Plotting: A Case Study in Lifted Planning with Constraints

Joan Espasa[1*], Ian Miguel[1], Peter Nightingale[3], András Z. Salamon[1] and Mateu Villaret[2]

[1*]School of Computer Science, University of St Andrews, St Andrews, Fife, KY16 9SX, UK.
[2]Departament d'Informàtica, Matemàtica Aplicada i Estadística, Universitat de Girona, Girona, Catalunya, 17003, Spain.
[3]Department of Computer Science, University of York, Deramore Lane, York, YO10 5GH, UK.

*Corresponding author(s). E-mail(s): jea20@st-andrews.ac.uk;
Contributing authors: ijm@st-andrews.ac.uk; peter.nightingale@york.ac.uk; Andras.Salamon@st-andrews.ac.uk; mateu.villaret@udg.edu;

Abstract

We study a planning problem based on Plotting, a tile-matching puzzle video game published by Taito in 1989. The objective of this turn-based game is to remove a target number of coloured blocks from a grid by sequentially shooting blocks into the same grid. Plotting features complex transitions after every shot: various blocks are affected directly, while others can be indirectly affected by gravity. We consider modelling and solving Plotting from two perspectives. The puzzle is naturally cast as an AI Planning problem and we first discuss modelling the problem using the Planning Domain Definition Language (PDDL). We find that a model in which planning actions correspond to player actions is inefficient with a grounding-based state-of-the-art planner. However, with a more fine-grained action model, where each change of a block is a planning action, solving performance is dramatically improved. We also describe two lifted constraint models, able to capture the inherent complexities of Plotting and enabling the application of efficient solving approaches from SAT and CP. Our empirical results with these models demonstrates that they can compete with, and often exceed, the performance of the dedicated planning solvers, suggesting that the richer languages available to constraint modelling can be of benefit when considering planning problems with complex changes of state. CP and SAT solvers solved almost all of the largest and most challenging instances within 1 hour, whereas the best planning approach solved approximately 30%. Finally, the flexibility provided by the constraint models allows us to easily curate interesting levels for human players.

Keywords: Case Study, AI Planning, Modelling, Constraint Programming

Acknowledgments. Ian Miguel is funded by EPSRC grant EP/V027182/1, Mateu Villaret is funded by MCIN/AEI/10.13039/501100011033 grant PID2021-122274OB-I00 and Peter Nightingale is funded by EPSRC grant EP/W001977/1.

1 Introduction

Automated planning is a fundamental discipline in Artificial Intelligence [1]. Given a model of the environment, a planning problem is to find a sequence of actions to progress from an initial state of the environment to a goal state while respecting some constraints. Examples of planning problems in industry and academia are numerous, such as drilling operations [2], logistics [3] or chemistry [4]. Among other techniques, Constraint Programming (CP) has been successfully used to solve planning problems [5, 6]. CP is especially suited to solve planning problems when the problem requires a certain level of expressivity, such as temporal reasoning or optimality [7, 8].

Herein, we focus on finding optimal solutions for a discrete time and space puzzle, *Plotting*, a puzzle video game published by Taito in 1989 and ported to many platforms. The objective is to reduce a given grid of coloured blocks to a goal number or fewer (Figure 1). This is achieved by the avatar character repeatedly shooting the block it holds into the grid. It is also known as *Flipull* in Japan as well as in versions for the Famicom and Game Boy.

Plotting [9] is naturally characterised as a planning problem, to find a sequence of positions from which to fire such that enough blocks are removed to beat the current scenario. It is of interest because of the complexity of the state transitions after every shot: some blocks are affected directly, while others can be indirectly affected by gravity, as explained in Section 2.

Modelling the complex dynamics of the game in the de-facto standard Planning Domain Definition Language (PDDL [10]) is challenging, as we will demonstrate. Most state-of-the-art AI planners rely on grounding, instantiating every action schema for all meaningful combinations of parameters. A PDDL model in which planning actions correspond to player actions (presented in Section 4.1) is sufficiently complex that grounding becomes so inefficient as to severely hinder the ability of current planning systems to produce a valid plan (see Section 7). For comparison, we also present a more fine-grained action model in Section 4.2, where each change in the position of a block is a planning action, which performs significantly better in practice.

Problems with grounding are now attracting attention in the planning community [11], with suggestions to avoid grounding *lifted* representations as far as possible. A lifted representation succinctly defines actions by grouping them with their preconditions and effects using action schemas with parameters [12]. Constraint modelling languages can be used to express planning problems [6, 8, 13, 14]. They are richer than PDDL and, while still a challenge to formulate, permit a succinct lifted representation of Plotting and provide access to lifted *solving* approaches, i.e. solving approaches that don't need exhaustive grounding.

We present two models of the game in Essence Prime [15] and employ Savile Row [16] to transform them into SAT and CP instances for solution. Our empirical results with these models demonstrates that they can compete with, and often exceed, the performance of the dedicated planning solvers on the fine-grained PDDL model. With the largest and most challenging instances, we found that CP and SAT solvers could solve almost all of them within 1 hour, while the most competitive planner (and planning formulation) solved approximately 30%. The results suggest that the richer languages available to constraint modelling can be of benefit when considering planning problems with complex changes of state.

Plotting is also of interest as an example application in the video games industry, which in 2021 was valued at over USD 300 billion [17]. Puzzle games are perennially popular, with other examples similar to Plotting including Puzznic (Taito, 1989) and Lumines (Q Entertainment, 2004). Constraint Programming can provide a tool to assist game designers [18]. Randomly generated levels are commonly used either to save developer time or to generate more content for players. The ability to model game mechanics and solve generated levels provides the opportunity to check if they have a solution, or to gauge their difficulty [19]. This paper contributes to such aspects of game design; in addition to the constraint and PDDL models we provide a parameterised instance generator, and an empirical evaluation of the proposed models with a variety of solving back-ends.

2 Plotting

Plotting is played by one agent with full information of the state, and the effects of each action are deterministic. This situation is common in puzzle-style video games, and similar to pen and paper puzzles [20], some variants of patience like Black Hole [21], and board games such as peg solitaire [22] or the knight's tour [23]. The objective in Plotting is to reduce a given grid of coloured blocks down to a goal number or fewer. This is achieved by the avatar character shooting the block it holds into the grid, either horizontally directly into the grid, or by shooting at the wall blocks above the grid, and

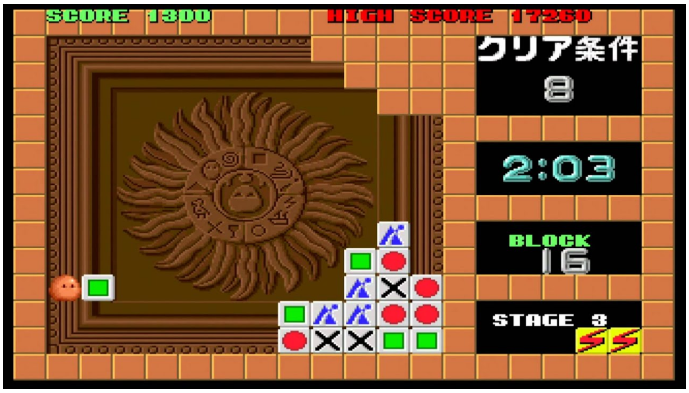


Fig. 1: Plotting (Taito, 1989). The avatar is seen on the left, holding a green block. The objective is to reduce the number of blocks in the middle pile. In this particular case there are 16 left (see center-right of the image), and the goal is 8 or less (see top-right of image).

bouncing down vertically onto the grid. Note that we consider the topmost row as the first row and the leftmost column as the first column as well. When shooting a block, if it hits a wall as it is travelling horizontally, it falls vertically downwards. In a typical level, additional walls are arranged to facilitate hitting the blocks from above. Alternatively, if it falls onto the floor, it rebounds into the avatar's hand.

The rules for a shot block S colliding with a block B in the grid are a bit more complex:

- If the first block S hits is of a different type from itself, S rebounds into the avatar's hand and the grid is unchanged — a null move.
- If S and B are of the same type, B is consumed and S continues to travel in the same direction. All blocks above B fall one grid cell each.
- If S, having already consumed a block of the same type, hits a block B of a different type, then S replaces B, and B rebounds into the avatar's hand.

A simple horizontal shot is depicted in Figure 2. A red block is shot, consuming the two red blocks of the second row and traversing the empty space between them. It replaces the green block, which rebounds to the avatar's hand, ready for the next action. Blocks above the two removed red blocks fall. A more complex shot is depicted in Figure 3, where a green block consumes an entire row of the grid, hits the wall, and continues to consume blocks as it falls until it finds a block of a different colour (red). Finally, the block shot replaces the final red block, which rebounds to the avatar's hand. As before, blocks above the consumed green blocks fall. If, after making a shot, the block that rebounds into the avatar's hand is such that there is now no possible shot that can further reduce the grid, we reach a dead end and the block in the avatar's hand is transformed into a wildcard block, which transforms into the same type as the first block it hits. Each level also begins with the avatar holding a wildcard block. In our models we consider the task of finding a solution while avoiding dead ends, since each dead end causes the loss of one of the player's lives. Therefore we are solving a strictly harder problem than the original game, and as a consequence some of the instances that we are considering in the experimental sections are unsatisfiable (i.e. there exists no plan of any length).

Considered as a planning problem, Plotting's initial state is the given grid, and there are usually multiple goal states where the grid is sufficiently reduced to meet the target. We abstract the avatar's movement to consider the key decisions: the rows or columns chosen at which to shoot the held blocks. Therefore, the sequence of actions to get us from the initial to the goal state is comprised of individual shots at the grid, either horizontally or vertically.

3 Background

This section introduces the needed background concepts that underpin the contributions of this work. We will start by explaining the assumptions of Classical Planning and how a problem is formalised. We will describe how a planning problem can be solved using Constraint Programming and end with a description of the standard representation language used by the Automated Planning community.

3.1 Classical Planning

The problem of planning, in its most basic form, consists in finding a sequence of actions that will allow reaching a goal state from a given initial state. The notion of state and the kind of allowed transitions between them determine the planning framework best suited to the problem at hand. The classical planning problem considers a finite state scope where actions are deterministic and instantaneous, planners have full world observability and the world is only modified by the agent.

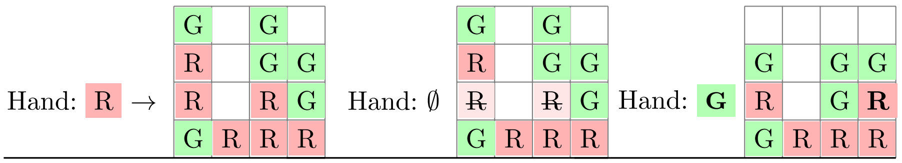


Fig. 2: Diagram of a horizontal shot to the third row and reaching the fourth column. R and G denote red and green blocks respectively. The initial state is shown in the left figure. The middle figure shows the blocks directly affected: the two light-red crossed out blocks will be removed, and all of the blocks above them will fall downwards. Finally, the right figure shows the resulting state after the shot, having swapped the hand's initial colour for the first one found in the trajectory that is different. A vertical shot works similarly.

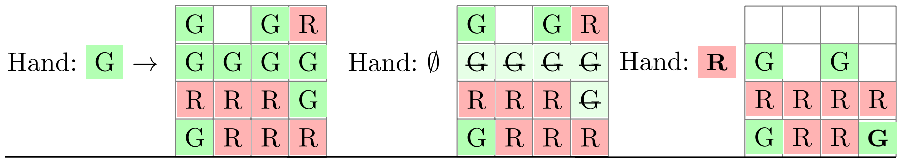


Fig. 3: A more complex shot where the firing block reaches the end and goes downwards. Note the top right red block has to fall a variable number of positions (two in this case), depending on the state of the board and the colour of the shot.

More specifically, in the classical planning problem [24] a finite set of finite-domain variables determines the state at each moment; and actions are formalized as pairs $\langle p, e\rangle$, where p is a precondition on the state variables and e denotes the changes to variables which will be made when the action is applied. Applying an action $a = \langle p, e\rangle$ to a state s requires that s satisfies the precondition p and results in a state $a(s)$ where changes in the variables listed in e have been made, and that all variables not occurring in e remain unchanged. Given an initial state I, a set of possible actions A, and a goal formula G, a solution (*plan*) to the planning problem consists of finding a sequence of actions from A: $a_1, \ldots, a_n$ such that, for all $i \in 1 \ldots n$, a_i is applicable in s_{i-1}, being $s_0 = I$ and $s_i = a_i(\ldots(a_1(s_0))\ldots), i \in 1 \ldots n$, and such that s_n satisfies G.

3.2 Planning as Satisfiability

Although planning was initially considered a deduction problem, it was rapidly seen that it could be also addressed by looking at it as a satisfiability (model finding) problem [25]. Works such as [26] showed that off-the-shelf SAT solvers could be effectively used to solve planning problems. In the last decade, the power of SAT technology has been leveraged for planning [27], making reduction to SAT a competitive approach for classical planning. When all solutions of a planning problem have a same fixed length, such as peg solitaire [22], modelling in a constraint language is simplified to deciding a fixed-length sequence of actions. Otherwise, the modeller must consider how to find a plan of unknown length. It is common in this situation to solve the planning problem by considering a sequence of queries in the form of satisfaction problems ϕ_0, ϕ_1, ϕ_2, ..., where ϕ_i encodes the existence of a plan that reaches a goal state from the initial state in i steps. The solving procedure will sequentially test the satisfiability of ϕ_0, ϕ_1, ϕ_2, and so on, until a satisfiable formula ϕ_n is found, proving the existence of a valid plan of n steps. There have been various successful approaches to encoding a planning problem into SAT [25, 28], into SMT [29, 30] and to CP [6, 8, 14, 31], amongst others.

As described in Section 5, in constructing each ϕ herein we take the common approach [6, 22] of formulating a "state and action" constraint model of the planning problem, where we employ decision variables to capture both the state of the puzzle at each time step and the action taken to transform the preceding into the succeeding state. Constraints ensure that when an action is executed, its preconditions hold with respect to the current state and its effects are applied to modify the current state. Constraints on the variables representing the state of the final step require that the goal conditions are met. Finally, *frame axioms* are made explicit, by constraints specifying that if no action has modified a variable, then the variable keeps its value between steps. There are semantics such as the $\forall$ and $\exists$-step [28], or transition-systems [32] that allow more than one action per step. Since we are interested in optimal plans in the total number of actions, we consider sequential plans with one action per step.

3.3 PDDL: Planning Domain Definition Language

There are several ways of defining planning domains but the Planning Domain Definition Language (PDDL) [10] has become a *de facto* standard. PDDL separates a planning problem into two files: the *domain*, which defines the general characteristics of the problem such as the representation of the state and how the actions operate, and the *problem*, which defines the objects, the initial state and the goal of a particular instance of the planning problem to solve. Since historically many planners were written in LISP, PDDL uses a similar syntax. A PDDL domain consists of the following components:

Requirements: PDDL is a standard but it allows many extensions. If a domain requires some of these extensions they must be declared as requirements. These include the ability to introduce new types, allowing negative preconditions, supporting conditional effects, or usage of universal and existential quantifiers in preconditions.
Types: Objects and parameters (of actions, predicates and functions) in PDDL can have types. Types form a hierarchy, with a root type (named `object`) containing all objects.
Predicates and Functions: These describe the state of the problem at each time step. Predicate and function parameters may have a specific type, or refer to all objects. Predicates are either true or false at any point in a plan and when not declared are assumed to be false (except when the Open World Assumption is included as a requirement). Functions always return a real number.[1]
Actions: These are also referred to as operators, and may have typed parameters. They embody the preconditions and effects, and usually constrain the action parameters. Preconditions define the requirements a state must satisfy in order for the action to be applicable. Effects define how the actions change the state once the action has been applied.

A PDDL problem (or instance) consists of the following components:

Objects: The (finite set of) elements playing a role in the world to be described is defined in each particular instance of the domain. Objects must have an associated type (with `object` being the ultimate super type).

[1]If they have not been initialized they simply have an undefined value.

Initial state: The initial configuration of the world in each particular instance is described through a set of atoms, which use the predicates and the functions of the domain on the defined set of objects.
Goal state: A description of the final state is given as a formula, using the predicates and the functions of the domain and the objects of the current instance. Depending on the *requirements* in the domain, there are several expressivity levels for the goal formula, with the simplest form being a conjunction of literals.

3.4 Limitations of Classical Approaches to Planning

Tools to solve problems such as Plotting should ideally support natural ways of expressing models for these problems. Natural ways to structure such models include using matrices to represent the state of play, a way to index the entries in such matrices, and a representation of the states of the blocks. A popular extension to PDDL [33] added support for numeric and temporal features, extending the expressivity of the language. Still, such an extension is insufficient for naturally modelling Plotting. In [33], it is stated:

> Numeric expressions are not allowed to appear as terms in the language (that is, as arguments to predicates or values of action parameters) . . . Functions in PDDL2.1 are restricted to be of type $Object_n \rightarrow \mathbb{R}$, for the (finite) collection of objects in a planning instance, *Object* and finite function arity n.

Namely, no action, predicate or function can have a number as a parameter. Sadly, these severe limitations render this PDDL extension useless for our needs. Note that an essential construct in the preconditions and effects of the actions would be the usage of arithmetic to deal with indices of rows and columns that actions should have as parameters. For example, when we remove a block in a given `row` and `col`, if there was a block above it, this block would fall and we would need to refer to its colour. As we will show in Section 5, this can be easily expressed in Essence Prime by arithmetically operating on the indices of the matrix: `grid[row-1, col]`. Unfortunately, since `row` cannot be a numeric parameter in PDDL, we are forced to use quantifiers to be able to refer to the "block that is above it" (i.e. its row is equal to `row-1`). Therefore, as we will see in Section 4, we are forced to define predicates to simulate some basic arithmetic operations on indices.

Works such as Functional STRIPS [34] or Planning Modulo Theories [35] would alleviate the expressivity problems faced with Plotting. On one hand, with Functional STRIPS extensions such as [36] we would be able to both simulate matrices thanks to proper support for functions in the language, and to operate on their indices thanks to the arithmetic support. On the other hand, with the Planning Modulo Theories paradigm one could ideally devise a specific theory solver supporting both matrices and arithmetic.

Unfortunately, those approaches have either not been released, or are not actively maintained, and we have been unable to use these off the shelf. In particular, the FS planner [36] has dependencies on software which we have not been able to find and the Planning Modulo Theories planner in [35] was not released. Therefore, it would require a significant engineering effort to either reproduce or re-engineer them. Hence, by considering the available and well supported planners, we are limited to using state-of-the-art classical planners based on PDDL despite their severe limitations in terms of expressivity.

4 Modelling Plotting in PDDL

In this section we provide fragments of the two models to illustrate the main drawbacks of PDDL for modelling Plotting. The game board is abstracted as a grid of coloured cells. The colour of each cell is the colour of the block it contains, or `null` if empty. Therefore, the full viewpoint is the colour of each cell and the colour of the block in the avatar's hand.

The model described in Section 4.1 maps the possible player actions in the game to actions in PDDL, aligning very closely to the CP model of Section 5.3. Given the complexities introduced by this PDDL model, we will see in the evaluation that it does not suit the planners well. Hence, we propose a second model in Section 4.2 which better suits the strengths of the planners.

4.1 A Direct Model

As previously discussed, shots are performed either horizontally and directly into the grid, or "vertically" by shooting at the wall blocks above the grid, and bouncing down vertically onto the grid. This naturally entails two actions: shoot horizontally or shoot vertically. Still, the effects of any horizontal shot are complex to express, as there are multiple different outcomes depending on the grid configuration: either the shot bounces back before reaching the end of the row, the shot consumes the whole row and bounces back before consuming any block in the last column, or the shot consumes the full row and also some blocks from the last column. Therefore, to improve the model's clarity and simplify the actions we have further subdivided the horizontal shot into three different actions that correspond to the aforementioned cases.

To parameterise the actions and the predicates defining the state, we use two types of objects: `colour` and `number`, where `number` is the name of a type used to manually encode the basic required numerical properties. The predicate `hand` has one colour parameter, and encodes if the avatar has a block of the given colour. Given parameters `row`, `col` and `c`, the `coloured` predicate expresses if the block in that row and column has the given colour.

```
(hand ?c - colour)
(coloured ?row ?col - number ?c - colour)
```

Auxiliary predicates such as `islastcolumn` or `isbottomrow` are added for clarity and to reduce the use of quantifiers and so the burden on the planner's preprocessor.

```
(isfirstcolumn ?n - number)
(islastcolumn ?n - number)
(istoprow ?n - number)
```

```
(isbottomrow ?n - number)
```

Moreover, we need to encode some integer relations as Boolean predicates:

```
(succ ?p1 ?p2 - number)          ; p1 is successor of p2
(lt ?p1 ?p2 - number)            ; p1 is less than p2
(distance ?p1 ?p2 ?p3 - number) ; p3 is p2 - p1
```

These predicates must be defined in each instance file, along with the information specific to each scenario. For instance, when dealing with a 5×5 board we need to state `succ` for every pair of successive numbers between 1 and 5, and `lt` and `distance` for every pair of two numbers (p_1, p_2) between 1 and 5 such that $p_1 < p_2$.

Figure 4 is an excerpt of the action consisting of partially removing blocks of colour `?c` in row `?r` until column `?t`, i.e. not reaching the last column. One of the principal difficulties is in identifying successors and predecessors of particular rows or columns (e.g. Lines 10, 16, 25, 36), which could have been addressed through support for arithmetic expressions on parameters.

The lack of support for multi-valued variables makes the encoding of some transitions difficult. For example, when changing the colour held by the avatar we must state: *remove previous colour in the hand and set the new colour* (lines 31-32). Multi-valued variables would make this change straightforward. Due to the lack of support for function symbols in the considered PDDL fragment, we must also employ quantification to name specific objects. For instance, the column of the cell next to `?t` (`?nextcolumn`) and its colour (`?nextcolour`) have to be discovered. This quantification is introduced in line 25, and the values of `?nextcolumn` and `?nextcolour` are discovered in lines 26-28 as a condition for the effect to take place.

If we could use function symbols and arithmetic, we could remove variables `?nextcolumn` and `?nextcolour`, changing the `coloured` symbol to a function that, given a row and column, maps to the colour in that cell. Overall, lines 25-32 could theoretically be simplified to:

```
(assign (hand (coloured ?r (?t + 1))))
(assign (coloured ?r (?t + 1)) ?c)
```

Unfortunately, as per the previous subsection, functions cannot have numeric expressions as parameters. Essence Prime naturally deals with these kinds of statements (see Section 5).

Finally, we must define the initial and goal states for every instance. The initial state is simply stated with a `coloured` statement for each cell. However, the goal state is more complex to express if we do not have arithmetic or aggregate functions to count the number of cells coloured with `null`. In our instances we define the goal as follows. Let g be the maximum allowed number of non-`null` cells in order to satisfy the goal state. We require that there exist g different cells such that any other cell is `null`. For instance, requiring at most 2 non-`null` cells creates the following statement:

```
(:goal  ;; at most 2 cells are not null, i.e. g=2
  (exists (?x1 ?x2 ?y1 ?y2 - number)
    (and (or (not (= ?x1 ?x2))
(:action shoot-partial-row
  ;; ?r - what row we are shooting at
  ;; ?t - the end cell where the shot ends
  ;; ?c - the colour we are removing

  :parameters (?r - number ?t - number ?c - colour)

  :precondition (and
    ;; ?col is the successor of ?t with a different colour than ?c
    (exists (?col - number)
      (and (succ ?col ?t)
           (not (coloured ?r ?col ?c))
           (not (coloured ?r ?col null))))
    ...
    ;; all the blocks up to ?t have either the colour ?c or are null
    (forall (?col - number)
      (or  (lt ?t ?col)
           (and (= ?col ?t) (coloured ?r ?t ?c))
           (or (coloured ?r ?col ?c)
               (coloured ?r ?col null)))))

  :effect (and
      ;; Change hand colour
      ;; the next cell that we cannot remove gets the hand colour
      (forall (?nextcolumn - number ?nextcolour - colour)
        (when
          (and (succ ?nextcolumn ?t)
               (coloured ?r ?nextcolumn ?nextcolour))
          (and (not (coloured ?r ?nextcolumn ?nextcolour))
               (coloured ?r ?nextcolumn ?c)
               (hand ?nextcolour)
               (not (hand ?c)))))

      ;; Move everything downwards.
      ;; Consider 2 cases: base case (top row), and general case (rest).
      (forall (?currentrow ?nextrow ?currentcol - number)
        (and ;; First, the general case. Any row except the top one
          (forall (?currentcolor ?nextcolor - colour)
            (when
              (and (lt ?currentrow ?r)
                   (succ ?nextrow ?currentrow)
                   (or (lt ?currentcol ?t) (= ?currentcol ?t))
                   ;; ensure that the cells have the pertaining colours
                   (coloured ?currentrow ?currentcol ?currentcolor)
                   (coloured ?nextrow ?currentcol ?nextcolor)
                   ;; avoid a contradiction:
                   (not (= ?currentcolor ?nextcolor)))
              (and (not (coloured ?nextrow ?currentcol ?nextcolor))
                   (coloured ?nextrow ?currentcol ?currentcolor)))))))

          ; Then, the base case of firing on the top row.
          ...))
```

Fig. 4: Fragment of the action *shoot-partial-row* of the the PDDL model. Note that the `when` operator has two parameters: the condition and the effect.

```
            (not (= ?y1 ?y2)))
      (forall (?x3 ?y3 - number)
          (or ; Or is one of cell 1 or cell 2, or is null
            (and (= ?x1 ?x3) (= ?y1 ?y3))
            (and (= ?x2 ?x3) (= ?y2 ?y3))
            (coloured ?x3 ?y3 null))))))
```

The length of this goal is $\Theta(g^2)$, since the g cells must be pair-wise different. Again, this is much simpler to state in a constraint language with, for example, an `atleast` constraint.

4.2 A Granular Model

Given the complexities of the previous model, we now present an alternative formulation with the aim of simpler actions that may ease the grounding process of the planners and that are easier to define. The grid representation is now slightly changed: we add one additional leftmost empty column to allocate the first position of the shot block, one additional rightmost column to represent the wall and one additional wall row to represent the floor. Note that in the Direct model rows are numbered top-down but in this model we are numbering rows bottom-up, such that the floor row is always row 1. Note also that `loc_c_r` refers to the location at column `c` and row `r`.

Now every single block change will correspond to one action, therefore we will have actions for shooting, for advancing the shot block and for making the floating blocks fall. We will model the shot block as a "bullet" that will move through the grid according to the rules of the game. For instance, consider again the example in Figure 2. In the previous PDDL model, this shot would just correspond to action `(shoot-partial-row n3 n3 red)` because it partially removes row 3 until column 3. When considering the granular model the shot of Figure 2 would correspond to the following sequence of actions:

```
(shooting_row_3)
(horiz_bullet_adv_absorbing loc_1_3 loc_2_3 red n12 n11)
(horiz_bullet_adv_non_absorbing loc_2_3 loc_3_3)
(horiz_bullet_adv_absorbing loc_3_3 loc_4_3 red n11 n10)
(horiz_bullet_bouncing loc_4_3 loc_5_3 red green)
(fall_block loc_2_4 loc_2_3 red)
(fall_block loc_2_5 loc_2_4 green)
(fall_block loc_4_4 loc_4_3 green)
(fall_block loc_4_5 loc_4_4 green)
```

In addition to this granularity, we now consider locations as objects (e.g. `loc_3_3`) that are related to adjacent locations with the corresponding direction with the `next` predicate (e.g. `(next loc_2_3 loc_3_3 right)`). Then, to indicate the position of the shot bullet we will use the `coloured` predicate and the special colour `bullet` (e.g. `(coloured loc_4_3 bullet)`), and to indicate that a location is a wall we will use the special colour `wall`. Summing up, the used predicates are the following:

```
 (:predicates
    (hand ?c - colour)
    (coloured ?l - location ?c - colour)
    (next ?from ?to - location ?dir - direction)

    (pred ?n1 ?n2 - number)        ; ?n1 is the predecessor of ?n2
    (blocks_remaining ?n - number) ; ?n coloured blocks remain

    (falling_flag)              ; some block must fall
    (shooting_horizontal_flag)  ; horizontal shot in progress
    (shooting_vertical_flag)    ; vertical shot in progress
    (block_removed_flag)        ; shot has already removed some block
    )
```

Moreover, we also need to encode some integer relations as Boolean predicates as before. Notice that we use the unary predicate `blocks_remaining` as a substitute for a numeric variable (which we cannot use in classical planning), and it will track the number of blocks remaining. With this model blocks are removed one at a time, and therefore tracking the remaining blocks is now easy. For instance, if the goal state needs at most two blocks remaining, this can be defined as follows:

```
(:goal (and  (or (blocks_remaining n2)
                 (blocks_remaining n1)
                 (blocks_remaining n0))
             (not (shooting_horizontal_flag))
             (not (shooting_vertical_flag))
             (not (falling_flag))))
```

As we can see, there are now some `*_flag` predicates. More concretely, the three flag predicates represent: that a shoot is in progress, that there are floating blocks that must fall and that some block has been consumed[2]. The game will only reach its goal when the amount of blocks remaining is the required one, the last shot has been fully completed and all suspended blocks have fallen.

Roughly, the model has three kinds of actions: `shoot`, `advance_bullet` and `fall_block`, which use the aforementioned flags. The flags allow us to define a state machine that captures the semantics of the game, depicted in Algorithm 1. The algorithm forces the planner to select the required actions when needed, such as the `fall_block` actions when blocks need to fall and no shot is in process.

All flags are appropriately updated in the effects of actions with the exeption of the `falling_flag`, which is a *derived predicate* [37]. These predicates are instead automatically updated after each action application.

```
(:derived (falling_flag)
         (exists (?l1 ?l2 - location)
             (and (next ?l1 ?l2 down)
                  (not (coloured ?l1 none))
                  (coloured ?l2 none))))
```

Shooting actions can now be expressed concisely. We will have one shooting action for each row and each column of the problem instance we are solving. A sample action that shoots on the fifth row follows.

```
(:action shooting_row_5
```

[2]This last flag allows us to check that ending a shot by bouncing is correct.

Algorithm 1 Game Semantics

```
while goal has not been reached do
    if shooting_*_flag then
        only allow advance_*_bullet actions
    else if falling_flag then
        only allow fall_block actions
    else
        only allow shoot_* actions
    end if
end while
```

```
  :precondition (and (not (shooting_horizontal_flag))
                     (not (shooting_vertical_flag))
                     (not (falling_flag)))
  :effect (and (coloured loc_1_5 bullet)
               (shooting_horizontal_flag)
               (increase (total-cost) 1)))
```

The falling action simply takes care of one block "in suspension".

```
(:action fall_block
  :parameters (?l1 ?l2 - location ?c - colour) ; l1 falls to l2
  :precondition (and
    (not (shooting_horizontal_flag))
    (not (shooting_vertical_flag))
    (falling_flag)        ; some block must fall
    (next ?l1 ?l2 down)   ; l1 is on top of l2
    (coloured ?l1 ?c)     ; l1 has some colour and needs to fall
    (coloured ?l2 none)) ;   because l2 is empty
  :effect (and
    ; the colours gets properly assigned:
    ;   l1 loses the colour and l2 gains the colour of l1
    (not (coloured ?l2 none))
    (coloured ?l2 ?c)
    (not (coloured ?l1 ?c))
    (coloured ?l1 none)))
```

Finally, when the bullet "hits" a block of the same color, the block is removed and the bullet continues advancing:

```
(:action horizontal_bullet_advancing_absorbing
  :parameters
    ; l1 has the bullet, l2 is at its right and c is the hand's colour
    ; ni nj are numbers for counting the blocks remaining
    (?l1 ?l2 - location ?c - colour ?ni ?nj - number)
  :precondition (and ; we are in the process of shooting horizontally
    (shooting_horizontal_flag)
    (hand ?c)
    (next ?l1 ?l2 right)
    (coloured ?l1 bullet)
    (coloured ?l2 ?c)
    (blocks_remaining ?ni) ; there are ni blocks left
    (pred ?nj ?ni))        ;   and nj is the predecessor of ni
  :effect (and
    ; the colours gets properly assigned advancing the bullet
    (not (coloured ?l1 bullet))
    (not (coloured ?l2 ?c))
    (coloured ?l2 none)
    (coloured ?l2 bullet)
    (block_removed_flag)
    (not (blocks_remaining ?ni)) ; the counter is updated
    (blocks_remaining ?nj)))
```

These actions illustrate the main interesting parts of the model. The rest of the actions deal with vertical shots, a bullet advancing but not absorbing blocks, and bullets reaching the end and bouncing back.

5 Constraint Models in Essence Prime

Rendl et al. [38] provide a brief description of an incomplete constraint model of Plotting, as it does not support the difficult case of a shot travelling horizontally all the way through the grid and then continuing to consume blocks in the final column. We present two complete models of the problem, formulated in action-based and state-based styles.

Kautz and Selman [25] outlined a set of sufficient conditions to ensure that all models of the proposed encoding correspond to valid plans. These conditions are the implication of preconditions and effects by actions, the occurrence of exactly one action at each step, a completely specified initial state, and classical frame conditions, where a change of a state variable truth value means something has forcefully changed it. The action-based model aligns closely with these classical linear encodings, where the sequence of actions determines the unfolding of the plan step by step. More concretely, the state is the current grid configuration and the contents of the hand of the avatar, and the single action is a shot along a particular row or column.

In contrast, the state-based model bears resemblance to the state-based encodings by Kautz and Selman [26]. The term "state-based" is used in connection with axioms that check the validity of individual states. More concretely, the underlying idea of this model is based on the belief that a relatively small number of axioms can describe the state transitions for each variable. In other words, these axioms explain what has to happen for changes in state variables to be valid. Notably, these axioms share similarities with the "backward-chaining" axioms employed in Graphplan [39].

Our constraint models broadly conform to the state and action structure of this earlier work [26, 39]. However, they are lifted in the sense that it is possible for the solver partially to decide the details of an action rather than having to commit to an entire grounded action at once. This ability is retained as the models are encoded both for the constraint and SAT solvers we employ in our experiments. Of course, further encodings could be designed, as for example parallels can be drawn between the time-indexed state-recording grid and a state-recording automaton unfolded over time in the regular-constraint modelling approach of [8].

5.1 A Common Viewpoint

Our models share a common *viewpoint*, i.e. the choice of variables and domains, which we summarise before describing each individual model.

Each block type is identified with a colour, and the colours are represented by a contiguous range of natural numbers in Essence Prime. Empty grid cells are represented by 0. Step 0 is the initial state, with the action chosen at step 1 transforming the initial state into the state at step 1, and so on. Hence, the parameters and constants for the models are:

```
given initGrid : matrix [int(1..gridHeight),
                         int(1..gridWidth)] of int(1..)
letting GRIDCOLS be domain int(1..gridWidth)
letting GRIDROWS be domain int(1..gridHeight)
letting NOBLOCKS be gridWidth * gridHeight
letting COLOURS be domain int(1..max(flatten(initGrid)))
letting EMPTY be 0
letting EMPTYANDCOLOURS be domain int(EMPTY) union COLOURS
given goalBlocksRemaining : int(1..NOBLOCKS)
given noSteps : int(1..)
letting STEPSFROM1 be domain int(1..noSteps)
letting STEPSFROM0 be domain int(0..noSteps)
```

We capture the current state of the grid and the contents of the avatar's hand at each time step with a time-indexed 2-dimensional array of decision variables and an individual variable per time step respectively. In this model, the grid is indexed in the same way as in the Direct PDDL model, i.e. the upper-left location is index (1,1). Only one action is possible per time step, which is the decision as to where to fire the block held. Here we introduce a pair of variables per time step, one representing the column fired down (if any) and one representing the row fired along (if any):

```
find fpRow : matrix [STEPSFROM1] of int(0..gridHeight)
find fpCol : matrix [STEPSFROM1] of int(0..gridWidth)
find hand  : matrix [STEPSFROM0] of COLOURS
find grid  : matrix [STEPSFROM0, GRIDROWS, GRIDCOLS] of EMPTYANDCOLOURS
```

Note that these decision variables provide a natural lifted representation of the problem, enabling the SAT and CP solvers to directly search on them.

5.2 Common Constraints

The two models also share some constraints on the viewpoint described above, which we describe in what follows. The initial state constrains the 0th 2-dimensional array of `grid` to be equal to the parameter `initGrid`. The goal state counts the number of empty grid cells:

```
$ Initial state:
forAll gCol : GRIDCOLS .
  forAll gRow : GRIDROWS .
    grid[0, gRow, gCol] = initGrid[gRow, gCol],
$ Goal state:
atleast(flatten(grid[noSteps,..,..]),
        [NOBLOCKS - goalBlocksRemaining], [EMPTY]),
```

Having transformed Plotting into a decision problem that asks if there is a plan with a fixed number of steps, we might take the view that moves that do not alter the state of the puzzle (e.g. firing the held block into one of a different colour) might be used to "pad" a short plan to the given length. This is of little benefit and could lead to redundant search, so we disallow moves that do not progress the solution of the puzzle:

```
$ Each move must do something useful:
forAll step : STEPSFROM1 .
  sum(flatten(grid[step-1,..,..])) > sum(flatten(grid[step,..,..])),
```

Care will be necessary with our frame constraints, which we will describe in the context of the two individual models. Any cell unconstrained will be vulnerable to the solver assigning an arbitrary (low-numbered) colour so as to satisfy the sum constraint above.

The other constraint we consider here states that we must fire horizontally or vertically (a shot at the wall blocks above the grid that then bounces down) but not both:

```
$ Exactly one fp axis must be 0. (XOR, only ONE fired angle)
forAll step : STEPSFROM1 .
  fpRow[step] * fpCol[step] = 0 /\ fpRow[step] + fpCol[step] > 0,
```

5.3 An Action-focused Constraint Model of Plotting

Our two models differ in the way they describe the transition from one state to another via the action selected. We start describing a model that focuses on the action selected and what must therefore be true of the grid at the preceding step (the action's preconditions) and of the grid subsequently (the action's effects). Herein, we give an overview along with some illustrative fragments of the model. The constraints in this model are divided into two, depending on whether the shot is down a column or along a row. The column shot is simpler, as it only affects the selected column:

```
forAll step : STEPSFROM1 .
  (fpCol[step] > 0) ->
  $ All other columns are untouched.
  (forAll col : GRIDCOLS .
   (col != fpCol[step]) ->
   (forAll row : GRIDROWS . grid[step,row,col] = grid[step-1,row,col])
  ) /\
  $ Must exist a row where grid[step-1,row,fpCol[step]] = hand.
  (exists row : GRIDROWS .
   (grid[step-1,row,fpCol[step]] = hand[step-1]) /\
   $ Everything above is empty or same colour as the hand.
   (forAll above : int(1..row-1) .
     grid[step-1,above,fpCol[step]] = EMPTY \/
     grid[step-1,above,fpCol[step]] = hand[step-1]) /\
   $ Effect is to make everything down to this row empty
   (forAll clear : int(1..row) . grid[step,clear,fpCol[step]] = EMPTY)
   /\
   ($ Either this is bottom in which case hand remains same.
    (row = gridHeight) /\ (hand[step] = hand[step-1])
    \/
    $ Or the next row down is of a different colour, swaps with hand.
    (grid[step-1,row+1,fpCol[step]] != hand[step-1] /\
     grid[step,row+1,fpCol[step]] = hand[step-1] /\
     hand[step] = grid[step-1,row+1,fpCol[step]] /\
     forAll below : int(row+2..gridHeight) .
       grid[step,below,fpCol[step]] = grid[step-1,below,fpCol[step]]))
  ),
```

The row shot is considerably more complex, since its effects typically include blocks falling as a result of gravity. We must also support a horizontal shot reaching the wall on the right and falling. We sub-divide into three cases: the shot block is exchanged with another in the same row; the block is exchanged with another in the final column, having hit the wall and fallen; and the block travels all the way to the rightmost column and falls to the floor, consuming only blocks of the same colour, resulting in the same colour block returning to the hand. For brevity we show the first of these below. The two remaining cases can be found in the full model contained in the supplementary material.

```
forAll step : STEPSFROM1 .
  (fpRow[step] > 0) ->
  (exists col : GRIDCOLS .
   $ Preconds: col with a block different from hand.
   ( (grid[step-1,fpRow[step],col] != hand[step-1]) /\
     $Left, empty/hand colour, must exist a block of hand colour.
     (forAll left : int(1..col-1) .
        grid[step-1,fpRow[step],left] = EMPTY \/
        grid[step-1,fpRow[step],left] = hand[step-1]) /\
     (exists left : int(1..col-1) .
        grid[step-1,fpRow[step],left] = hand[step-1]))
   /\
   $ Effects:
   ($ left: Blocks falling, staying fixed.
    (forAll left : int(1..col-1) .
       $ Everything below is fixed
       (forall below : GRIDROWS .
          (below > fpRow[step]) ->
          (grid[step,below,left] = grid[step-1,below,left])) /\
       (grid[step,1,left] = EMPTY) /\ $ Top row guaranteed to be empty.
       $ Otherwise fall from above.
       ((fpRow[step] > 1) ->
        (forAll above : int(2..gridHeight) .
           above <= fpRow[step] ->
           grid[step,above,left] = grid[step-1,above-1,left]))
    ) /\
    $ this col: all fixed except fprow which exchanges with the hand
    (hand[step] = grid[step-1, fpRow[step], col]) /\
    (grid[step, fpRow[step], col] = hand[step-1]) /\
    (forAll colBlock : GRIDROWS .
       (colBlock != fpRow[step]) ->
       (grid[step,colBlock,col] = grid[step-1,colBlock,col])) /\
    $ right: all fixed
    (forAll right : int(col+1..gridWidth) .
       forAll colBlock : GRIDROWS .
         grid[step,colBlock,right] = grid[step-1,colBlock,right])))
```

5.4 A State-focused Constraint Model of Plotting

We now describe an alternative model that focuses on the state of the hand and each cell of the grid, how each might change or remain the same, and the valid reasons for doing so. Again, due to its substantial size we give an overview along with some illustrative model fragments. The full model is provided in the supplementary material.

We found it expedient to introduce a time-indexed set of auxiliary variables to this model to capture the distance travelled in the final column when a block is shot horizontally, reaches the wall, then consumes blocks as it falls down the last column. We use these auxiliary variables throughout the model to simplify the statement of the constraints.

```
find wallFall : matrix [STEPSFROM1] of int(0..gridHeight)
```

The constraints to make the calculation enumerate each possible value for the `wallFall` variable and stipulate what must be true for that value to be valid:

```
forAll step : STEPSFROM1 .
 forAll i : int (1..gridHeight) .
  (wallFall[step] = i)
  <->
  (exists row : int(1..gridHeight) .
    (fpRow[step] = row) /\
    $ Travelled to the rightmost column
    (forAll col : int(1..gridWidth) .
      (grid[step-1,row,col] = EMPTY) \/
      (grid[step-1,row,col] = hand[step-1])) /\
    $ Travelled i in the last column
    (forAll underRow : int (row..row+i-1) .
      (grid[step-1,underRow,gridWidth] = hand[step-1]) \/
      (grid[step-1,underRow,gridWidth] = EMPTY)) /\
    $ And no more
    (((grid[step-1,row+i,gridWidth] != hand[step-1]) /\
      (grid[step-1,row+i,gridWidth] != EMPTY)) \/
     (row+i > gridHeight)) /\
    $ And consumed a block somewhere, otherwise not a progressing move.
    ((exists col : GRIDCOLS .
        grid[step-1,row,col] = hand[step-1]) \/
     (exists underRow : int(row..row+i-1) .
        grid[step-1,underRow,gridWidth] = hand[step-1]))
  ),
```

The constraints in the state-focused model are subdivided into four cases: The hand is unchanged, a grid cell becomes empty, a grid cell stays the same and grid cell changes colour to something other than empty, which can affect the hand. These are all stated in an if-and-only-if form to ensure that no part of the state (hand or grid) is left unconstrained and therefore vulnerable to the solver assigning arbitrary values.

There are two scenarios leaving the hand unchanged when we require a progressing move. First, firing down a column of the same colour blocks as the block fired. Second, along a row of the same colour, hitting the wall, then consuming everything beneath on the rightmost column before hitting the floor. The `wallFall` variables simplify this second scenario:

```
forAll step : STEPSFROM1 .
(hand[step-1] = hand[step])
=
( $ Fired down col, hitting wall
  ( (forAll colBlock : GRIDROWS .
      ((grid[step-1,colBlock,fpCol[step]] = hand[step-1]) \/
       (grid[step-1,colBlock,fpCol[step]] = EMPTY))
    )
  ) \/
  $ Fired row, hitting wall, dropping through hand-colour only.
  $ We can test this by comparing the wallFall value with fpRow:
  (wallFall[step] = gridHeight-fpRow[step]+1)
),
```

A grid cell remains empty if it was empty at the previous time step. Otherwise it becomes empty if the block that was occupying it is deleted by the chosen shot, or the block that was occupying it falls through the action of gravity. In both of these scenarios we must check that another block has not fallen into this cell and of course we must cater for the fact that in the rightmost column several blocks can be consumed or fall. We present an illustrative fragment below, again exploiting `wallFall`, and refer the reader to the full model for the complete constraint covering this case:

```
forAll step : STEPSFROM1 .
  forAll gRow : GRIDROWS .
    forAll gCol : GRIDCOLS .
      (grid[step,gRow,gCol] = EMPTY)
      =
      ( $ When a cell is EMPTY, it stays EMPTY
        (grid[step-1,gRow,gCol] = EMPTY) \/
        ...
        $ Final Column shot along a row
        $ consuming several blocks underneath
        ( $ Only the final column
          (gCol = gridWidth) /\
          $ There was a wallfall, hence a successful row shot
          (wallFall[step] > 0) /\
          $ The shot was beneath here
          (fpRow[step] > gRow) /\
          $ Nothing there to fall into here
          (grid[step-1,gRow-wallFall[step],gridWidth] = EMPTY \/
           gRow-wallFall[step] < 1)
        ) \/ ...
      )
```

A grid cell remains unchanged from one time step to the next primarily if it is unaffected by the action chosen. This may be, for example, because a shot was fired down a different column or along a row above. A more subtle scenario is when a block falls down from the current cell, but another of the same colour falls from above to take its place. In all, we have subdivided this case into nine such scenarios, which can be seen in the full model. An illustrative fragment is shown below:

```
forAll step : STEPSFROM1 .
  forAll gRow : GRIDROWS .
    forAll gCol : GRIDCOLS .
      (grid[step,gRow,gCol] = grid[step-1,gRow,gCol])
      =
      ( $ Fired along row above, last col.
        $ Something in way on row or last col.
        ( (gCol = gridWidth) /\
          (fpRow[step] != 0) /\
          (fpRow[step] < gRow) /\
          ( (exists rowBlock : int(1..gridWidth) .
             ((grid[step-1, fpRow[step], rowBlock] != EMPTY) /\
              (grid[step-1, fpRow[step], rowBlock] != hand[step-1]))
            ) \/
            (exists colBlock : int(1..gRow-1) .
             ((colBlock >= fpRow[step]) /\
              (grid[step-1, colBlock, gridWidth] != EMPTY) /\
              (grid[step-1, colBlock, gridWidth] != hand[step-1]))
            )
          )
        ) \/
        $ This row or below. Same colour block falls here. Last col.
        ( (gCol = gridWidth) /\
          (fpRow[step] >= gRow) /\
          (wallFall[step] > 0) /\
          (grid[step-1,gRow-wallFall[step],gCol] =
           grid[step-1,gRow,gCol])
        ) \/ ...
      )
```

Finally, the contents of a grid cell change to something other than empty either as a result of an exchange with the hand or if a different coloured block falls into the cell. Here, we have subdivided into five scenarios, depending on whether a row or column shot was selected, and whether the final column is involved. A fragment is shown below:

```
forAll step : STEPSFROM1 .
  forAll gRow : GRIDROWS .
    forAll gCol : GRIDCOLS .
      ((grid[step,gRow,gCol] != grid[step-1,gRow,gCol]) /\
       (grid[step,gRow,gCol] != EMPTY))
      =
      ( ...
        $ Cell swaps with hand: row then down last col.
        ( $ rightmost col
          (gCol = gridWidth) /\
          $ WallFall implies travel row then col.
          (wallFall[step] > 0) /\
          $ and this cell must be at fpRow+wallFall
          (gRow = wallFall[step] + fpRow[step]) /\
          $ Exchanges with hand
          (hand[step] = grid[step-1,gRow,gridWidth]) /\
          (hand[step-1] = grid[step,gRow,gridWidth]) /\
          $ Which was a different colour
         (hand[step-1] != grid[step-1,gRow,gridWidth])
        ) \/ ...
      )
```

6 Instance Generation

There are two principal components of a Plotting instance: the initial grid, and the goal of the number of blocks remaining. We can take further advantage of constraint solving by formulating the generation of the initial grid as a constraint model, and in doing so break the symmetry among the block colours so as not to consider symmetric instances in our empirical work. The model, which is parameterised by the grid dimensions and number of colours, follows:

```
given gridWidth : int (1..)
given gridHeight : int (1..)
letting GRIDCOLS be domain int(1..gridWidth)
letting GRIDROWS be domain int(1..gridHeight)

given noColours : int (1..)
letting COLOURS be domain int(1..noColours)

find initGrid : matrix [GRIDROWS, GRIDCOLS] of COLOURS
find firstColourOccurrenceRow : matrix [COLOURS] of GRIDROWS
find firstColourOccurrenceCol : matrix [COLOURS] of GRIDCOLS

such that

$ Use a row-wise, left-to-right ordering to break symmetry.
forall colour : COLOURS .
  $ Block identified as firstColourOccurrence must have correct colour
  initGrid[firstColourOccurrenceRow[colour],
           firstColourOccurrenceCol[colour]] = colour
  /\
  $ All blocks above must have a smaller colour
  (forall row : GRIDROWS .
    (row < firstColourOccurrenceRow[colour]) ->
    (forall col : GRIDCOLS . initGrid[row, col] < colour))
  /\
  $ All blocks to the left on the same row must have a smaller colour
  (forall col : GRIDCOLS .
    (col < firstColourOccurrenceCol[colour]) ->
    (initGrid[firstColourOccurrenceRow[colour], col] < colour))
```

In addition to a matrix to represent the grid itself, we add a pair of variables for each of the colours. Each pair represents the first occurrence of the corresponding colour in the row-wise, left-to-right ordering we have chosen as canonical. Symmetry is broken by insisting that all the rows above, and the columns to the left on the same row, as this first occurrence contain a 'smaller' colour. Since the first occurrence variables must be assigned, these constraints also have the side-effect that every colour specified must be used.

For given grid dimensions and a given number of colours, our approach to generating complete instances for experimentation is to sample from the space of solutions to the model above. We solve the model with MINION [40] with random variable and value ordering.

We generated 174 new grids, where their number of blocks range from 8 to 49, the number of colours range from 2 to 6 and their sizes scale gradually from 2×4 to 7×7. As rows and columns behave differently due to gravity, the dataset contains grids both with more rows than columns and grids with more columns than rows. For each of the 174 grids, we create three instances: an easy goal of half of the initial number of blocks, a hard one being exactly the number of colours of the grid, and the hardest one, being the number of colours minus one. This hard bound is derived using the following observation.

It is possible to establish a simple lower bound on the achievable number of blocks remaining in the grid, as the following lemma shows:

Lemma 1 *The lower bound on the number of blocks remaining in the grid is one less than the number of distinct colours in the initial grid.*

Proof Consider, without loss of generality, the transition from a red to a green block in the hand. This means a shot has been made that has consumed at least one red block and hit a green block, whereupon the green block has rebounded into the hand. The red block replaces the green block that it hit, rather than disappear however. This is a general pattern: every time there is such a transition, the previous colour is left behind in the grid. The lower bound in the grid is one less than the number of colours because the colour to which we transition (green in our example) can appear in the hand alone. The initial "wild card" block does not change matters, since it simply "becomes" the colour of the first block it touches. □

Note that if we consider Lemma 1, we know that we can never reach a state with fewer blocks than the initial number of colours, but we can arrive to a valid state where we have one block in the hand and the number of colours minus one in the grid. Considering the three objectives, the generated grids create a total of 522 instances with an interesting gradient of difficulty.

7 Basic Empirical Evaluation

All experiments were run on a 64 Core 2.4GHz AMD EPYC with 1TB of RAM. Each process was given a limit of 8GB of memory and allowed a 1-hour timeout. We used Savile Row 1.10.0 [16] with three different backend solvers: kissat 3.1.1 [41], Chuffed 0.12.1 [42] and OR-Tools 9.7.2996 [43]. We also used the Fast Downward 23.06 [44] planner with the blind search A^* heuristic and SymK (rev. c97ce836a) [45] with the default configuration. We considered all planners present in the last International Planning Competition and only 9 claimed to support the features required. Of those, 7 were based on the Fast Downward preprocessor and the other two crashed when given the instances. We also considered SymK due to its native support for axioms.

As per Section 3.2, when not using a planner, for each instance we consider a sequence of decision problems from 1 up to $(width \times height) - \max(goal_b, colours)$ steps, where $goal_b$ is the number of allowed blocks remaining in the goal states and *colours* is the number of different colours in the grid. Note that this takes into account both Lemma 1 and the fact that each shot should remove at least one block from the grid.

In our experimental evaluation, we consider an instance as *solved* when we either find a solution (satisfiable), or prove there is none (unsatisfiable). The longest satisfiable instance solved within the time and memory limits requires 32 steps. As expected, we experimentally observe that for each instance, the

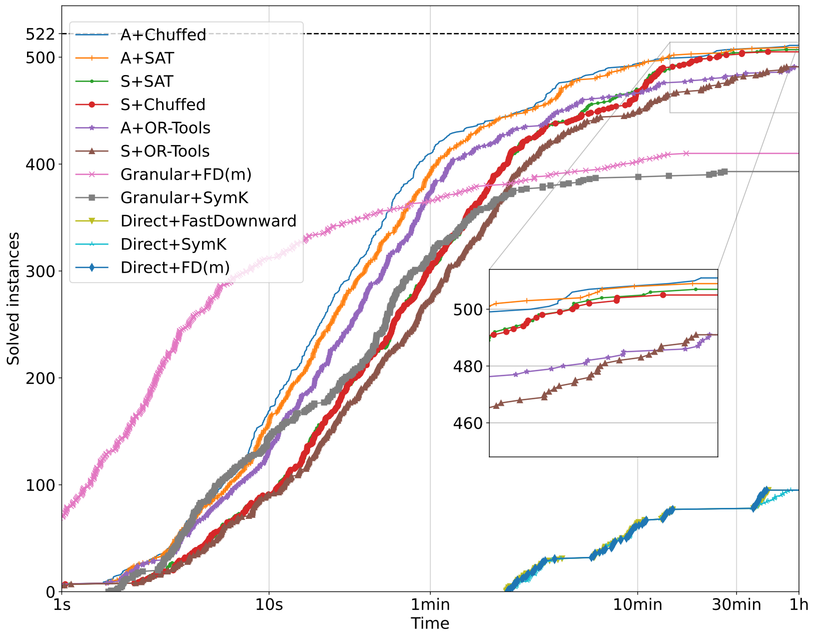


Fig. 5: Cumulative instances solved with each solving approach. Both the state-focused (S) and action-focused (A) models are considered when using Kissat, Chuffed or OR-Tools.

sequence of decision problems becomes incrementally harder until a solution is found.

From now on we will refer to the action-focused model (Section 5.3) as model A, and to the state-focused model (Section 5.4) as model S. On the other hand, we will refer to the direct PDDL model from Section 4.1 as Direct, while the granular model from Section 4.2 as Granular.

In most cases preprocessing by Savile Row is significant. For the instances that took more than 5 minutes to solve, an average of 33% of the total time is spent on preprocessing for the A models, while this proportion rises up to 56% for the S models. Still, for some intermediate steps, Savile Row can prove an instance unsatisfiable before encoding it for the backend solver. In contrast, the instances solved by both Fast Downward and SymK spent over 96% of their time in translating the instance to an intermediate representation for the Direct model. For the Granular model SymK used less than 1%. However, the time for Fast Downward was still dominated by the translation phase. Due these performance problems, we modified Fast Downward to disable the computation of negative axioms. This allowed the planner to also reduce its preprocessing time to less than 1%. From now on, we will refer to this modified Fast Downward as FD(m).

Figure 5 shows a cumulative mortality plot, displaying the total number of instances solved within the previously described per-instance resource

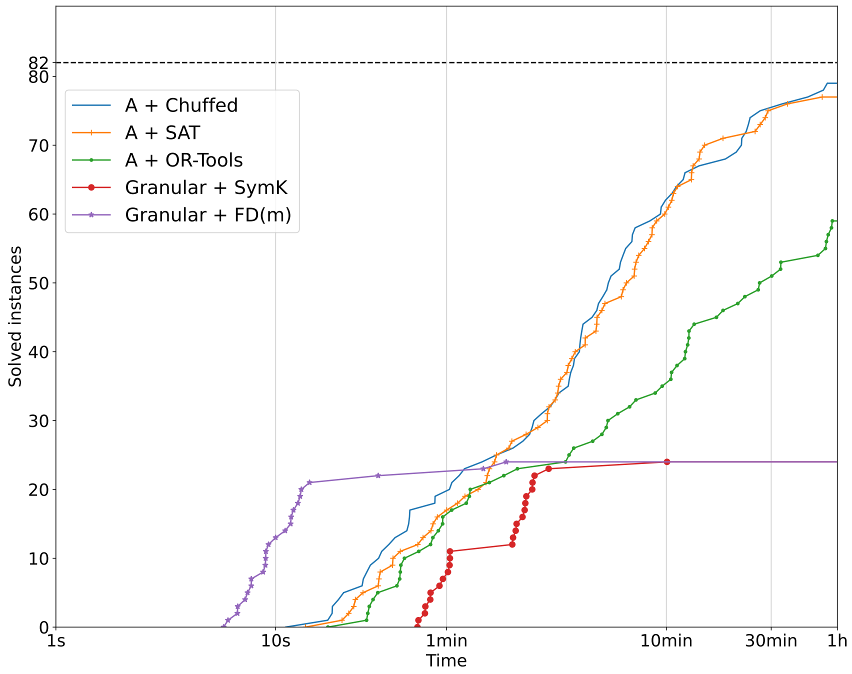


Fig. 6: Cumulative instances solved with each solving approach on the subset of largest instances.

bound of 8GB and 1-hour timeout, summarising the performance of these two CP and two PDDL models with each of the respective solvers on our set of instances.This figure illustrates that the three CP solvers all perform reasonably well, being able to solve most instances within the given time. In contrast, the Direct encoding exhibits a significantly worse performance scaling with both planners, only solving the smallest instances (96 in total) and getting stuck during the grounding process. Moreover, in every instance when the planners time out or run out of memory, they do so during grounding. Model A scales much better in terms of memory usage when compared to model S. Similarly, the Granular model scales better than the Direct model. More concretely, no solver runs out of memory with model A. When using model S, SAT runs out of memory in 14 instances, Chuffed in 24 and OR-Tools in 20. Both planners run out of memory in 234 instances with the Direct model, while with the Granular model Fast Downward runs out of memory with 111 but only 11 with SymK.

Similarly to Figure 5, Figure 6 plots the cumulative instances solved when focusing only on the biggest instances, composed of all grids of sizes 6×5, 5×6 and 7×7. It can be seen that given enough time, the CP approaches are able to scale and solve most of them. The alternative Granular model is able to scale much better than the Direct one, mitigating most of the problems with the Direct model. Still, it seems it is not able to further scale as no instance is solved between the 10 minute and 1 hour mark.

As we have shown, we can use classical planning to solve Plotting despite the PDDL limitations highlighted in Section 3.4. Over the last few decades, the prevalent method of solving classical planning problems has been heuristic search [46]. In such approaches, a grounded representation of the problem is generally needed to be able to compute heuristic values that guide the search [47]. The grounding component in most of the planners struggles when presented with the Direct model described in Section 4. More concretely, memory is exhausted due to the generation of large intermediate data structures, which is unavoidable if the expansion phase of grounding is performed before any pruning of the intermediate expressions. Plotting when expressed via the Direct model therefore appears to be a *hard-to-ground problem* [11].

Such problems with grounding could be mitigated by using a more expressive language, like the ones proposed in [35, 36], allowing more concise and efficient problem representation. However one may need as well to deal with the grounding of this richer language to be able to use similar solving methods. Recent and promising developments [11] in the Automated Planning community adapt the heuristic search framework to allow search in the non-grounded representation of the problem. Yet, those approaches are still too limited in their expressivity to be able to reason with essential constructs needed for Plotting, such as conditional effects and quantifiers. Our Granular model sidesteps these issues by using a lower level of abstraction which is easier to ground. Expressing the planning problem in a higher-level formalism such as Hierarchical Domain Description Language (HDDL) [48] (solved with a planner that implements a lifted planning method) may also avoid excessive grounding, although we leave this for future work.

In Table 1 we provide a summarized performance comparison using the PAR2 score for the proposed solvers and models. The PAR2 score[3] (sometimes referred to as the PAR-2 score) is equal to the CPU time of the solver when the instance is solved, and 2 times the timeout when the instance is unsolved for any reason. As we can see, the overall best approach is Chuffed with model A, which is able to solve 512 out of the 522 instances within the given time. Moreover, it is also slightly faster than the other approaches when solving these instances. SAT is very close, solving 510 instances. Finally OR-Tools has a noticeably poorer performance, being able to solve only 492 instances. When considering Model S we can see that these performance differences change. SAT is the winner with 508 instances, while Chuffed solves 506 instances and OR-Tools significantly less at 492. SAT and Chuffed have very similar PAR2 scores, both being substantially better than OR-Tools. Intuitively, this means that OR-Tools in general is slower when solving. These PAR2 differences can be seen in Figure 5 as the distance between the respective curves. Our best planning approach, using FD(m) with the Granular model, solves 411 instances.

[3]This score is broadly used in benchmarking, in particular this has been the standard score in recent SAT competitions.

Model A			Model S		
Solver	Instances	PAR2	Solver	Instances	PAR2
SAT	510	132818	SAT	**508**	**175025**
Chuffed	**512**	**121366**	Chuffed	506	187425
OR-Tools	492	278864	OR-Tools	492	309046

Direct PDDL Model			Granular PDDL Model		
Solver	Instances	PAR2	Solver	Instances	PAR2
FD	96	3142206	FD	0	3758400
SymK	96	3147702	SymK	394	943864
FD(m)	96	**3141101**	FD(m)	**411**	**815435**

Table 1: For each solving approach and model, the number of instances solved within the limits and their PAR2 Score.

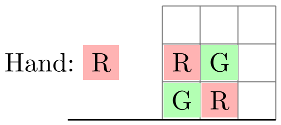
8 Evaluation of Model Enhancements

One of the advantages of using Essence Prime over PDDL is that additional information can be straightforwardly added to the model, such as implied or symmetry breaking constraints. This section discusses a list of such improvements to the Essence Prime models. Note that the constraints in this section are applicable to both models (Sections 5.3 and 5.4) as they share the same viewpoint. Finally, an experimental evaluation about how they affect the overall performance is presented.

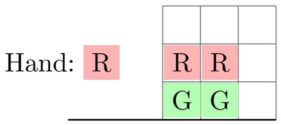
8.1 Symmetry Breaking

Since they do not interfere with each other in terms of the grid state, it is tempting to think that we can freely permute a sequence of consecutive column shots. This is to ignore the state of the hand, however. Consider Figure 7a: we can shoot down the left column, resulting in a green block in the hand, followed by the right column – but not vice versa. If the column "prefix" is the same, as per Figure 7b, we can now shoot down either column. However, after one such shot we could not immediately fire down the other column because the hand would now contain a green block. Therefore, there can be no consecutive column shots (with this pair of columns) to permute. If, however, the columns are monochrome, consecutive column shots are possible, and so we can insist that they are ordered:

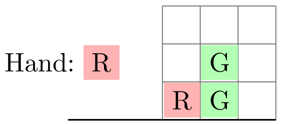
```
$ Symmetry breaking constraint
forAll step : int(1..noSteps-1) .
  forAll gCol : int(1..gridWidth-1) .
    forAll gCol2 : int(gCol+1..gridWidth) .
     $ Monochrome
     (forAll gRow : int(1..gridHeight) .
       ((grid[step-1,gRow,gCol] = EMPTY) \/
        (grid[step-1,gRow,gCol] = hand[step-1])) /\
       ((grid[step-1,gRow,gCol2] = EMPTY) \/
        (grid[step-1,gRow,gCol2] = hand[step-1])))
     -> ( $ If consecutive must be left to right
       fpCol[step] = gCol2 -> fpCol[step+1] != gCol),
```



(a) A state with non-interchangeable column shots



(b) A state with non-interchangeable column shots



(c) A state that can only lead to dead ends.

Fig. 7: Illustrative Plotting game situations.

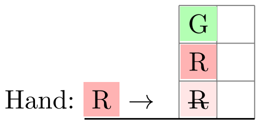
8.2 Implied Constraints

Consider an arbitrary grid with one red block. If that red block is transferred to the avatar's hand then there is no possible move, we find ourselves in a dead-end. Hence, this state is only permissible following the final shot in the sequence. If red is already in the hand then the next move must shoot at the red block in the grid, again resulting in another colour in the hand and one red block in the grid, except in a situation like Figure 7c, where we could shoot down the first column, consume the red block and keep red in the hand. Again, however, there will be no possible move. So, the implied constraint is: given a single block of colour c in the grid at time step t, then colour c cannot be in the hand until the goal state (when no further shots are necessary):

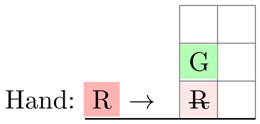
```
$ Dead-end implied constraint
forAll step : int(0..noSteps-2) .
  forAll colour : COLOURS .
    atmost(flatten(grid[step,..,..]), [1], [colour]) ->
      forAll step2 : int(step+1..noSteps-1) . hand[step2] != colour,
```

It might be conjectured that a similar condition holds for two blocks of a particular colour remaining. Consider an arbitrary grid with two red blocks. When one is hit, having consumed a block of another colour, it appears in the hand. The next shot must be at the other red block. That seems to suggest that red can appear at most once in the hand in the remainder of the sequence. Consider, however, Figure 8a. If we shoot on the bottom row the red block is consumed and the shot block hits the wall, rebounding into the hand, resulting in Figure 8b. Similarly, if we again shoot on the bottom row, the result is Figure 8c. Hence, a counterexample: red appears twice in the hand when there are only two blocks in the grid.

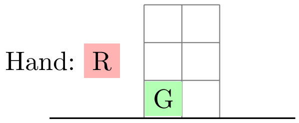
In general, if there are k red blocks in the grid then there can be at most $k-1$ occurrences of red in the hand subsequently before the goal state: if there



(a) State 1



(b) State 2



(c) State 3

Fig. 8: With two red blocks remaining, red can appear in the hand twice.

are k occurrences of red in the hand before the goal state then there will be nothing to shoot at upon the kth occurrence. However, we experiment only with the case where $k = 1$, which is the most straightforward to detect and exploit.

Lemma 1 leads naturally to a set of implied constraints. Each colour in the initial grid must remain in either the grid or the hand at every time step throughout the game. A global cardinality constraint (GCC) could be used here, however GCC would be decomposed for Chuffed, OR-Tools, and SAT backends. Instead, we chose the relatively simple formulation below.

```
$ Colours are preserved: implied constraint
forAll colour : COLOURS .
  (exists r : GRIDROWS .
    exists c : GRIDCOLS . (initGrid[r, c]=colour)) ->
      (forAll step : STEPSFROM1 .
          atleast(flatten([[hand[step]],
                  flatten(grid[step,..,..])]), [1], [colour])),
```

We evaluated the colour preservation implied constraint in a preliminary experiment with a set of 20 6×6 initial grids with either 3 or 4 colours (all such grids from [49]). We fixed the number of timesteps to each value from 10 to 19, producing 200 Essence Prime instances. We used Chuffed with the action-focused model (with the dead-end implied constraint above, and with the progress constraint removed as described in Section 9). We found that the colour preservation implied constraint slowed solving down on average and increased the number of timeouts (at one hour). Therefore we disregard the colour preservation implied constraint from here on.

8.3 Dominance Breaking

It is natural to conjecture whether we might identify and exploit *dominance* relations [50] in Plotting. In this context a dominance relation translates to a situation in the game where, given two possible moves, we can rule out the selection of one of them safe in the knowledge that if there is a path to a solution through either move then there is a path through the remaining choice.

A simple example of a dominance relation in Plotting is to recognise that shooting along an empty row has the same effect as shooting down the last column. These two actions are interchangeable, so we can disallow the former:

```
$ Empty row dominance constraint
forAll step : STEPSFROM1 .
  $ Assume bottom row not going to be empty.
  forAll gRow : int(1..gridHeight-1) .
    ((sum gCol : int(1..gridWidth) .
        grid[step-1,gRow,gCol]) = 0) ->
            (fpRow[step] != gRow),
```

This remains true if the row is empty except for the last column, and the block in the last column on that row has nothing above it:

```
forAll step : STEPSFROM1 .
  $ Assume bottom row not going to be empty.
  forAll gRow : int(1..gridHeight-1) .
```

```
    ((sum gCol : int(1..gridWidth-1) .
        grid[step-1,gRow,gCol]) = 0) /\
        ((gRow = 1) \/ (grid[step-1,gRow-1,gridWidth] = EMPTY))
        ->
        (fpRow[step] != gRow),
```

We might further conjecture that a dominance relation holds between certain pairs of column shots. For example, if there are two identical monochrome columns can we exclude either the left or the right? The fact that row shots may be compromised by making either of these choices presents some doubt about the validity of this conjectured dominance relation. Rather than searching for counterexamples by hand, however, we can automate this search by adapting our instance generation model given in Section 6. Specifically, we constrain the initial grids generated to exhibit the situation conjectured to feature a dominance relation:

```
given gridWidth : int (1..)
given gridHeight : int (1..)
letting GRIDCOLS be domain int(1..gridWidth)
letting GRIDROWS be domain int(1..gridHeight)

given noColours : int (1..)
letting COLOURS be domain int(1..noColours)

find initGrid : matrix [GRIDROWS, GRIDCOLS] of COLOURS
find firstColourOccurrenceRow : matrix [COLOURS] of GRIDROWS
find firstColourOccurrenceCol : matrix [COLOURS] of GRIDCOLS
find col1 : GRIDCOLS
find col2 : GRIDCOLS

such that

$ Use a row-wise, left-to-right ordering to break symmetry
col1 < col2,
forAll row : GRIDROWS .
  initGrid[row,col1] = initGrid[row,col2],
forAll row : int(1..gridHeight-1) .
  initGrid[row,col1] = initGrid[row+1,col1],
```

We search for counterexamples to our conjecture as follows: we iterate over the solutions to the above model for a given grid size, and from 2 to 4 colours, based on the assumption that 4 colours should be sufficient to exhibit a counterexample. For each such initial grid we then iterate, in ascending order over the number of goal blocks remaining and find the minimum number of steps required to solve the resulting instance. We can then test our dominance conjecture by solving the same instance twice more: first excluding from consideration for the first action the left of the two monochrome columns, and again but this time excluding the right monochrome column. If either is unsatisfiable we have a counterexample proving that the dominance-breaking constraint to exclude the left or the right such monochrome column is unsound.

As might be expected given the intricacies of the game, the conjectured dominance-breaking constraints are unsound. Two small counterexamples were found by the above procedure, as presented in Figures 9 and 10.

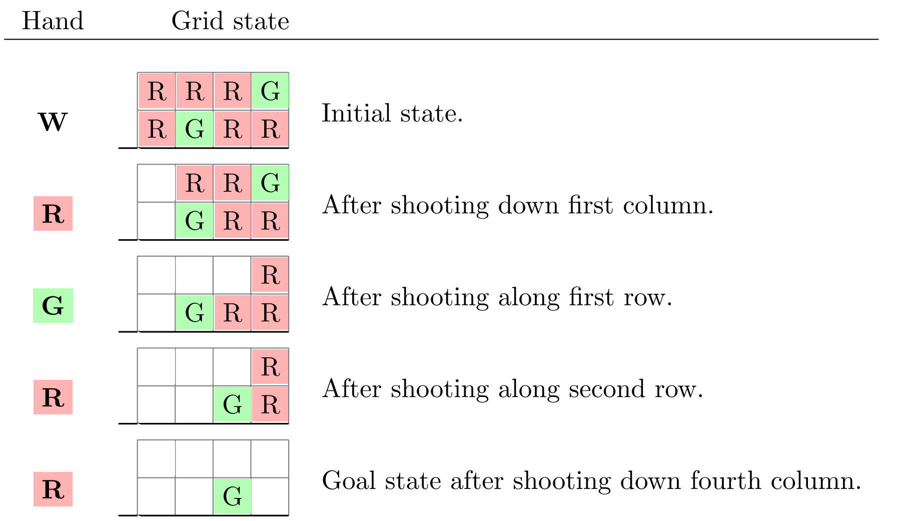

Hand
Grid state
W
R R R G
R G R R
Initial state.
R
R R G
G R R
After shooting down first column.
G
R
G R R
After shooting along first row.
R
R
G R
After shooting along second row.
R
G
Goal state after shooting down fourth column.

Fig. 9: Counterexample to the conjectured dominance-breaking constraint that we can arbitrarily disallow shooting down the left of two identical monochrome columns. Presented is a four-step solution to an instance to reach a goal of 1 block remaining, but it cannot be solved if we disallow firing down the first column as the first step.

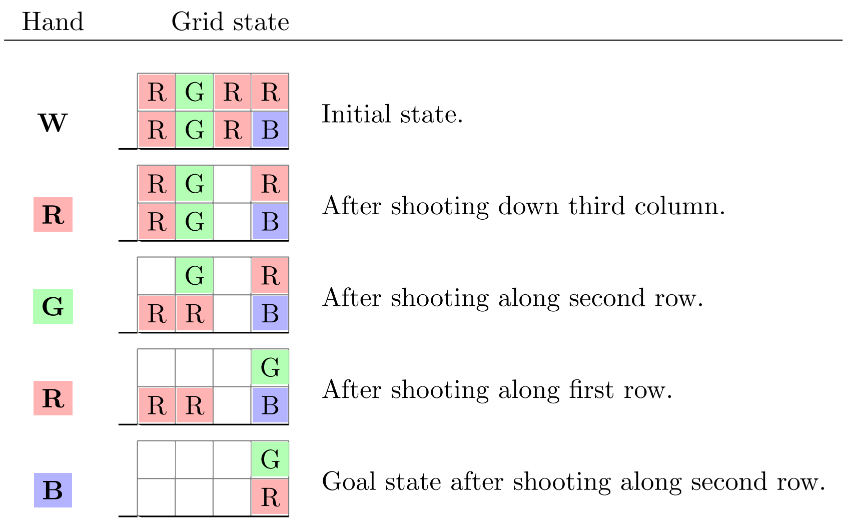

Hand
Grid state
W
R G R R
R G R B
Initial state.
R
R G R
R G B
After shooting down third column.
G
G R
R R B
After shooting along second row.
R
G
R R B
After shooting along first row.
B
G
R
Goal state after shooting along second row.

Fig. 10: Counterexample to the conjectured dominance-breaking constraint that we can arbitrarily disallow shooting down the right of two identical monochrome columns. Presented is a four-step solution to an instance to reach a goal of 2 blocks remaining, but it cannot be solved if we disallow firing down the third column as the first step.

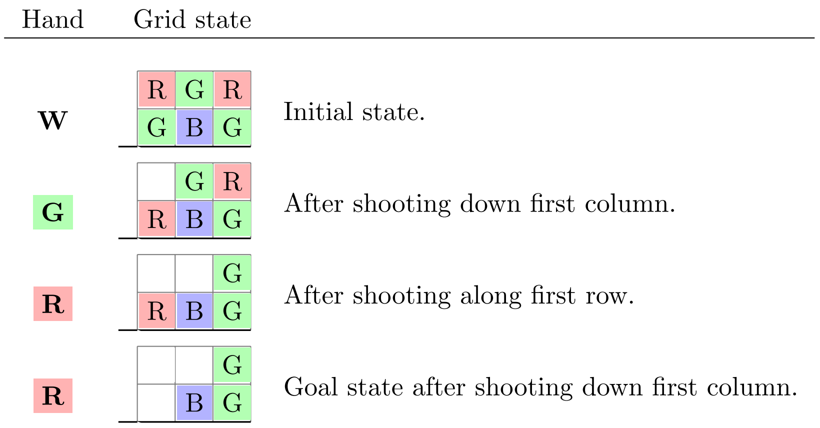

Hand
Grid state
W
R G R
G B G
Initial state.
G
G R
R B G
After shooting down first column.
R
G
R B G
After shooting along first row.
R
G
B G
Goal state after shooting down first column.

Fig. 11: Counterexample to the conjectured dominance-breaking constraint that we can arbitrarily disallow shooting down the left of two columns with an identical prefix: a sequence of one colour terminated by a single, different colour. Presented is a three-step solution to an instance to reach a goal of 3 blocks remaining, but it cannot be solved if we disallow firing down the first column as the first step.

A similar conjecture might be made about columns with the same *prefix*: an identical sequence of the same colour terminated with a block of another colour. Such instances can be generated by modifying the instance generator model to contain the following set of constraints, to replace the requirement for a pair of monochrome columns:

```
col1 < col2,
exists row : int(1..gridHeight) .
  $ Equal down to row
  (forAll row2 : int(1..row) .
    initGrid[row2,col1] = initGrid[row2,col2]) /\
  $ monochrome down to row
  (forAll row2 : int(1..row-2) .
    initGrid[row2,col1] = initGrid[row2+1,col1]) /\
  $ Bottom row is different from rows above
  initGrid[row,col1] != initGrid[row-1,col1]
```

Again, our search finds two small counterexamples to demonstrate that this conjecture is false, as presented in Figures 11 and 12.

In summary, by exploiting our instance generation model to capture situations in which dominances might occur we can use our Plotting models to automatically test dominance conjectures effectively, rather than laboriously look for counterexamples by hand.

8.4 Evaluation

We have now discussed possible improvements of the base models with additional information such as implied or symmetry breaking constraints. In this

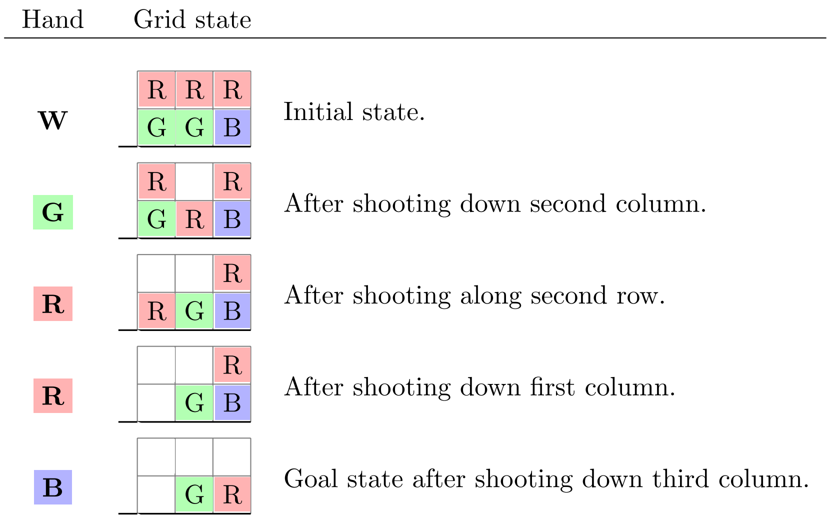

Hand
Grid state
W
R R R
G G B
Initial state.
G
R R
G R B
After shooting down second column.
R
R
R G B
After shooting along second row.
R
R
G B
After shooting down first column.
B
G R
Goal state after shooting down third column.

Fig. 12: Counterexample to the conjectured dominance-breaking constraint that we can arbitrarily disallow shooting down the right of two columns with an identical prefix: a sequence of one colour terminated by a single, different colour. Presented is a four-step solution to an instance to reach a goal of 2 blocks remaining, but it cannot be solved if we disallow firing down the second column as the first step.

section we evaluate three of them. More concretely, we analyse the early *dead-end* detection in Section 8.2, the *empty* row dominance constraint in Section 8.3 and the symmetry breaking constraint of *monochrome* columns from Section 8.1.

Table 2 shows how each of the considered additional constraints affects the performance of each solver. As a reference, we add the *none* column which shows each solving approach with no additional constraints, a column for each additional constraint in isolation and the *all* column which shows the effect of considering all of them together. We can see that SAT and Chuffed are in general indifferent or slightly hindered by them, as the number of solved instances range between +1 and -3. When considering the best model for both solvers, the already small difference between them becomes negligible. More interestingly, OR-Tools notably benefits from the additional constraints, solving up to 5 extra instances and a 12.52% improvement in the PAR2 score.

A model improvement that has been considered from the beginning has been a custom variable search order. That is, both Essence Prime models include a `branching on` statement that specifies a search order for the problem variables. More concretely, it forces decisions to be done in a chronological order: for any given step t, all variables in t are assigned before assigning variables in $t + 1$. Both SAT and Chuffed use a Variable State Independent Decaying Sum (VSIDS) branching heuristic by default. Still, Chuffed is able to

Model A

	Solver	none	dead-end	empty	monochrome	all
Solved	SAT	510	**0**	0	0	-1
Instances	Chuffed	**512**	-2	0	-1	-3
	OR-Tools	492	-10	+3	**+5**	-10
PAR2	SAT	132818	**-0.03%**	+3.06%	+1.90%	+10.02%
Score	Chuffed	**121366**	+10.78%	+1.73%	+5.70%	+15.32%
	OR-Tools	278864	+20.10%	-6.96%	**-12.52%**	+20.00%

Model S

	Solver	none	dead-end	empty	monochrome	all
Solved	SAT	508	0	+1	**+1**	-1
Instances	Chuffed	506	**+1**	0	+1	+1
	OR-Tools	492	0	**+2**	+1	+1
PAR2	SAT	175025	+0.36%	-0.88%	**-2.10%**	+4.39%
Score	Chuffed	187425	**-3.01%**	+2.24%	-2.02%	+0.32%
	OR-Tools	309046	-0.50%	**-3.51%**	-0.74%	-0.94%

Table 2: Instances solved and PAR2 score per solver and model variation. Column *none* shows the performance without the extra constraints. Columns *dead-end*, *empty* and *monochrome* show the differences in performance when considering the constraints from Sections 8.2, 8.3 and 8.1 respectively. Column *all* shows their combined effect. A decreasing value for the PAR2 score signals that problems are solved faster, and so a negative value is better.

Removal of the custom search strategy in Chuffed

	Model	none	dead-end	empty	monochrome	all
Solved	A	512(-6)	510(-5)	512(-7)	511(-5)	509(-5)
Instances	S	506(-6)	507(-7)	506(-7)	507(-7)	507(-7)
PAR2	A	+34.71%	+29.38%	+38.15%	+29.83%	+30.36%
Score	S	+26.56%	+30.66%	+28.13%	+29.39%	+29.3%

Table 3: Summary of how detrimental it is to remove the custom search order for Chuffed. The first two rows show, for each model and variation, the number of solved instances when removing the custom search order; and in parentheses the difference from when the search order is specified. The last two rows show the differences in the PAR2 score.

take into account our user-defined branching heuristic by alternating it with VSIDS during search. To quantify its effectiveness, Table 3 summarises the effect of removing the `branching on` statement, in which case the solver will alternate between the default variable ordering (variable declaration order in the Essence Prime model) and VSIDS. Across all the models and variations, we see a clear reduction of performance, losing between 5 and 7 instances and increasing solving times between 26.56% and 38.15%. Figure 13 visually depicts in a cumulative mortality plot the best final model for each solver. Finally, all the additional considered constraints do not fundamentally change the trends in memory usage that were discussed in Section 7.

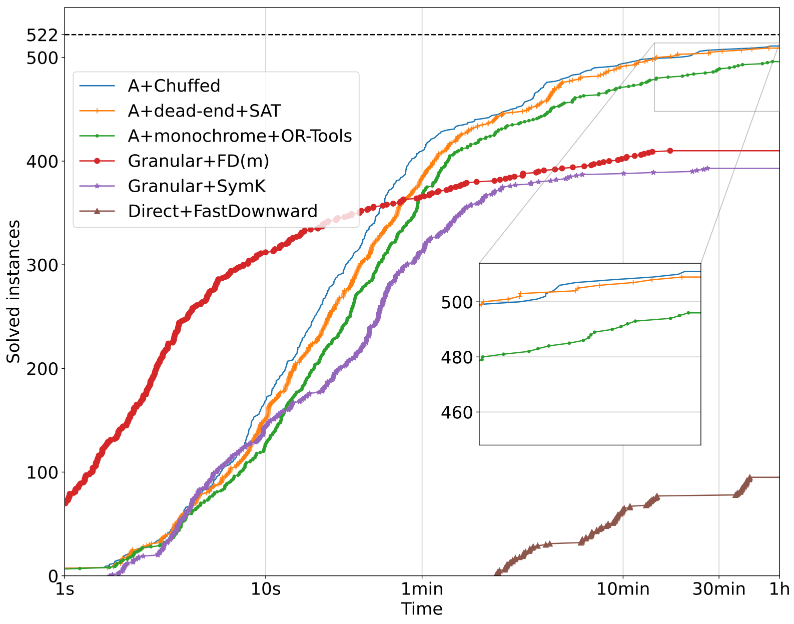


Fig. 13: Cumulative mortality plot for the best combination for each model and solver.

9 Fine-Tuning the SAT Encoding

We have shown that a SAT encoding approach (combined with the Kissat solver) is very effective and is competitive with the best alternative. Kissat is able to solve 510 instances within the 1 hour time limit, whereas the winning solver (Chuffed) solves 512. However, the models are not optimised for SAT encoding, and also the default encodings of Savile Row are used and these may not be the best choices for the problem class. In this section we investigate the SAT encoding and propose some improvements. As a starting point we use the model A+dead-end, shown in Table 2 to be the most effective model.

We examined the SAT encoding produced by Savile Row for a set of 20 6×6 initial grids with either 3 or 4 colours (all such grids from [49]). We fixed the number of timesteps to each value from 10 to 19, producing 200 Essence Prime instances (as used in Section 8.2). First, we discovered that approximately half of the SAT encoding of each instance (over 62% of the clauses and over 44% of the variables) represents one set of constraints: the progress constraints (described in Section 5.2). Second, encodings of reified integer equality constraints (i.e. $b \leftrightarrow (x = y)$) also make up a large part of the instance encodings. In this case, reified equality constraints arise from matrix indexing (specifically where a matrix of decision variables is indexed by an expression containing one or more decision variables). Matrix indexing is essential for concise expression of the actions. In the following subsections we propose more compact SAT encodings for the progress constraints and matrix indexing. The improved encodings lead to a modest improvement in the performance of Kissat, allowing it to slightly outperform Chuffed.

9.1 Progress Constraints

The progress constraints rule out actions that make no progress towards the goal. The version below (henceforth version 1) of the progress constraint directly sums the grid (i.e. sums the colours of all blocks in the grid) rather than the number of non-empty squares in the grid.

```
$ Progress constraint: version 1
forAll step : STEPSFROM1 .
  sum(flatten(grid[step-1,..,..])) > sum(flatten(grid[step,..,..])),
```

The correctness of this version of the progress constraint is not entirely straightforward, it relies on details of the actions in Plotting (specifically that the block added to the grid (from the hand) has the same colour as the first deleted block, and at least two blocks will be removed from the grid). The advantage of this formulation for CP solvers is that no auxiliary variables are required, each progress constraint is simply a linear integer constraint on the original variables. However, version 1 is poor for encoding to SAT because the set of values that the sums can take is unnecessarily large. Therefore we propose a second version that simply adds up the number of non-empty squares. It is not semantically identical to version 1, but it has the same effect in context. Version 2 is shown below.

```
$ Progress constraint: version 2
forAll step : STEPSFROM1 .
  sum([grid[step-1, i, j]!=0 | i : GRIDROWS, j: GRIDCOLS]) >
           sum([grid[step, i, j]!=0 | i : GRIDROWS, j: GRIDCOLS]),
```

We also propose a third version that avoids arithmetic entirely, shown below. It states that there exists at least one location in the grid that becomes empty, a weaker condition than versions 1 and 2. However we expect the encoding of version 3 to be smaller and more efficient to unit-propagate than the others.

```
$ Progress constraint: version 3
forAll step : STEPSFROM1.
  or([ grid[step-1, i,j]!=EMPTY /\ grid[step, i,j]=EMPTY
     | i : GRIDROWS, j : GRIDCOLS]),
```

Finally we note that the progress constraint (when used in an action-based model) is an implied constraint, i.e. it does not affect the set of solutions. The preconditions of the actions (combined with other constraints on `fpCol` and `fpRow` that require an action to occur at each timestep) ensure that progress will be made at each timestep in an action-based model. Version 4 of the model is simply SAT+A+dead-end without the progress constraint.

9.2 Matrix Indexing and Half-Reification

Indexing matrices with decision variables is pervasive in both models. For example, consider the following expression taken from the action-focused model.

```
forAll step : STEPSFROM1 .  ...
   (forAll above : int(1..row-1) .  ...
     grid[step-1,above,fpCol[step]] = hand[step-1]) ...
```

In each concrete constraint derived from this expression, `step` and `above` are constants so a column is extracted from the `grid` matrix to be indexed by `fpCol[step]`. The expression takes the form $M[x] = y$ (where x and y are integer variables and M is a one-dimensional matrix of variables). When the target is SAT, Savile Row decomposes $M[x] = y$ as follows (assuming the index of M and domain of x are both $\{1 \dots n\}$).[4]

$$\forall i \in \{1 \dots n\} : (x = i) \rightarrow (M[i] = y) \tag{1}$$

Until now, Savile Row has had two types of encoding for constraints that are nested in logic expressions (*nested* constraints): single literal or reified. The expression $x = i$ is encoded with a single SAT literal. $M[i] = y$ has no single literal encoding so the reified encoding must be used with a new Boolean variable b_i, as follows.

$$\forall i \in \{1 \dots n\} : (x = i) \rightarrow b_i \;\wedge\; b_i \leftrightarrow (M[i] = y) \tag{2}$$

Half-reified constraints (also called reifyimplied constraints) are commonly used in constraint solvers and modelling languages to simplify models and improve solver efficiency [52, 53]. In the literature, the half-reified form of a constraint C is usually defined as $b \rightarrow C$ where b is a Boolean variable. We apply the same idea to SAT encoding by adding half-reification as a third type of encoding for nested constraints. Here, b may be a Boolean variable, a negated Boolean variable, or an expression with a single literal encoding (i.e. $(x\#c)$ for any decision variable x, constant c, and comparator $\# \in \{=, \neq, \leq, >\}$). The context of a nested constraint determines whether half-reification can be used in place of full reification: in this respect our implementation is very similar to Feydy et al [53].

An encoding of $b \rightarrow C$ is generated by adding the literal $\neg b$ to each clause of the encoding of C. Note that any auxiliary SAT variables produced when encoding C will be unconstrained when b is false. With half-reification enabled, Savile Row encodes each implication of Equation (1) directly without needing to introduce the b_i variables in Equation (2). As an example, suppose M has length 10 and all variables have domain $\{1 \dots 10\}$. Without half-reification, $M[x] = y$ is encoded with 310 clauses and 10 additional SAT variables. With half-reification the same constraint is encoded with 200 clauses and no additional variables.

[4]Details of the handling of undefinedness [51] are not relevant here and have been removed.

Finally, it is worth considering whether the default decomposition in Equation (1) strikes a good balance between size and propagation strength. The SAT encoding of Equation (1) is not GAC (i.e. unit propagation on the encoding will not enforce GAC on the original constraint). We have implemented a GAC encoding based on the idea of support [54]. The encoding has three parts to enforce GAC on x, y, and M respectively. For example, a variable $M[i]$ and domain value d must be pruned when $x = i$ and $y \neq d$, and this is achieved with the following set of clauses.

$$\forall i \in \{1 \dots n\} : d \in D(M[i]) : (M[i] \neq d) \vee (x \neq i) \vee (y = d) \tag{3}$$

We did a preliminary experiment using the same set of 200 instances described above, with the model SAT+A+dead-end. Half-reification was enabled and the progress constraints were removed (i.e. version 4). The GAC encoding takes 73% longer to solve and has 38% more clauses (mean averages). It seems that the GAC encoding is not worthwhile in this case. We use the non-GAC encoding (Equation (1)) for all experiments other than this one.

9.3 Evaluation

We compare encoding size and solver performance with the four versions of the progress constraints, named v1 to v4, and v4 with half-reification enabled. Table 4 shows the performance summarised by PAR2 score and instances solved within the time limit. By these measures, there is a noticeable improvement in the SAT solver's PAR2 time from v1 to v4, and a further improvement when we add half-reification to v4. The improvements does not allow Kissat to solve more instances than the baseline model in the given timeout. Still, the PAR2 scores of the best SAT and Chuffed model surprisingly only differ by 70 seconds.

Figure 14 plots the number of SAT variables and the number of clauses for each instance, comparing v1 (A+dead-end) to v4+half-reification. The reductions in encoding size (particularly the number of clauses) are very substantial. This is particularly apparent on the largest instances where the number of clauses is reduced by approximately 8 times. However, Kissat is able to perform well on v1 despite its size, and the gains in PAR2 time do not match the reductions in encoding size. To reiterate: our best-performing SAT model overall is v4 with half-reification enabled.

10 Level Design

We can use our instance generation procedure and our models of Plotting for the curation of an interesting set of levels for human players. This involves the definition of what we regard as an interesting level, followed by a search, either systematic or through sampling, to identify instances of such levels. For example, we might favour levels that can be solved in a variety of ways: a level

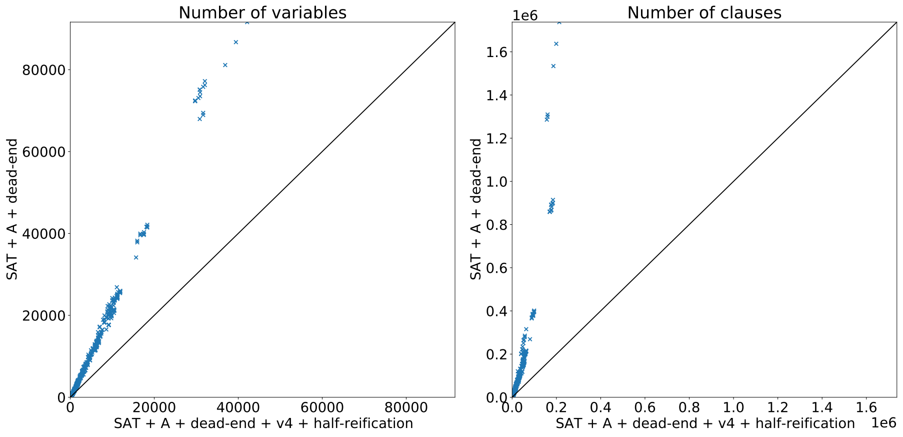


Fig. 14: Comparing number of SAT variables (left) and number of clauses (right) of the models SAT+A+dead-end and SAT+A+dead-end+v4+half-reification.

Performance of the progress constraints

	v2	v3	v4	v4+half-reif
instances solved	510	510	510	510
PAR2 score	123322	121713	121780	121436

Relative performance from the base model

	v2	v3	v4	v4+half-reif
Instances solved	+0	+0	+0	+0
PAR2 score	-9.93%	-11.11%	-11.06%	-11.31%

Table 4: Performance of the progress constraints and relative performance from the base v1. This aims to showcase the best overall PAR2, which is SAT + v4 + half-reification

that admits a short solution but does not punish the player for missing it, also containing longer solutions.

As a simple example, we might consider the set of 14 (up to symmetry) initial 2x2 grids with up to 4 colours. One of these, presented as the initial state in Figure 15 best illustrates the characteristics described above. A goal of 1 block remaining can be achieved in a single step by shooting along the first row. It can also be solved in two steps by shooting down the second column first, and then either along the first row or down the first column. However, even if the player chooses to shoot along the second row first all is not lost: a three-step solution is presented in Figure 15.

A general procedure for identifying such levels is to enumerate or sample the possible grids with our instance generation model, then iterate ascending first over the number of steps and, for each, the goal blocks remaining to establish the minimum blocks remaining possible to reach for a given number of steps.

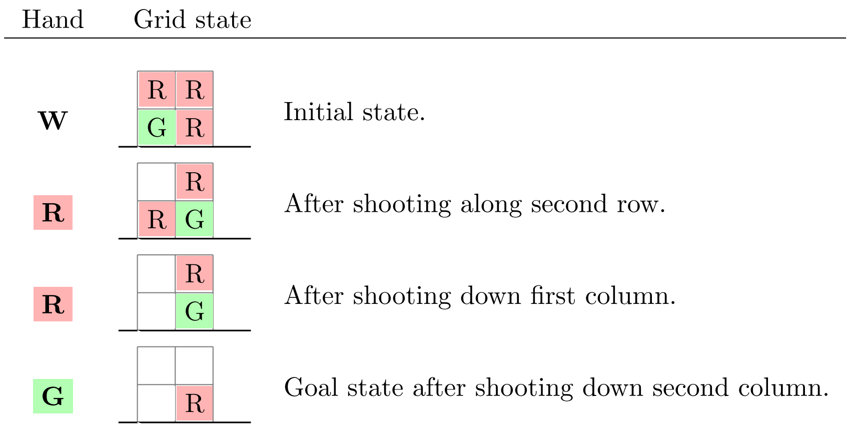

Hand
Grid state
W
R R
G R
Initial state.
R
R
R G
After shooting along second row.
R
R
G
After shooting down first column.
G
R
Goal state after shooting down second column.

Fig. 15: Small instance admitting solutions of 1, 2 and 3 steps for a goal of single block remaining. 3-step solution presented.

We can exploit Lemma 1 to bound the number of goal blocks remaining for which we search. The "interesting" levels are then those such as that presented in Figure 15 that allow several different paths of different lengths to reach the same goal.

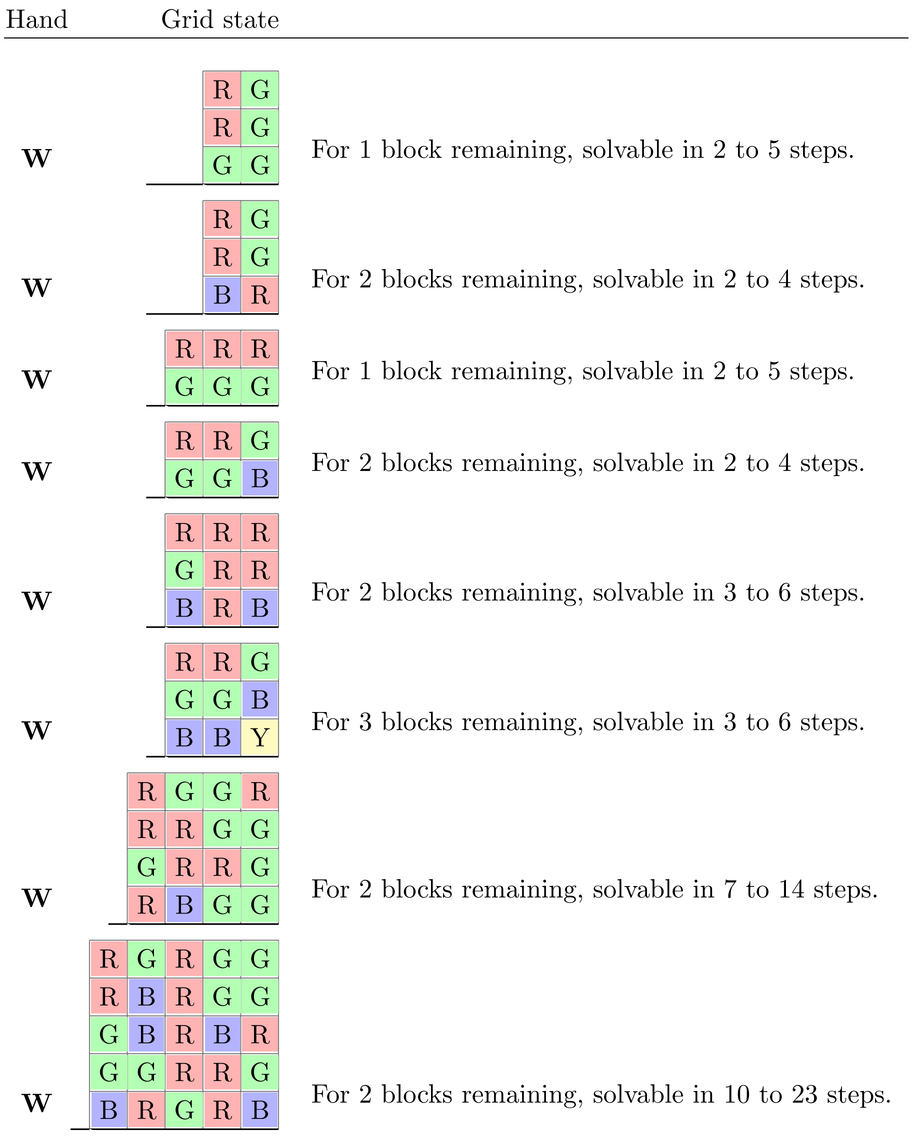
For small grids and numbers of colours, a systematic search is feasible. As a further illustration, in addition to 2×2 grids, we have performed a systematic search up to four colours for: width 2, height 3 (186 grids); width 3, height 2 (186 grids); and width 3, height 3 (11,050 grids). We have also taken 100 samples for 4×4 and 5×5 grids with 3 colours. Interesting instances, according to the criteria above, are presented in Figure 16.

Of course, this is just one example of how we might wish to define an "interesting" instance. There are numerous alternatives, such as the greatest number of steps required to reach the minimum achievable goal blocks remaining. An example is presented in Figure 17.

11 Conclusions and Further Work

Plotting is a deceptively simple puzzle. However, the effects of gravity on the puzzle blocks and the rules governing how a block behaves when it hits the right-hand wall give rise to significant complexity in modelling the problem, as we have shown. We have modelled Plotting as a planning problem using a fragment of PDDL, and as a constraint satisfaction problem using the constraint modelling language ESSENCE PRIME.

Plotting provides an illustrative example of the difficulty of representing complex actions (in this case, actions with complex consequences) in the most commonly implemented fragment of the PDDL AI planning modelling language. We have taken two distinct approaches to modelling Plotting in this fragment of PDDL. In the first approach, PDDL actions correspond closely



Hand	Grid state	
W	R G R G G G	For 1 block remaining, solvable in 2 to 5 steps.
W	R G R G B R	For 2 blocks remaining, solvable in 2 to 4 steps.
W	R R R G G G	For 1 block remaining, solvable in 2 to 5 steps.
W	R R G G G B	For 2 blocks remaining, solvable in 2 to 4 steps.
W	R R R G R R B R B	For 2 blocks remaining, solvable in 3 to 6 steps.
W	R R G G G B B B Y	For 3 blocks remaining, solvable in 3 to 6 steps.
W	R G G R R R G G G R R G R B G G	For 2 blocks remaining, solvable in 7 to 14 steps.
W	R G R G G R B R G G G B R B R G G R R G B R G R B	For 2 blocks remaining, solvable in 10 to 23 steps.

Fig. 16: Selected starting grids with a range of paths to the goal.

to game moves, leading to a cumbersome model that is very challenging to ground prior to solving. As a result, state-of-the-art AI planners cannot cope with more than the smallest instances. In the second approach, one game move is broken down into many relatively simple PDDL actions, easing the problem with grounding and leading to far better performance from the planning systems we evaluated. However, we found that the richer constraint modelling languages fare better, both in permitting a more succinct representation of the

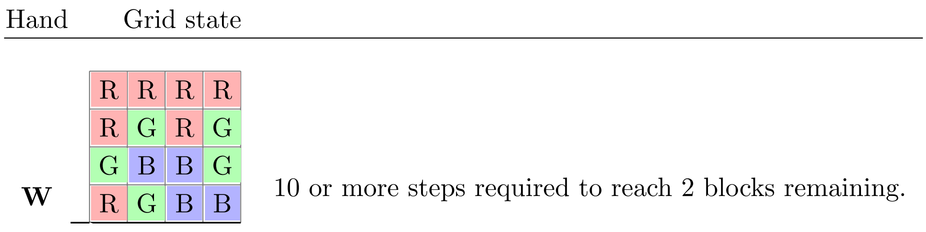


Fig. 17: Example of an instance requiring a long path to the goal.

problem and providing access to solvers that can work directly with the lifted representation. We have presented a detailed study of two different constraint models, their variants, and their performance relative to a number of different solving backends.

Our findings support the view that PDDL should gain support for a richer set of types in order to model more complex actions and state changes such as those found in Plotting. Furthermore, that AI Planning systems should explore ways of avoiding the grounding bottleneck. There has been progress in this direction, for instance the HDDL extension of PDDL for hierarchical planning [48]. Future work also includes extending our constraint models further. For instance, scoring could be added allowing us to optimise on this criterion rather than simply accepting any path to the goal blocks remaining.

Supplementary information. The generated instances, models and variants can be found in https://github.com/stacs-cp/Plotting-Journal

Statements and Declarations. A preliminary version of this study appeared in the proceedings of CP 2022 [49]. The authors declare no competing interests.

References

[1] Ghallab, M., Nau, D., Traverso, P.: Automated Planning: theory and practice. Elsevier, San Francisco, USA (2004). https://doi.org/10.1016/B978-1-55860-856-6.X5000-5

[2] Long, D.: Drilling Down: Planning in the Field. Invited Talk, Twenty-Ninth International Conference on Automated Planning and Scheduling, (ICAPS), Berkeley, California (2019). https://www.youtube.com/watch?v=Zwhnlw118D4

[3] Niemueller, T., Karpas, E., Vaquero, T., Timmons, E.: Planning competition for logistics robots in simulation. In: Proceedings of the 4th Workshop on Planning and Robotics (PlanRob) at the 26th International Conference on Automated Planning and Scheduling (ICAPS), pp. 131–134 (2016). https://web.archive.org/web/20221008151837/https://icaps16.icaps-conference.org/proceedings/planrob16.pdf

[4] Masoumi, A., Antoniazzi, M., Soutchanski, M.: Modeling Organic Chemistry and Planning Organic Synthesis. In: Global Conference on Artificial Intelligence (GCAI), pp. 176–195 (2015). https://doi.org/10.29007/493z

[5] Barták, R., Salido, M.A., Rossi, F.: Constraint satisfaction techniques in planning and scheduling. Journal of Intelligent Manufacturing **21**(1), 5–15 (2010). https://doi.org/10.1007/s10845-008-0203-4

[6] Barták, R., Toropila, D.: Reformulating constraint models for classical planning. In: Wilson, D., Lane, H.C. (eds.) Proceedings of the Twenty-First International Florida Artificial Intelligence Research Society Conference (FLAIRS), pp. 525–530 (2008). https://web.archive.org/web/20220202110048/https://kti.mff.cuni.cz/~bartak/downloads/FLAIRS2008.pdf

[7] Vidal, V., Geffner, H.: Branching and pruning: An optimal temporal POCL planner based on constraint programming. Artificial Intelligence **170**(3), 298–335 (2006). https://doi.org/10.1016/j.artint.2005.08.004

[8] Babaki, B., Pesant, G., Quimper, C.: Solving classical AI planning problems using planning-independent CP modeling and search. In: Harabor, D., Vallati, M. (eds.) Proceedings of the Thirteenth International Symposium on Combinatorial Search (SOCS), pp. 2–10. AAAI Press, Washington, DC (2020). https://doi.org/10.1609/socs.v11i1.18529

[9] Arxer, J.E., Miguel, I., Villaret, M.: CSPLib Problem 088: Plotting. http://www.csplib.org/Problems/prob088

[10] Haslum, P., Lipovetzky, N., Magazzeni, D., Muise, C.: An Introduction to the Planning Domain Definition Language. Synthesis Lectures on Artificial Intelligence and Machine Learning. Springer, Cham, Switzerland (2019). https://doi.org/10.2200/S00900ED2V01Y201902AIM042

[11] Corrêa, A.B., Pommerening, F., Helmert, M., Francès, G.: Lifted Successor Generation Using Query Optimization Techniques. In: Beck, J.C., Buffet, O., Hoffmann, J., Karpas, E., Sohrabi, S. (eds.) Proceedings of the Thirtieth International Conference on Automated Planning and Scheduling (ICAPS), pp. 80–89. AAAI Press, San Francisco (2020). https://doi.org/10.1609/icaps.v30i1.6648

[12] Ridder, B.: Lifted Heuristics: Towards more scalable Planning Systems. PhD thesis, King's College London (2014). https://kclpure.kcl.ac.uk/portal/files/13561312/Studentthesis-Bernardus_Ridder_2014.pdf

[13] Espasa, J., Miguel, I., Coll, J., Villaret, M.: Towards lifted encodings for numeric planning in essence prime. Proceedings of the 18th International Workshop on Constraint Modelling and Reformulation (ModRef) (2019).

https://modref.github.io/papers/ModRef2019_Towards%20Lifted%20Encodings%20for%20Numeric%20Planning%20in%20Essence%20Prime.pdf

[14] van Beek, P., Chen, X.: CPlan: A Constraint Programming Approach to Planning. In: Proceedings of the Sixteenth National Conference on AI and Eleventh Conference on Innovative Applications of AI (AAAI), pp. 585–590 (1999). https://web.archive.org/web/20221125213531/https://www.aaai.org/Library/AAAI/1999/aaai99-083.php

[15] Nightingale, P., Rendl, A.: Essence' Description. arXiv (2016). https://arxiv.org/abs/1601.02865

[16] Nightingale, P., Akgün, Ö., Gent, I.P., Jefferson, C., Miguel, I., Spracklen, P.: Automatically improving constraint models in Savile Row. Artificial Intelligence **251**, 35–61 (2017). https://doi.org/10.1016/j.artint.2017.07.001

[17] Accenture: The Global Gaming Industry Value Now Exceeds $300 Billion, New Accenture Report Finds. https://web.archive.org/web/20230207025112/https://newsroom.accenture.com/news/global-gaming-industry-value-now-exceeds-300-billion-new-accenture-report-finds.htm [accessed 2-Feb-2022] (2021)

[18] Glorian, G., Debesson, A., Yvon-Paliot, S., Simon, L.: The dungeon variations problem using constraint programming. In: Michel, L.D. (ed.) Proceedings of the 27th International Conference on Principles and Practice of Constraint Programming (CP). LIPIcs, vol. 210, pp. 27–12716. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2021). https://doi.org/10.4230/LIPIcs.CP.2021.27

[19] Jefferson, C., Moncur, W., Petrie, K.E.: Combination: Automated generation of puzzles with constraints. In: Proceedings of the 2011 ACM Symposium on Applied Computing (SAC), pp. 907–912 (2011). https://doi.org/10.1145/1982185.1982383

[20] Espasa, J., Gent, I.P., Hoffmann, R., Jefferson, C., McIlree, M.J., Lynch, A.M.: Towards generic explanations for pen and paper puzzles with MUSes. In: Proceedings of the SICSA eXplainable Artifical Intelligence Workshop (2021). https://ceur-ws.org/Vol-2894/short8.pdf

[21] Gent, I.P., Jefferson, C., Kelsey, T., Lynce, I., Miguel, I., Nightingale, P., Smith, B.M., Tarim, S.A.: Search in the patience game 'black hole'. AI Communications **20**(3), 211–226 (2007). https://content.iospress.com/articles/ai-communications/aic405

[22] Jefferson, C., Miguel, A., Miguel, I., Tarim, A.: Modelling and solving

English Peg Solitaire. Comput. Oper. Res. **33**(10), 2935–2959 (2006). https://doi.org/10.1016/j.cor.2005.01.018

[23] Akgün, Ö., Gent, I.P., Jefferson, C., Miguel, I., Nightingale, P., Salamon, A.Z.: Automatic discovery and exploitation of promising subproblems for tabulation. In: Proceedings of the 24th International Conference on Principles and Practice of Constraint Programming (CP). LNCS, vol. 11008, pp. 3–12. Springer, Cham, Switzerland (2018). https://doi.org/10.1007/978-3-319-98334-9_1

[24] Ghallab, M., Nau, D., Traverso, P.: Automated Planning and Acting. Cambridge University Press, Cambridge (2016). https://doi.org/10.1017/CBO9781139583923

[25] Kautz, H.A., Selman, B.: Planning as Satisfiability. In: Proceedings of the 10th European Conference on Artificial Intelligence (ECAI), pp. 359–363 (1992). https://web.archive.org/web/20230209175344/https://henrykautz.com/papers/satplan.pdf

[26] Kautz, H.A., McAllester, D.A., Selman, B.: Encoding plans in propositional logic. In: Proceedings of the Fifth International Conference on Principles of Knowledge Representation and Reasoning (KR), Cambridge, Massachusetts, pp. 374–384. Morgan Kaufmann, San Francisco (1996). https://web.archive.org/web/20230213123050/https://henrykautz.com/papers/plankr96.pdf

[27] Rintanen, J.: Engineering efficient planners with SAT. In: Raedt, L.D., Bessiere, C., Dubois, D., Doherty, P., Frasconi, P., Heintz, F., Lucas, P.J.F. (eds.) Proceedings of the 20th European Conference on Artificial Intelligence (ECAI). Frontiers in Artificial Intelligence and Applications, vol. 242, pp. 684–689. IOS Press, Amsterdam (2012). https://doi.org/10.3233/978-1-61499-098-7-684

[28] Rintanen, J., Heljanko, K., Niemelä, I.: Planning as Satisfiability: Parallel Plans and Algorithms for Plan Search. Artificial Intelligence **170**(12-13), 1031–1080 (2006). https://doi.org/10.1016/j.artint.2006.08.002

[29] Bofill, M., Espasa, J., Villaret, M.: The RANTANPLAN planner: system description. Knowl. Eng. Rev. **31**(5), 452–464 (2016). https://doi.org/10.1017/S0269888916000229

[30] Leofante, F., Giunchiglia, E., Ábrahám, E., Tacchella, A.: Optimal Planning Modulo Theories. In: Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence (IJCAI), pp. 4128–4134 (2020). https://doi.org/10.24963/ijcai.2020/571

[31] Miguel, I., Jarvis, P., Shen, Q.: Flexible Graphplan. In: Horn, W.

(ed.) Proceedings of the 14th European Conference on Artificial Intelligence (ECAI), pp. 506–510. IOS Press, Amsterdam (2000). https://www.frontiersinai.com/ecai/ecai2000/pdf/p0506.pdf

[32] Ghooshchi, N.G., Namazi, M., Newton, M.A.H., Sattar, A.: Encoding domain transitions for constraint-based planning. Journal of Artificial Intelligence Research **58**, 905–966 (2017). https://doi.org/10.1613/jair.5378

[33] Fox, M., Long, D.: PDDL2.1: an extension to PDDL for expressing temporal planning domains. Journal of Artificial Intelligence Research **20**, 61–124 (2003). https://doi.org/10.1613/jair.1129

[34] Geffner, H.: Functional STRIPS: a more flexible language for planning and problem solving. In: Logic-based Artificial Intelligence, pp. 187–209. Springer, Boston, MA (2000). https://doi.org/10.1007/978-1-4615-1567-8_9

[35] Gregory, P., Long, D., Fox, M., Beck, J.C.: Planning modulo theories: Extending the planning paradigm. In: Proceedings of the Twenty-Second International Conference on Automated Planning and Scheduling, ICAPS. AAAI Press, Washington, DC (2012). https://doi.org/10.1609/icaps.v22i1.13505

[36] Francès, G., Geffner, H.: Effective Planning with More Expressive Languages. In: Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence, IJCAI, pp. 4155–4159 (2016). https://www.ijcai.org/Abstract/16/621

[37] Thiébaux, S., Hoffmann, J., Nebel, B.: In defense of PDDL axioms. Artif. Intell. **168**(1-2), 38–69 (2005). https://doi.org/10.1016/J.ARTINT.2005.05.004

[38] Rendl, A., Miguel, I., Gent, I.P., Gregory, P.: Common subexpressions in constraint models of planning problems. In: Proceedings of the Eighth Symposium on Abstraction, Reformulation, and Approximation (SARA), pp. 128–135. AAAI Press, San Francisco (2009). https://web.archive.org/web/20210802163656/https://www.aaai.org/ocs/index.php/SARA/SARA09/paper/view/823/1163

[39] Blum, A., Furst, M.L.: Fast planning through planning graph analysis. Artif. Intell. **90**(1-2), 281–300 (1997). https://doi.org/10.1016/S0004-3702(96)00047-1

[40] Gent, I.P., Jefferson, C., Miguel, I.: Minion: A Fast Scalable Constraint Solver. In: Brewka, G., Coradeschi, S., Perini, A., Traverso, P. (eds.) Proceedings of the 17th European Conference on Artificial Intelligence

(ECAI). Frontiers in Artificial Intelligence and Applications, vol. 141, pp. 98–102. IOS Press, Amsterdam (2006). https://ebooks.iospress.nl/volumearticle/2658

[41] Biere, A., Fazekas, K., Fleury, M., Heisinger, M.: CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020. In: Balyo, T., Froleyks, N., Heule, M., Iser, M., Järvisalo, M., Suda, M. (eds.) Proceedings of SAT Competition 2020 – Solver and Benchmark Descriptions. Department of Computer Science Report Series B, vol. B-2020-1, pp. 50–53 (2020). University of Helsinki. http://hdl.handle.net/10138/318754

[42] Chu, G., Stuckey, P.J., Schutt, A., Ehlers, T., Gange, G., Francis, K.: Chuffed, a lazy clause generation solver (version 0.10.4). https://github.com/chuffed/chuffed (2019)

[43] Perron, L., Furnon, V.: OR-Tools (version 9.7.2996). https://developers.google.com/optimization/

[44] Helmert, M.: The Fast Downward Planning System. J. Artif. Intell. Res. **26**, 191–246 (2006). https://doi.org/10.1613/jair.1705

[45] Speck, D., Geißer, F., Mattmüller, R., Torralba, Á.: Symbolic planning with axioms. In: Benton, J., Lipovetzky, N., Onaindia, E., Smith, D.E., Srivastava, S. (eds.) Proceedings of the Twenty-Ninth International Conference on Automated Planning and Scheduling, ICAPS, pp. 464–472 (2019)

[46] Bonet, B., Geffner, H.: Planning as heuristic search. Artif. Intell. **129**(1-2), 5–33 (2001). https://doi.org/10.1016/S0004-3702(01)00108-4

[47] Helmert, M.: Concise finite-domain representations for PDDL planning tasks. Artif. Intell. **173**(5-6), 503–535 (2009). https://doi.org/10.1016/j.artint.2008.10.013

[48] Höller, D., Behnke, G., Bercher, P., Biundo, S., Fiorino, H., Pellier, D., Alford, R.: HDDL: an extension to PDDL for expressing hierarchical planning problems. In: Proceedings of AAAI, pp. 9883–9891 (2020). https://doi.org/10.1609/AAAI.V34I06.6542

[49] Espasa, J., Miguel, I.J., Villaret, M.: Plotting: a planning problem with complex transitions. In: Proceedings of the 28th International Conference on Principles and Practice of Constraint Programming (CP). LIPIcs, vol. 235, pp. 22–12217. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2022). https://doi.org/10.4230/LIPIcs.CP.2022.22

[50] Prestwich, S., Beck, J.C.: Exploiting dominance in three symmetric problems. In: Fourth International Workshop on Symmetry and Constraint Satisfaction Problems, pp. 63–70 (2004). https://web.archive.org/web/20220121052208/https://tidel.mie.utoronto.ca/pubs/pseudo.pdf

[51] Frisch, A.M., Stuckey, P.J.: The proper treatment of undefinedness in constraint languages. In: Principles and Practice of Constraint Programming-CP 2009: 15th International Conference, CP 2009 Lisbon, Portugal, September 20-24, 2009, pp. 367–382. Springer, Berlin (2009). https://doi.org/10.1007/978-3-642-04244-7_30

[52] Jefferson, C., Moore, N.C., Nightingale, P., Petrie, K.E.: Implementing logical connectives in constraint programming. Artificial Intelligence **174**(16-17), 1407–1429 (2010). https://doi.org/10.1016/j.artint.2010.07.001

[53] Feydy, T., Somogyi, Z., Stuckey, P.J.: Half reification and flattening. In: Proceedings of the 17th International Conference on Principles and Practice of Constraint Programming (CP). LNCS, vol. 6876, pp. 286–301. Springer, Berlin (2011). https://doi.org/10.1007/978-3-642-23786-7_23

[54] Gent, I.P.: Arc consistency in SAT. In: van Harmelen, F. (ed.) Proceedings of the 15th European Conference on Artificial Intelligence (ECAI), pp. 121–125. IOS Press, Amsterdam (2002). https://frontiersinai.com/ecai/ecai2002/p0121.html