A Good Snowman Is (also Numerically) Hard to Plan

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

[1]DeMaCS, University of Calabria, Italy
[2]School of Computing and Engineering, University of Huddersfield, Huddersfield, UK
matteo.paparo@hotmail.com, doria.francesco27.fd@gmail.com, marco.maratea@unical.it, m.vallati@hud.ac.uk

Abstract

The single-agent puzzle game Good Snowman aims to navigate a confined, maze-like environment with the goal of constructing snowmen by stacking three snowballs. It can be naturally characterised as a planning problem, and a first encoding has been provided via classical planning.

Motivated by the fact that parts of the problem can be conveniently represented using numeric predicates, in this paper we present a numeric PDDL representation of the Good Snowman problem. Results show that the numeric representation leads to computational advantages over the classical formulation, that can be further improved by a dedicated heuristic.

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 (Silli 2010). 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 the PSPACE-complete game *A Good Snowman is Hard to Build* (He and Yang 2017), where a character has to generate and move snowballs to build a required number of snowmen. This game (A Good Snowman, for short, hereafter) can be naturally characterised as a planning problem, where the task is 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. Recently, an encoding for the game has been provided via classical planning (Bofill et al. 2023); 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 PDDL2.1 representation (Fox and Long 2003; Scala et al. 2020) 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 proposed PDDL2.1 model coupled with the ENHSP planning engine (Scala et al. 2020) show that the numeric representation leads to computational advantages over the classical formulation running both ENHSP and the best planner in the analysis of (Bofill et al. 2023), namely SYMK (Speck et al. 2019). 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.

Background on Numeric Planning

Let us very compactly introduce numeric planning, following (Scala and Bonassi 2025). A numeric planning problem P is a tuple $\langle X, F, I, G, A, c\rangle$, where X and F are numeric and Boolean variables. The state space of P is the set of all valuations for X and F. Let $v \in X \cup F$, and s be a state, $s(v)$ indicates the value of v in s. I is the initial state. Let $\mathcal{L}$ be the language of propositional formulas having as terminals either numeric conditions, i.e., $\xi\{\geq, >, =\}0$ with ξ a numeric expression over X and $\mathbb{Q}$, or Boolean equality, i.e., $p \in \{\top, \bot\}$ with $p \in F$. G, the goal, is an element of $\mathcal{L}$. A is a set of actions such that each $a \in A$ is a pair $\langle \mathrm{pre}(a), f(a)\rangle$ where $\mathrm{pre}(a) \in \mathcal{L}$ and $f(a)$ is a set of numeric and Boolean assignments. Numeric assignments are of the form $x \leftarrow \xi$ with $x \in X$ and ξ being a numeric expression; Boolean ones are of the form $p \in \{\top, \bot\}$. c is a function assigning non-negative values to actions in A. Applying an action $a \in A$ in s generates a new state s' (denoted $s' = s[a]$), such that for all $v \in X \cup F$ we have $s'(v) = s(v)$ if v is not affected by a. Otherwise, if $v \in X$, then $s'(v) = \xi$ with $(v \leftarrow \xi) \in f(a)$, else (if $v \in F$) $s'(v) = \top$ if $(v \leftarrow \top) \in f(a)$ or $s'(v) = \bot$ if $(v \leftarrow \bot) \in f(a)$. We assume no conflicting effects.

A plan for $P = \langle X, F, I, G, A, c\rangle$ is an action sequence $\pi = \langle a_0, \ldots, a_{n-1}\rangle$ from A. Let $\langle s_0, s_1, \ldots, s_n\rangle$ be the induced state trajectory from I, where $s_0 = I$ and $s_{i+1} = s_i[a_i]$. π solves P if $\forall i = 0, \ldots, n-1 : s_i \models \mathrm{pre}(a_i)$, and $s_n \models G$.

The Good Snowman Problem

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 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 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 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 unoccupied.

The objective of each level is to utilize every 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. Because snowmen can be built on any playable cell, the player must 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.

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.

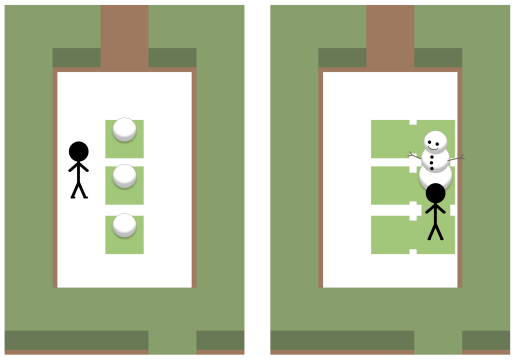

Figure 1: Tanya level, illustrating the execution of the optimal solution sequence `dRdrUlluuRurDlldR`. Each character denotes an atomic movement in a cardinal direction, where uppercase characters represent snowball pushes and lowercase characters denote agent walk moves (r/R=right, l/L=left, u/U=up,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

PDDL2.1 Numeric Model

Our numeric model modifies the PDDL model introduced in (Bofill et al. 2023) by replacing its predicate-based size system 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. A `location` represents a grid cell that may contain the character, a stack of balls, or snow. A `direction` enumerates the four directions (up, 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)`, to indicate the current size of ball `?b`, `(snowmen_count)` to count the number of built snowmen, and finally `(total-cost)`.

Three actions are defined: `move_character`, which allows the character to navigate between adjacent free locations; `move_ball` (Listing 1), which enables the character to push balls across the grid, including stacking them or increasing their size when moving over snow; and `build_snowman`, 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 cornerstone of the proposed model is the move_ball action. This 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 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.

A problem can be straightforwardly characterised based on the introduced predicates, by defining the position of the character, the initial position and size of balls, the locations that are covered in snow, and the required number of snowmen.

Listing 1: The move_ball operator.

```
1  (:action move-ball
2       :parameters (?b - ball ?pos ?from ?
                 to - location ?dir - direction)
3       :precondition (and
4          (next ?pos ?from ?dir)
5          (next ?from ?to ?dir)
6          (character_at ?pos)
7          (not(snowman_at ?to))
8          (ball_at ?b ?from)
9          (not (ball_used_in_snowman ?b))
10
11         (forall (?o - ball)
12             (or
13                 (= ?o ?b)
14                 (not (ball_at ?o ?from))
15                 (< (ball_size ?b) (
                       ball_size ?o))))
16
17             (or
18                 (forall (?o - ball)
19                     (or
20                         (= ?o ?b)
21                         (not (ball_at ?o
                               ?from))))
22                 (forall (?o - ball)
23                     (not (ball_at ?o ?to
                           ))))
24             (forall (?o - ball)
25                 (or
26                     (not (ball_at ?o ?to
                           ))
27                     (< (ball_size ?b) (
                           ball_size ?o)))))
28       :effect (and
29          (occupancy ?to)
30          (not (ball_at ?b ?from))
31          (ball_at ?b ?to)
32          (increase (total-cost) 1)
33
34          (when (forall (?o - ball)
35                  (or
36                      (= ?o ?b)
37                      (not (ball_at ?o ?
                            from))))
38                  (and
39                      (not (character_at ?
                            pos))
40                      (character_at ?from)
41                      (not (occupancy ?
                            from))))
42
43                (when (and
44                    (snow ?to)
45                    (< (ball_size ?b) 3))
46                    (and
47                        (increase (ball_size
                             ?b) 1)
48                        (not (snow ?to))))))
```

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: JOINTSNOW(b_A, b_B, t, σ), CLEANDIST(b_C, t, σ) and POPPENALTY(ρ, t, σ), described in the following:

- JOINTSNOW(b_A, b_B, t, σ) 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 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).

Informally, it is easy to notice that the proposed heuristic is admissible. In a nutshell, $h^{snowman}(\sigma)$ 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.

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
```

Experimental Evaluation

We evaluate the proposed numeric model and heuristic on the benchmark introduced in (Bofill et al. 2023), consisting of 51 instances ranging from 1 to 3 snowmen and from 14 to 99 valid locations. All experiments were run 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. Note that the results in Bofill et al. (Bofill et al. 2023) have a time limit of one hour per run and have been obtained on a different machine, so results between the two papers are not directly comparable.

We compare the performance of four planning engines: basic-ENHSP uses the original classical encoding under Greedy Best-First Search with ENHSP-25 (Scala et al. 2020, 2016); basic-SymK uses the same encoding under Fast Downward v22.12 (Helmert 2006) and SymK (Speck et al. 2019) v3.0 bidirectional search, replicating the state-of-the-art configuration from (Bofill et al. 2023); 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.

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 (Bofill et al. 2023), 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

	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 $\times$ timeout for unsolved instances) for each approach.

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 $h^{snowman}$-ENSHP, which solves 43 instances (10 more than numeric-ENHSP) at half the PAR-2 score (6.285).

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.

Conclusion

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.

Overall, this work confirms that a Good Snowman is hard plan, also numerically – extending the results in (Bofill et al. 2023). Future work will focus on extending the empirical evaluation and considering the use of reformulation techniques to tame the snowmen.

References

Bofill, M.; Borralleras, C.; Espasa, J.; Martin, G.; Patow, G.; and Villaret, M. 2023. A Good Snowman is Hard to Plan. In *ICAPS 2023 KEPS workshop*.

Fox, M.; and Long, D. 2003. PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains. *J. Artif. Intell. Res.*, 20: 61–124.

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

Helmert, M. 2006. The Fast Downward Planning System. *Journal of Artificial Intelligence Research*, 26.

Scala, E.; and Bonassi, L. 2025. On using lazy greedy best-first search with subgoaling relaxation in numeric planning problems. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 35, 245–249.

Scala, E.; Haslum, P.; Thiebaux, S.; and Ramirez, M. 2016. Interval-based relaxation for general numeric planning. In *Proceedings of the Twenty-second European Conference on Artificial Intelligence*, 655–663.

Scala, E.; Haslum, P.; Thiébaux, S.; and Ramírez, M. 2020. Subgoaling Techniques for Satisficing and Optimal Numeric Planning. *J. Artif. Intell. Res.*, 68: 691–752.

Silli, E. 2010. Mirror's Edge - Level Design Challenges and Solutions. In *GDC Europe*.

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