this problem: the idea of this heuristic is to prioritize states that are closer to the goal, by estimating the minimum number of pushes required to build a snowman given the available balls, which led to further improvements.

The remainder of this paper is structured as follows. Section 2 introduces needed preliminaries about numeric planning. Then, the Good Snowman problem is described in more details in Section 3, whose PDDL2.1 numeric model is presented in Section 4. Section 5, instead, presents the domain-specific heuristic, whose results, together with those of the PDDL2.1 numeric model, are showed in Section 6. The paper ends in Section 7 by mentioning alternative numeric planners and by drawing some conclusion and possible topics for future research.

2. Background on Numeric Planning

Here we focus on numeric planning problems [6] as those that can be expressed in PDDL2.1 level 2 [7, 8]. Without loss of generality, we restrict our attention to the case with untyped objects, and with only numeric fluents[2] (see below).

A numeric planning problem consists of two elements: a domain model and a problem model. Following the PDDL terminology, the domain model contains the definition of the predicates, the numeric fluents, and a set of actions. In particular, numeric fluents indicate properties of lists of objects; mathematically, they define mappings between lists of objects to numeric values. The domain model defines them in an abstract way: it specifies a name, a string label for each such mapping, and a list of variables. Variables specify the order and the number of objects to be mapped.

An action a is defined by means of a name (which we will often omit in the interest of space), a list of variables (called the parameters of the actions), a precondition formula (i.e., $pre(a)$) and a set of effects (i.e., $eff(a)$). The precondition formula is a first-order logic proposition having equalities or inequalities involving numeric fluents as terms (e.g., `(> (battery ?r1) 4)` $\wedge$ `(> (battery ?r2) 5)`). Each formula can make use of the standard logical connectives from propositional logic, i.e., $\wedge, \vee, \neg$, together with arbitrary nesting of universal ($\forall$) and existential ($\exists$) quantifier over the objects of the

[2] A Boolean fluent can be mapped into a $\{0, 1\}$ numeric fluent.

problem. Effects are triplets of the form $\langle\{increase, decrease, assign\}, x, \xi\rangle$, where the first term is the modifier, and can either be an increase (i.e., *increase*), a decrease (i.e., *decrease*) or an assignment (i.e., *assign*), the second term is a numeric fluent, and the third term is a numeric expression that together with the modifier determines the state of the numeric fluent if the action is applied.

Each numeric fluent in the action structure can have its parameters expressed as concrete objects (i.e., actual objects of the problem to be solved) or variables. When all parameters are concrete objects, a numeric fluent is said to be ground. Similarly, an action with all parameters and free variables substituted with concrete objects is said to be ground. This also requires to eliminate all quantifiers in the preconditions using standard quantifier elimination techniques. In this work we focus on actions whose effects can increase, decrease or assign the value to a numeric fluent by means of a constant (e.g., `(increase (battery ?r1) 5.4)`).

A domain model is a tuple $\langle\mathcal{X}, A\rangle$ where $\mathcal{X}$ is the numeric fluents set as above, and A the set of actions. Let $\mathcal{O}$ be a set of objects and x a numeric fluent from $\mathcal{X}$. The grounding of x is the set of numeric fluents each having the same name of x but the list of variables replaced with concrete objects from some subset of $\mathcal{O}$. The set of ground numeric fluents given $\mathcal{O}$ is denoted by $\mathcal{X}[\mathcal{O}]$. Finally, we use $abs(x)$ to denote the abstraction of an object x into a variable, i.e., the ungrounded version of the numeric fluent.

A state s gives a value to each numeric fluent in $\mathcal{X}[\mathcal{O}]$. The domain of each numeric fluent is the set of rational number plus the special term $\perp$; $\perp$ is used to state that a given numeric fluent is undefined. Let $x \in \mathcal{X}$ and s be a state, we denote with $[x]_s$ the value of numeric fluent x in state s. Then, we use $succ(s)$ for the set of states reachable by s through actions from A.

A ground action is applicable in state s iff its precondition is satisfied in s. A precondition is satisfied iff, by assigning all numeric fluents their values as for state s, the evaluation of the formula returns true. The application of a ground action in a state s generates a new state $s' = s[a]$ such that all numeric fluents conform with the effects of the action. For instance, if an action features a numeric effect $\langle increase, x, 1\rangle$ and the state is such that $x = 1$, then the successor state will be such that $x = 2$.

A problem model is given by a set of objects, a state, called the initial state, and a goal. The goal is structured as the precondition of an action, with the difference that any component which is not quantified only involves ground numeric fluents. A problem model is formally expressed as a tuple $\langle\mathcal{O}, I, G\rangle$. The combination of a domain and a problem instance is a planning problem $\mathcal{P} = \langle D, P\rangle$. A plan for a planning problem is a sequence of actions τ such that τ can be iteratively applied starting from the initial state I, and the last produced state is one where the goal G is satisfied.

3. The Good Snowman Problem

The single-agent puzzle game, A Good Snowman, challenges players to navigate a confined, maze-like environment with the specific goal of constructing snowmen by stacking three snowballs in descending order of size. The game world is composed of a grid of playable cells, some of which are bare while others are covered in a layer of fresh snow. Within this space, the player controls a small black character whose only means of interaction is moving in the four cardinal directions. Success depends entirely on the player's ability to manipulate the snowballs, which are initially scattered across the map in various sizes categorized as small, medium, and large.

Movement and interaction follow a strict set of logical rules reminiscent of the classic game Sokoban, meaning the character can only push objects and never pull them. When the agent moves into a vacant cell, it simply occupies that space. However, when the agent moves toward a cell occupied by a single snowball, a roll occurs, provided the space on the opposite side of the ball is empty. If a ball is rolled onto a cell covered in snow, it absorbs that snow and grows one size larger, such as a small ball becoming medium or a medium ball becoming large. Once a ball reaches the large size, it cannot grow further, yet rolling it over snowy tiles will still clear the snow from the ground, effectively consuming a resource that might be needed for other balls.

The stacking mechanics introduce a vertical dimension to the puzzle logic. An agent can perform a

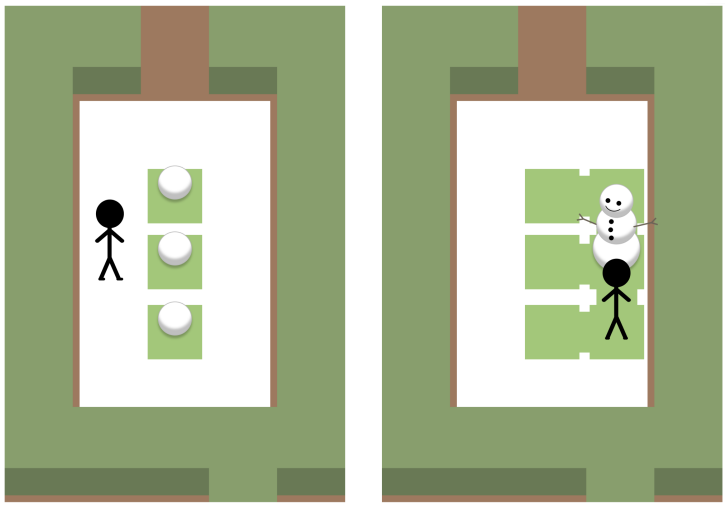

Figure 1: Tanya level, illustrating the execution of the optimal solution sequence `dRdrUlluuRurDlldR`. Each character denotes an atomic movement in a cardinal direction, where uppercase characters represent snowball pushes and lowercase characters denote agent walk moves (r/R=right, l/L=left, u/U=up, d/D=down).

push to move a snowball onto an existing stack only if the ball being pushed is smaller than the one currently at the top of the stack. This action successfully creates or adds to a pile and allows the agent to move into the ball's previous coordinates. If the player needs to disassemble a pile, they can attempt to move into a cell containing a stack to trigger a pop action. This manoeuvre ejects the topmost ball into the adjacent empty cell without changing the agent's own position, though it can only be performed if the landing spot is entirely unoccupied.

The ultimate objective of each level is to utilize every available snowball to form complete snowmen, each consisting of a large base, a medium torso, and a small head. Depending on the specific scenario, the player may need to build one, two, or three snowmen using sets of three, six, or nine snowballs, respectively.

Because snowmen can be built on any playable cell, the player must carefully plan their route to ensure they do not accidentally grow a snowball too large too soon or trap a ball against a wall where it can no longer be pushed.

The problem can be 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.

Figure 1 illustrates an example of how to solve one of the levels of the game, where the left side shows the initial state with the character and three snowballs of size one on a grid with some locations covered in snow, and the right side shows the final state with the snowballs joined into a snowman.

4. PDDL2.1 Numeric Model

Our numeric model modifies the PDDL model introduced in [5] by replacing its predicate-based size modelling with a single numeric fluent, `(ball_size ?b)`. In the original domain, ball size was tracked using three mutually exclusive predicates (`(ball_size_small ?b)`, `(ball_size_medium ?b)`, `(ball_size_large ?b)`), and stacking rules required enumerating all valid combinations explicitly. The numeric version simplifies this by expressing stacking rules as numeric comparisons. Rolling over snow is also simpler, as the model just increments `(ball_size ?b)` instead of updating the predicates. Similarly, the goal condition checks that three balls of sizes one, two and three occupy the same location, rather than checking all valid combinations. We now describe the model in detail.

The domain defines three types: `location`, `direction` and `ball`. A location represents a grid cell that may contain the character, a stack of balls, or snow. A direction enumerates the four directions (up,

Listing 1: The operator used to move the character in the environment.

```
(:action move-character
 :parameters (?from ?to - location ?dir - direction)
 :precondition (and
        (next ?from ?to ?dir)
        (character_at ?from)
        (not (occupancy ?to))
        (not (snowman_at ?to)))
 :effect (and
        (not (character_at ?from))
        (character_at ?to)))
```

down, left, right) in which the character and balls can move. An object ball is used to identify each ball in the model.

The following predicates are defined:

- `(snow ?l - location)`: true if location `?l` is covered in snow.
- `(next ?from ?to - location ?dir - direction)`: true if location `?from` is adjacent to location `?to` in direction `?dir`.
- `(occupancy ?l - location)`: true if location `?l` contains at least one ball.
- `(character_at ?l - location)`: true if the character is at location `?l`.
- `(ball_at ?b - ball ?l - location)`: true if ball `?b` is at location `?l`.
- `(snowman_at ?l - location)`: true if location `?l` contains a snowman.
- `(ball_used_in_snowman ?b - ball)`: true if ball `?b` is part of a snowman.

Three numeric functions are defined:

- `(ball_size ?b - ball)`: current size of ball `?b`.
- `(total-cost)`: cumulative cost of `move_ball` actions.
- `(snowman_built)`: number of snowmen built.

Three actions are defined: `move_character` (Listing 1), which allows the character to navigate between adjacent free locations; `move_ball` (Listing 2), which enables the character to push balls across the grid, including stacking them or increasing their size when moving over snow; and `build_snowman` (Listing 3), which combines three balls of the required size at the same location into a snowman.

The `move_character` action takes a source location `?from`, a target location `?to` and a direction `?dir`. It requires that `?to` is adjacent to `?from` in direction `?dir`, character is at `?from` and `?to` is not occupied by any ball or snowman. When these conditions are met, it updates the character position from `?from` to `?to`.

The `move_ball` action takes a ball `?b`, the character position `?pos`, the ball current location `?from` and destination `?to`, and a direction `?dir`. It requires that `?pos`, `?from` and `?to` are consecutive locations in direction `?dir`, character is at `?pos`, location `?to` is not occupied by a snowman, ball `?b` is at `?from` and is not part of a snowman. Three additional constraints enforce the stacking and movement rules. First, for every other ball `?o` in the domain, one of the following must hold: `?o` is the ball being pushed, `?o` is not at `?from`, `?o` is at `?from` and is larger than `?b`. This ensures that `?b` is at the top of the stack at `?from` and is therefore the one that can be pushed (lines 11-15). Second, either `?b` is the only ball at `?from`, or `?to` is empty, preventing two stacks from merging (lines 17-23). Third, any ball already at `?to` must be larger than `?b`, ensuring a valid stacking order at destination (lines 24-27). When these conditions are met, the action moves `?b` from `?from` to `?to`, marking `?to` as occupied, and increases `total-cost` by one (lines 29-32). If `?b` is the only ball at `?from` it also moves the character from `?pos` to `?from`, marking `?from` as unoccupied (lines 34-41). If `?to` is covered in snow and `?b` has not yet reached its maximum size, it increases `?b`'s size by one and removes the snow at `?to` (lines 43-48).

The `build_snowman` action takes three balls `?b_p`, `?b_m`, `?b_g` and a location `?l`. It requires that all three balls are distinct, co-located at `?l`, have sizes one, two and three, respectively, are not already

Listing 2: The operator used to move the ball in the environment.

```
(:action move-ball
     :parameters (?b - ball ?pos ?from ?to - location ?dir - direction)
     :precondition (and
            (next ?pos ?from ?dir)
            (next ?from ?to ?dir)
            (character_at ?pos)
            (not(snowman_at ?to))
            (ball_at ?b ?from)
            (not (ball_used_in_snowman ?b))

            (forall (?o - ball)
                (or
                    (= ?o ?b)
                    (not (ball_at ?o ?from))
                    (< (ball_size ?b) (ball_size ?o))))

            (or
                (forall (?o - ball)
                    (or
                        (= ?o ?b)
                        (not (ball_at ?o ?from))))
                (forall (?o - ball)
                        (not (ball_at ?o ?to))))
            (forall (?o - ball)
                (or
                    (not (ball_at ?o ?to))
                    (< (ball_size ?b) (ball_size ?o)))))
     :effect (and
            (occupancy ?to)
            (not (ball_at ?b ?from))
            (ball_at ?b ?to)
            (increase (total-cost) 1)

            (when (forall (?o - ball)
                    (or
                        (= ?o ?b)
                        (not (ball_at ?o ?from))))
                (and
                    (not (character_at ?pos))
                    (character_at ?from)
                    (not (occupancy ?from))))

            (when (and
                    (snow ?to)
                    (< (ball_size ?b) 3))
                (and
                    (increase (ball_size ?b) 1)
                    (not (snow ?to))))))
```

part of a snowman, and that `?l` is free of snowmen. When these conditions are met, it increments the snowman count by one, places a snowman at `?l` and marks all three balls as part of a snowman.

Listing 4 encodes an extract of the problem instance of Figure 1. The initial state corresponds to the left side of the figure, where the character is at position `loc_1_3` and three snowballs of size 1 are located at `loc_2_2`, `loc_2_3` and `loc_2_4`. The other locations are covered in snow, enabling the snowballs to reach the size needed to build a snowman. The goal, shown on the right, requires building a snowman and is expressed as `(=(snowman_built) 1)`.

Listing 3: The operator used to build a snowman.

```
(:action build_snowman
 :parameters (?b_p ?b_m ?b_g - ball ?l - location)
 :precondition (and
      (not (= ?b_p ?b_m))
      (not (= ?b_p ?b_g))
      (not (= ?b_m ?b_g))

      (ball_at ?b_p ?l)
      (ball_at ?b_m ?l)
      (ball_at ?b_g ?l)

      (= (ball_size ?b_p) 1)
      (= (ball_size ?b_m) 2)
      (= (ball_size ?b_g) 3)

      (not (ball_used_in_snowman ?b_p))
      (not (ball_used_in_snowman ?b_m))
      (not (ball_used_in_snowman ?b_g))

      (not (snowman_at ?l)))

 :effect (and
      (increase (snowman_built) 1)
      (snowman_at ?l)
      (ball_used_in_snowman ?b_p)
      (ball_used_in_snowman ?b_m)
      (ball_used_in_snowman ?b_g)))
```

5. Domain-specific Heuristic

This section introduces $h^{snowman}$, a domain-specific heuristic for the snowman-building problem. The heuristic estimates the minimum number of ball pushes required to reach the goal.

To avoid expensive recomputation at search time, the heuristic relies on several structures precomputed from the static map: the minimum pushes required to move a ball between any two cells, computed via a 0-1 Breadth-First Search (BFS) that assigns cost 0 to walk moves and cost 1 to push moves; the all-pairs shortest path distances via the Floyd–Warshall algorithm, used to determine character reachability; the set of dead-end and corner cells, used to detect unsolvable configurations; and the set of all possible ball triples, used to evaluate every candidate snowman assignment.

The heuristic function takes as input a state σ and a goal condition G. It begins by extracting the set of active balls B (those not yet used in a snowman) and target locations T (those not yet occupied by a snowman) from σ (lines 2-3). It then verifies whether a solution can be reached from the current state, returning ∞ if not (lines 7-9), by checking five conditions: first, that there is enough snow to grow each active ball to its target size; second, that each active ball can reach at least one snow cell; third, that enough balls are free to be pushed — i.e., not in dead-end or corner cells — to complete all remaining snowmen; fourth, that at least one active ball can be pushed from the character's position; fifth, that at least one ball is reachable from the character's position, ignoring dynamic obstacles.

Next, the heuristic iterates over all possible triples of active balls (line 10), and initializes the cost of each one to ∞ (line 11). It then iterates over all combinations of a target location $t \in T$ and a permutation $\rho \in Perm(s)$ of the triple's balls, where $Perm(s)$ denotes the set of all permutations of the balls in s (line 12). The permutation ρ is unpacked into (b_A, b_B, b_C, t), where b_A represents the base (requires size three), b_B represents the chest (requires size two) and b_C represents the head (line 13). The cost of the triple is then updated to the minimum between its current value and the sum of three terms: $\textsc{JointSnow}(b_A, b_B, t, \sigma)$, $\textsc{CleanDist}(b_C, t, \sigma)$ and $\textsc{PopPenalty}(\rho, t, \sigma)$, described in the following:

- $\textsc{JointSnow}(b_A, b_B, t, \sigma)$ computes the minimum joint cost of moving b_A and b_B to t, accounting for the snow cells each must traverse to reach its required size — b_A must grow to size three, b_B

Listing 4: Encoding of the problem instance illustrated in Figure 1

```
(define (problem tanya)

  (:objects
      right - direction left - direction up - direction down - direction

      ball_0 - ball ball_1 - ball ball_2 - ball

      loc_1_1 - location loc_1_2 - location loc_1_3 - location
      loc_1_4 - location loc_1_5 - location loc_2_1 - location
      loc_2_2 - location loc_2_3 - location loc_2_4 - location
      loc_2_5 - location loc_3_1 - location loc_3_2 - location
      loc_3_3 - location loc_3_4 - location loc_3_5 - location
  )

  (:init
      (= (total-cost) 0)
      (= (snowman_built) 0)

      (character_at loc_1_3)

      (ball_at ball_0 loc_2_2) (=(ball_size ball_0) 1)
      (ball_at ball_1 loc_2_3) (=(ball_size ball_1) 1)
      (ball_at ball_2 loc_2_4) (=(ball_size ball_2) 1)

      (snow loc_1_1) (snow loc_1_2) (snow loc_1_3)
      (snow loc_1_4) (snow loc_1_5) (snow loc_2_1)
      (snow loc_2_5) (snow loc_3_1) (snow loc_3_2)
      (snow loc_3_3) (snow loc_3_4) (snow loc_3_5)

      (occupancy loc_2_2) (occupancy loc_2_3) (occupancy loc_2_4)
  )

  (:goal
      (=(snowman_built) 1)
  )

  (:metric minimize (total-cost))
)
```

to size two. The snow cells assigned to each are constrained to be disjoint, preventing both balls from consuming the same ones.

- CleanDist(b_C, t, σ) computes the minimum cost of moving b_C to t along a path that avoids all snow cells, since b_C must not increase in size.
- PopPenalty(ρ, t, σ) adds a penalty of +2 for each non-base ball already occupying t, since at least one push is required to move the ball out of t and one to move it back in the correct position.

When more than one snowman must be built ($|B| > 3$), the heuristic returns the minimum combined cost across all partitions of B into disjoint triples, where $Part(B)$ denotes the set of all such partitions (lines 17-19). Otherwise, since there are exactly three active balls, only one triple exists, and its minimum cost is returned directly (line 20).

5.1. Heuristic Admissibility

After having described the heuristic, we discuss its admissibility.

Given a state σ, we remind that a heuristic $h(\sigma)$ is admissible if it never overestimates the optimal cost to the goal, satisfying $h(\sigma) \leq h^*(\sigma)$ for every state σ, where $h^*(\sigma)$ denotes, in our case, the number of ball pushes required to complete all remaining snowmen. Informally, $h^{snowman}(\sigma)$ computes, for each

triple $s \in \binom{B}{3}$ of active balls, the minimum cost $c(s)$ over all role assignments $\rho \in Perm(s)$ and target locations $t \in T$ of the sum of JointSnow(b_A, b_B, t, σ), CleanDist(b_C, t, σ), and PopPenalty(ρ, t, σ). Each component is a lower bound on a disjoint subset of the required pushes. JointSnow and CleanDist count pushes involving $\{b_A, b_B\}$ and $\{b_C\}$, respectively, so no push is double-counted between them. Both use the shortest path on an obstacle-free map, which relaxes the original problem, ensuring their costs never exceed the true cost. PopPenalty accounts for a cost neither component captures: if a non-base ball (b_B, b_C) occupies t, it must first be pushed away to place the base, then pushed back to complete the snowman, contributing at least two additional pushes. Since each component underestimates its respective push cost and covers a disjoint subset of required pushes, their sum $c(s)$ is a lower bound on the total cost for s. When $|B| = 3$, the heuristic returns $c(s)$, which is therefore a lower bound. When $|B| > 3$, it returns the minimum of $\sum_{s \in \Pi} c(s)$ over all partitions $\Pi \in Part(B)$. Since each $c(s)$ is a lower bound and the triples are disjoint, their sum is also a lower bound. For the base cases, the heuristic returns 0 at goal states and ∞ at dead ends. Both match h^*: a goal state requires no further actions so $h^* = 0$, and a dead end has no solution so $h^* = \infty$. Therefore, $h^{snowman}(\sigma) \leq h^*(\sigma)$ for any state σ, making the heuristic admissible.

Algorithm 1: Heuristic $h^{snowman}$ computation for a given state σ and a goal condition G.

```
1:  function Heuristic(σ,G)
2:      B ← ActiveSnowballs(σ)
3:      T ← Locations(σ)
4:      if IsGoal(σ,G) then
5:          return 0
6:      end if
7:      if DeadEnd(σ) then
8:          return +∞
9:      end if
10:     for each s ∈ (B choose 3) do
11:         c(s) ← ∞
12:         for each (t, ρ) ∈ T × Perm(s) do
13:             (b_A, b_B, b_C) ← ρ
14:             c(s) ← min(c(s), JointSnow(b_A, b_B, t, σ)+ CleanDist(b_C, t, σ)
                + PopPenalty(ρ, t, σ))
15:         end for
16:     end for
17:     if |B| > 3 then
18:         return min{ Σ_{s∈Π} c(s) : Π ∈ Part(B) }
19:     end if
20:     return min{c(s) : s ∈ (B choose 3)}
21: end function
```

6. Experiments

We evaluate the proposed numeric model and heuristic on the benchmark introduced by Bofill et al. [5], consisting of 51 instances ranging from 1 to 3 snowmen and from 14 to 99 valid locations. Let us remind that the final position of the snowmen is not defined in the goal, but is constrained by the snow available and by the locations. All experiments were run once on a machine equipped with an Apple M3 chip (8-core CPU and 8-core GPU) running at 4.06 GHz and with 16 GB of RAM. A maximum timeout of 5 minutes (300 seconds) has been used.

We assess and compare the performance of four planning engines: basic-ENHSP uses the original classical encoding under Greedy Best-First Search with ENHSP-25 [8, 9]; basic-SymK uses the same

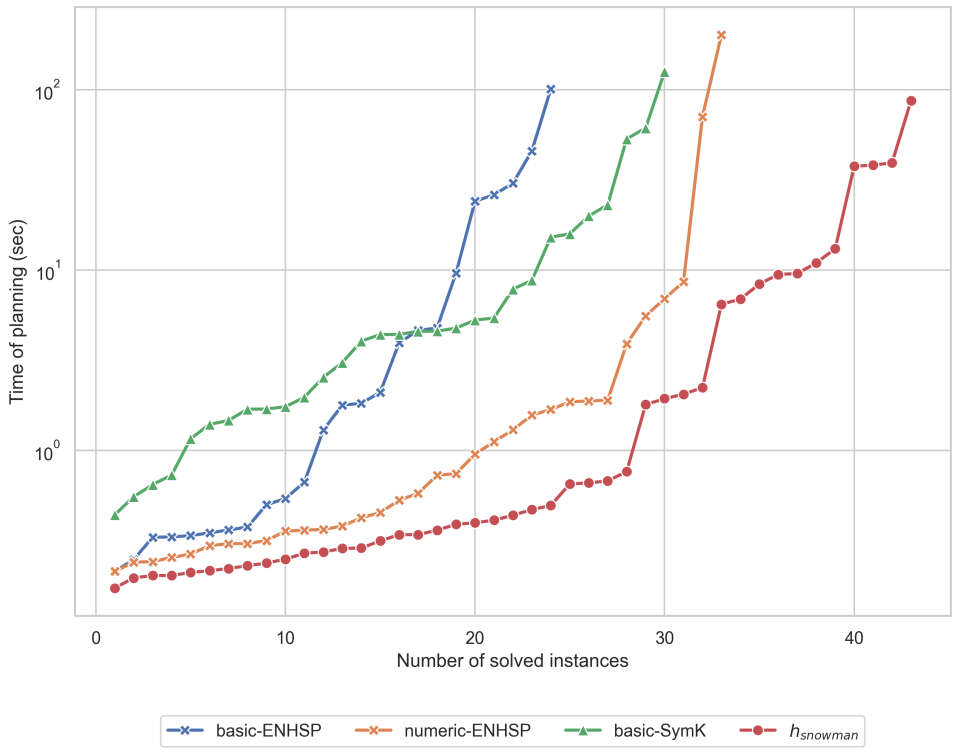

Figure 2: Cumulative number of solved instances for all the considered approaches, over time. Time is provided in CPU-time seconds, using a logarithmic scale.

encoding under Fast Downward v22.12 [10] and SymK [11] v3.0 bidirectional search, replicating the state-of-the-art configuration from [5]; numeric-ENHSP uses the numeric encoding under the same search; $h^{snowman}$-ENHSP pairs the numeric encoding with A^* and $h^{snowman}$, which has been implemented in ENHSP.

	Solved	**PAR-2**
basic-SymK	30	14.187
basic-ENHSP	24	17.661
numeric-ENHSP	33	12.317
$h^{snowman}$-ENHSP	43	6.285

Table 1
Number of solved instances, and PAR-2 score (the score of a solver is defined as the sum of all runtimes of solved instances + 2 × timeout for unsolved instances) for each approach.

Table 1 summarises, for each approach, the number of solved instances and PAR-2 (Penalised Average Runtime) score. The score, widely used in the algorithm configuration literature as well as in [5], allows to combine in a single metrics coverage and average runtime. In a nutshell, PAR-2 is calculated as the average of runtimes, where unsolved instances are assigned a value of twice the max timeout ($300 \times 2 = 600$ seconds in our case). First, focusing on the classical planning encoding, it is easy to notice that basic-SymK solves 6 more instances than basic-ENHSP. This is unsurprising, to some extent, given that ENHSP has been originally designed for numeric problems. Further, it is interesting to note that the numeric encoding appears to be capable of increasing the number of problems that can be solved by means of automated planning approaches. More specifically, the numeric encoding allows numeric-ENHSP to solve 3 more instances than basic-SymK, and 9 more instances than basic-ENHSP. However, the best results come from $h^{snowman}$-ENHSP, which solves 43 instances — 10 more than

numeric-ENHSP — at half the PAR-2 score (6.285). Figure 2 shows the cumulative number of solved instances over time for all four approaches. As expected given the discussed PAR-2 scores, instances are solved very quickly when the proposed heuristics is exploited, with the domain-independent approaches requiring a bit more time also for the simplest instances.

Overall, the experimental analysis confirms that the proposed numeric encoding is useful to increase the coverage of domain-independent planning engines. On top of that, the introduced domain-specific heuristic can provide a further boost, extending by 30% the coverage when compared to he best classical planning approach.

7. Related Work and Conclusion

Numeric planning problems have been traditionally addressed using heuristic approaches, usually by extending classical planning techniques to consider the numeric aspects. Well known examples include Metric-FF [12] and LPG [13], and more recent approaches leverage on interval, LM-cut, or subgoaling relaxations to guide efficient satisficing and optimal search, such as ENHSP and NLM_CutPlan [9, 14, 15]. A complementary line of work focuses on the translation of numeric planning problems into satisfiability or optimisation problems, as in the case of PATTY [16]. However, a recent edition of the International Planning Competition, highlighted the limited number of solvers available to the community for this class of problems [17]. An orthogonal class of approaches investigates the use of reformulation techniques, such as macro actions, to increase the efficiency planning engines on numeric domain models [18, 19]. For what concern the applications, differently for other PDDL levels and extensions such as classical, temporal and mixed-discrete continuos planning where real-wolrd applications have been defined (see, e.g., [20, 21, 22, 23]), there are few or no applications of numeric planning.

In this paper we introduced a numeric PDDL representation of the Good Snowman problem, whose results running ENHSP as planning engine show that the numeric representation leads to computational advantages over the classical formulation. Further, we have defined, implemented and tested a domain heuristic tailored for this problem, that leads to further improvements, confirming the validity of the proposed encoding.

We see several avenues for future work. First, we plan to extend the empirical evaluation to other numeric planning engines, and to include the SAT-based formulations of the problem available in [5]. Second, we would like to adapt the heuristic to the classical formulation. Finally, we are interested in investigating the use of reformulation techniques, such as numeric macro actions, to improve scalability and overall performance.

Declaration on Generative AI

The authors have not employed any Generative AI tools.

References

[1] E. Silli, Mirror's edge - level design challenges & solutions, in: GDC Europe, 2010.

[2] Z. He, W.; Liu, C. Yang, Snowman is pspace-complete, Theoretical Computer Science 677 (2017) 31–40.

[3] F. Ivankovic, P. Haslum, Optimal planning with axioms, in: Q. Yang, M. J. Wooldridge (Eds.), Proc. of IJCAI, AAAI Press, 2015, pp. 1580–1586.

[4] S. Miura, A. Fukunaga, Automatic extraction of axioms for planning, in: Proc. of ICAPS, AAAI Press, 2017, pp. 218–227.

[5] M. Bofill, C. Borralleras, J. Espasa, G. Martin, G. Patow, M. Villaret, A good snowman is hard to plan, in: Proc. of ICAPS 2023 KEPS workshop, 2023.

[6] M. Ghallab, D. S. Nau, P. Traverso, Automated Planning and Acting, Cambridge University Press, 2016.

[7] M. Fox, D. Long, PDDL2.1: an extension to PDDL for expressing temporal planning domains, Journal of Artificial Intelligence Research 20 (2003) 61–124.
[8] E. Scala, P. Haslum, S. Thiébaux, M. Ramírez, Subgoaling techniques for satisficing and optimal numeric planning, Journal of Artificial Intelligence Research 68 (2020) 691–752.
[9] E. Scala, P. Haslum, S. Thiebaux, M. Ramirez, Interval-based relaxation for general numeric planning, in: Proc. of ECAI, 2016, pp. 655–663.
[10] M. Helmert, The fast downward planning system, Journal of Artificial Intelligence Research 26 (2006).
[11] D. Speck, F. Geißer, R. Mattmüller, Á. Torralba, Symbolic Planning with Axioms, in: N. Lipovetzky, E. Onaindia, D. E. Smith (Eds.), Proc. of ICAPS, 2019, pp. 464–472.
[12] J. Hoffmann, The metric-ff planning system: Translating"ignoring delete lists"to numeric state variables, Journal of Artificial Intelligence Research 20 (2003) 291–341.
[13] A. E. Gerevini, A. Saetti, I. Serina, Planning through stochastic local search and temporal action graphs in lpg, Journal of Artificial Intelligence Research 20 (2003) 239–290.
[14] E. Scala, P. Haslum, S. Thiébaux, M. Ramirez, Subgoaling techniques for satisficing and optimal numeric planning, Journal of Artificial Intelligence Research 68 (2020) 691–752.
[15] R. Kuroiwa, A. Shleyfman, C. Piacentini, M. P. Castro, J. C. Beck, The lm-cut heuristic family for optimal numeric planning with simple conditions, Journal of Artificial Intelligence Research 75 (2022) 1477–1548.
[16] M. Cardellini, E. Giunchiglia, M. Maratea, Symbolic pattern planning, Artificial Intelligence (2026) 104482.
[17] A. Taitler, R. Alford, J. Espasa, G. Behnke, D. Fišer, M. Gimelfarb, F. Pommerening, S. Sanner, E. Scala, D. Schreiber, J. Segovia-Aguas, J. Seipp, The 2023 international planning competition, AI Magazine 45 (2024) 280–296.
[18] L. Chrpa, E. Scala, M. Vallati, Towards a reformulation based approach for efficient numeric planning: Numeric outer entanglements, in: Proc. of SOCS, 2015, pp. 166–170.
[19] D. Alarnaouti, F. Percassi, M. Vallati, An extensive empirical analysis of macro-actions for numeric planning, in: Proc. of AIxIA, 2024, pp. 23–36.
[20] F. Doria, F. Percassi, M. Maratea, M. Vallati, A domain-specific heuristic for PDDL+-based traffic signal optimisation, in: S. Koenig, C. Jenkins, M. E. Taylor (Eds.), Proc. of AAAI, AAAI Press, 2026, pp. 36207–36216.
[21] S. Fiorentino, C. Dodaro, M. Maratea, M. Vallati, AI-enabled connected autonomous vehicles sustainable routing in urban areas, in: Prof. of ITSC, IEEE, 2025, pp. 1819–1824.
[22] E. S. Barjuei, A. Capitanelli, R. Bertolucci, E. Courteille, F. Mastrogiovanni, M. Maratea, Digital workflow for printability checking and prefabrication in robotic construction 3d printing based on artificial intelligence planning, Engineering Applications of Artificial Intelligence 133 (2024) 108254.
[23] M. Cardellini, M. Maratea, M. Vallati, G. Boleto, L. Oneto, In-station train dispatching: A PDDL+ planning approach, in: S. Biundo, M. Do, R. Goldman, M. Katz, Q. Yang, H. H. Zhuo (Eds.), Proc. of ICAPS, AAAI Press, 2021, pp. 450–458.
[24] M. Cardellini, E. Giunchiglia, M. Maratea, Symbolic numeric planning with patterns, in: M. J. Wooldridge, J. G. Dy, S. Natarajan (Eds.), Proc. of AAAI, AAAI Press, 2024, pp. 20070–20077.