Plotting: A Planning Problem With Complex Transitions

Joan Espasa
School of Computer Science, University of St Andrews, Scotland

Ian Miguel
School of Computer Science, University of St Andrews, Scotland

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

Abstract

We focus on a planning problem based on Plotting, a tile-matching puzzle video game published by Taito. The objective of the game is to remove at least a certain number of coloured blocks from a grid by sequentially shooting blocks into the same grid. The interest and difficulty of Plotting is due to the complex transitions after every shot: various blocks are affected directly, while others can be indirectly affected by gravity. We highlight the difficulties and inefficiencies of modelling and solving Plotting using PDDL, the de-facto standard language for AI planners. We also provide two constraint models that are able to capture the inherent complexities of the problem. In addition, we provide a set of benchmark instances, an instance generator and an extensive experimental comparison demonstrating solving performance with SAT, CP, MIP and a state-of-the-art AI planner.

2012 ACM Subject Classification Theory of computation → Constraint and logic programming; Computing methodologies → Planning and scheduling

Keywords and phrases AI Planning, Modelling, Constraint Programming

Digital Object Identifier 10.4230/LIPIcs.CP.2022.23

Supplementary Material https://github.com/stacs-cp/CP2022-Plotting

Funding *Ian Miguel*: funded by EP/V027182/1
Mateu Villaret: funded by grant RTI2018-095609-B-I00 (MCIU/AEI/FEDER, UE).

Acknowledgements This work uses the Cirrus UK National Tier-2 HPC Service at EPCC (`http://www.cirrus.ac.uk`) funded by the University of Edinburgh and EPSRC (EP/P020267/1).

1 Introduction

Automated planning is a fundamental discipline in Artificial Intelligence [14]. 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 [22], logistics [25] or chemistry [23]. Among other techniques, Constraint Programming has been successfully used to solve planning problems [5, 6]. It is especially suited when the problem requires a certain level of expressivity, such as temporal reasoning or optimality [31, 3].

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 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

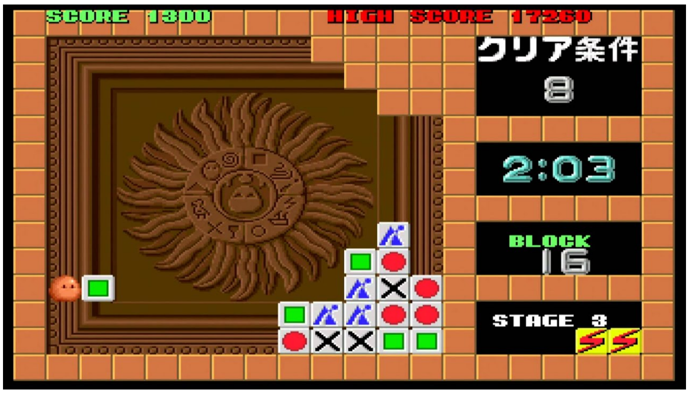


Figure 1 Plotting (Taito, 1989). The avatar is seen on the left, holding a green block. The objective is to reduce the number of blocks in the middle pile up to the goal. In this particular case there are 16 left (see center-right of the image), and the goal is 8 or less (see top-right of image).

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 3. Modelling the complex dynamics of the game in the de-facto standard modelling language for planning problems, PDDL [17], is difficult, as we will demonstrate. The resulting complexity of the model severely hinders the ability of planning systems to produce a valid plan.

Constraint modelling languages can be used to express planning problems [3, 6, 9, 30]. They are richer than PDDL and, while still a challenge to formulate, permit a more concise representation of Plotting. We present two models of the game in Essence Prime [27] and employ Savile Row [26] to transform them into SAT, MIP, and CP instances for solution.

Plotting is also of interest as an example application in the video games industry, which last year was last year valued at over USD 300 billion [1]. 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 [16]. 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 get an impression as to how difficult they are [20]. This paper contributes to this growing effort; 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 Background

A *classical planning problem* is a tuple $\prod = \langle F, A, I, G \rangle$, where: F is a set of propositional state variables, A is a set of actions, I is the initial state and G is the goal. A *state* is a variable-assignment (or valuation) function over state variables F, which maps each variable of F into a truth value. An action $a \in A$ is defined as a tuple $a = \langle Pre_a, E\!f\!f_a \rangle$, where Pre_a refers to the preconditions and $E\!f\!f_a$ to the effects of the action. Preconditions (*Pre*) and the goal G are first-order formulas over propositional state variables. Action effects (*Eff*) are sets of assignments to propositional state variables.

An action a is *applicable* in a state s only if its precondition is satisfied in s ($s \models Pre_a$). The outcome after the application of an action a will be the state where variables that are assigned in $E\!f\!f_a$ take their new value, and variables not referenced in $E\!f\!f_a$ keep their current values. A sequence of actions $\langle a_0, \ldots, a_{n-1} \rangle$ is called a *plan*. We say that the application of a plan starting from the initial state I brings the system to a state s_n. If each action is applicable in the state resulting from the application of the previous action and the finalstate satisfies the goal (i.e., $s_n \models G$), the sequence of actions is a *valid plan*. A planning problem has a solution if a valid plan can be found for the problem.

The Planning Domain Definition Language (PDDL) [17] is the de-facto standard modelling language for planning problems, supported by most planning systems. Its widespread use started thanks to the collaborative efforts and desire of the community to facilitate benchmarking and applications of planning systems. When using PDDL, the user describes the problem in terms of predicates, actions and functions with parameters. In turn, these parameters are instantiated with a set of defined objects.

2.1 Planning as Satisfiability

When a planning problem has a fixed length, such as peg solitaire [19], 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. There have been various successful approaches to encoding a planning problem into SAT [21, 29] and to CP [6, 30, 3, 24], amongst others. When encoding these problems, it is common in this situation to solve the planning problem by considering a sequence 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.

As described in Section 5, in constructing each ϕ herein we take the common approach [6, 19] 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 problem variables and its effects are applied to modify the 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, i.e. constraints specify that if no action has modified a variable, it keeps its value between steps. There are semantics such as the $\forall$ and $\exists$-step [29], or transition-systems [15] 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, i.e., one action per step.

3 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 [10], some variants of patience like Black Hole [12], and board games such as peg solitaire [19] or the knight’s tour [2]. 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 bouncing down vertically onto the grid. 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, 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 differently colored block (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. However, in our models we consider the task of finding a solution without reaching any dead end. Each level also begins with the avatar holding a wildcard block.

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.

4 Modelling Plotting in PDDL

As Plotting is naturally characterised as a planning problem, we start by modelling it in PDDL [17], the de-facto standard language for AI planners. Due to its length, the full PDDL model can be found in the supplementary material. PDDL is an expressive and modular modelling language, able to encode many real-life problems with complex dynamics. However, the complexity of its many features resulted in most AI planners lagging behind, supporting only a small core set of features.

To compactly model the sets of state variables F and actions A as described in Section 2, PDDL models use parameterised representations with types. PDDL is *action-oriented*: a PDDL model mainly defines the possible actions at each step. Also for each action, we must define the `precondition` over the state of the previous time step required to perform the action, and the `effect` over the state when that action is performed.

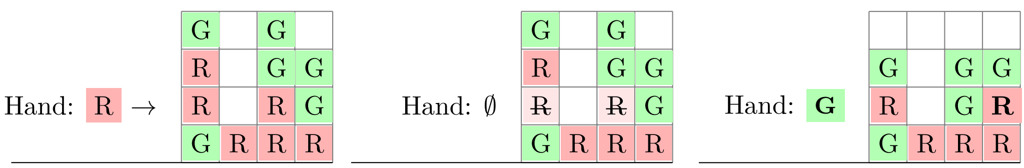


Figure 2 Diagram of a horizontal shot. R and G denote red and green blocks respectively. The initial state is shown on the left figure. The middle figure shows the blocks directly affected: the two light-red crossed out blocks will be removed, and all of the blocks on top will fall downwards. Finally, 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 not equal. A vertical shot works similarly.

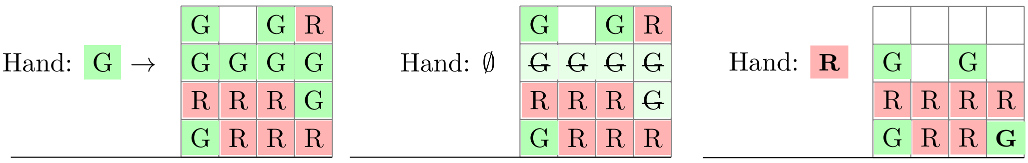


Figure 3 A more complex shot where the firing block reaches the end and goes downwards. Note the top right red block has to fall a variable number of positions (two in this case), depending on the state of the board and the colour of the shot.

4.1 On Numeric Planning

Naturally, one would gravitate towards the PDDL versions for numeric planning to be able to use numeric indexing. In [11], where PDDL is extended with numeric features, it is said:

> 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 $Object_n \to \mathbb{R}$, 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 numeric planning useless for our needs.

In addition, 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. 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 color. As we will see, 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 number in PDDL, here we are forced to use quantifiers to be able to refer to the "block that is above it" (i.e., its row is equals to `row+1`). Therefore, we must define predicates to simulate some basic arithmetic on indices.

4.2 The PDDL Model

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

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 perspicuity 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 specific scenario information. 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 $(p1, p2)$ between 1 and 5 such that $p1 < p2$.

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 6,12,19,28), which could have been alleviated 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 25-26). 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

```
1  (:action shoot-partial-row
2      ;; ?r - what row we are shooting at, ?t - the end cell, ?c - the colour we are removing
3      :parameters (?r - number ?t - number ?c - colour)
4      :precondition (and
5          ;; ?col is the successor of ?t with a different colour than ?c
6          (exists (?col - number)
7              (and  (succ ?col ?t)
8                    (not (coloured ?r ?col ?c))
9                    (not (coloured ?r ?col null))))
10         ...
11         ;; all the blocks up to ?t have either the colour ?c or are null
12         (forall (?col - number)
13             (or  (lt ?t ?col)
14                  (and (= ?col ?t) (coloured ?r ?t ?c))
15                  (or (coloured ?r ?col ?c)
16                      (coloured ?r ?col null)))))
17     :effect (and
18         ;; Change hands colour and the next cell that we cannot remove gets the colour from hand
19         (forall (?nextcolumn - number ?nextcolour - colour)
20             (when
21                 (and (succ ?nextcolumn ?t)
22                      (coloured ?r ?nextcolumn ?nextcolour))
23                 (and (not (coloured ?r ?nextcolumn ?nextcolour))
24                      (coloured ?r ?nextcolumn ?c)
25                      (hand ?nextcolour)
26                      (not (hand ?c)))))
27         ;; Move everything downwards. Consider 2 cases: base case (top row), and general case (rest).
28         (forall (?currentrow ?nextrow ?currentcol - number)
29             (and ;; First, the general case. Any row except the top one
30                 (forall (?currentcolor ?nextcolor - colour)
31                     (when
32                         (and (lt ?currentrow ?r)
33                              (succ ?nextrow ?currentrow)
34                              (or (lt ?currentcol ?t) (= ?currentcol ?t))
35                              ;; We ensure that the cells have the pertaining colours
36                              (coloured ?currentrow ?currentcol ?currentcolor)
37                              (coloured ?nextrow ?currentcol ?nextcolor)
38                              (not (= ?currentcolor ?nextcolor))) ; avoids a contradiction
39                         (and (not (coloured ?nextrow ?currentcol ?nextcolor))
40                              (coloured ?nextrow ?currentcol ?currentcolor)))))))
41                 ; Then, the base case of firing on the top row.
42                 ...))
```

Figure 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.

to be discovered. This quantification is introduced in line 19, and the values of `?nextcolumn` and `?nextcolour` are discovered in lines 20-22 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 19-26 could theoretically be simplified to:

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

Unfortunately, as per the previous subsection, functions can not 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))
                     (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.

5 Constraint Models in Essence Prime

Rendl et al. [28] 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 a state and action style, as noted in Section 2.1. Here, 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.

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 indexed by[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 2d array of decision variables and an individual variable per time step respectively. 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 indexed by[STEPSFROM1] of int(0..gridHeight)
find fpCol : matrix indexed by[STEPSFROM1] of int(0..gridWidth)
find grid  : matrix indexed by[STEPSFROM0, GRIDROWS, GRIDCOLS] of EMPTYANDCOLOURS
find hand  : matrix indexed by[STEPSFROM0] of COLOURS
```

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 2d 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:

```
    forAll step : STEPSFROM1 . $ Exactly one fp axis must be 0. (XOR, only ONE fired angle)
      (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). Due to the complexity of the state changes, this model is quite substantial in size and is provided in full in the supplementary material. 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 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]) /\
     (forAll left : int(1..col-1) . $Left, empty/hand colour, must exist a block of hand colour.
        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 apart from 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 indexed by[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]) \/
     (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. Test by comparing wallFall 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:

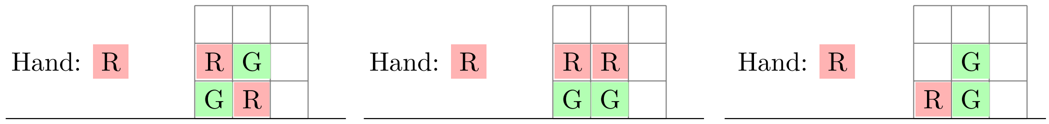
```
forAll step : STEPSFROM1 .
  forAll gRow : GRIDROWS .
    forAll gCol : GRIDCOLS .
      (grid[step,gRow,gCol] = EMPTY)
      =
      ( $ When a cell is EMPTY, it stays EMPTY
        (grid[step-1,gRow,gCol] = EMPTY) \/
        ...
        $ Final Column shot along a row consuming several blocks underneath
        ( $ Only the final column
          (gCol = gridWidth) /\
          $ There was a wallfall - this implies 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. 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))
```



(a) A game state with non-interchangeable column shots **(b)** A game state with non-interchangeable column shots **(c)** A state that can only lead to dead ends.

Figure 5 Illustrative Plotting game situations.

```
          =
          ( ...
            $ 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])
            ) \/ ...
          )
```

5.5 Symmetry Breaking

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:

```
  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),
```

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 5a 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 5b, 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:

```
  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),
```

5.6 An Implied Constraint

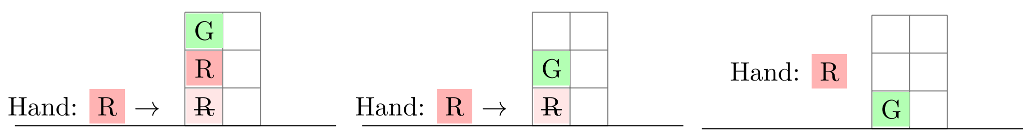
Consider an arbitrary grid with one red block. If that red block is transferred to the avatar's hand then there is no possible move. Hence, this state is only permissible following the final shot in the sequence. If red is already in the hand then the next move must shoot at the red block in the grid, again resulting in another colour in the hand and one red block in the grid, except in a situation like Figure 5c, 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):

```
    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 6a. 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 6b. Similarly, if we again shoot on the bottom row, the result is Figure 6c. Hence, a counterexample: red appears twice in the hand when there are only two blocks in the grid. Note that the constraints in Section 5.5 and this implied constraint are applicable to models in Sections 5.3 and 5.4 as they both share the same viewpoint.

6 Empirical Evaluation

We have created a dataset of 200 instances using our parameterised instance generator. These have similar properties to the original game levels in terms of size, number of colours and goals: their sizes range from 2×4 to 7×7, the number of colours range from 2 to 4 and the maximum allowed remaining blocks (goal) range from 5 to 2. In the original game, the scenario sizes range from 4×4 to 6×6 with 4 colors. The goal objectives also depend on the difficulty level but usually range from 7 to 3. The only difference in our synthetic instances is that we always allow firing on all rows and columns. Five of our synthetic instances are unsolvable, i.e., you always reach a state where you cannot make a progressing move.



(a) State 1 **(b)** State 2 **(c)** State 3

Figure 6 With two red blocks remaining, red can appear in the hand twice.

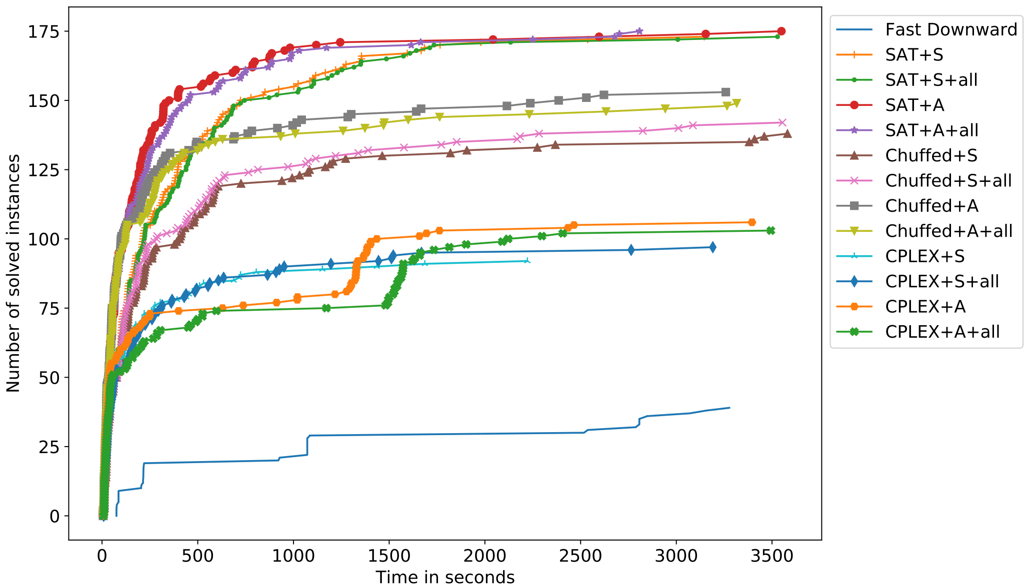


Figure 7 Cumulative instances solved for each model and solver. The *all* variant of the state- (S) and action-focused (A) constraint models includes implied and symmetry-breaking constraints.

Our experiments were executed on a cluster of compute nodes with two 2.1 GHz 18-core Intel Xeon (Broadwell) processors each. Each process was given a limit of 8GB of memory and 1-hour timeout. We used Savile Row [26] 1.9.1 with three different backend solvers: CaDiCaL [7], Chuffed [8] and CPLEX Optimisation Studio 20.10. We also used the Fast Downward [18] 20.06+ planner. We did consider all planners present in the last IPC and only 9 claimed to support the features required. Of those, 7 were based on the Fast Downward preprocessor and the others crashed when given the instances. We opted to include only results on Fast Downward because pre-processing for all planners based on Fast Downward is the same, and for the successfully pre-processed instances the search time is very small.

Fast Downward is the best-known, supported and reused state-of-the-art planning system, winning the last International Planning Competition (IPC) using some of its portfolio configurations. Its preprocessing module performs sophisticated transformations from PDDL to the more solver-amenable SAS+ format [4], and is reused by many state-of-the-art planners. Still, planning benchmarks do not usually require the expressivity in the language that Plotting does. The extensive use of quantifiers and complex conditional effects in the PDDL model are a heavy burden on the preprocessor, preventing the planner from pre-processing grids greater than 3×3 within the given time-out and memory constraints.

The longest satisfiable instance solved within the time and memory limits has 26 steps. As per Section 2.1, when not using Fast Downward, for each instance we consider a sequence of decision problems from 1 to $(width \times height) - goal$ steps. We generally observe a phase transition around the first satisfiable step. In most cases pre-processing by Savile Row is significant. For the solved instances, an average of 54% of the total time is spent on preprocessing for CPLEX, 51% for SAT and 53% for Chuffed. For some intermediate steps, Savile Row can prove an instance unsatisfiable before encoding it for the backend solver.

We refer to the action-focused (Section 5.3) state-focused (Section 5.4) as models A and S. Figure 7 shows a cactus plot, considering both with and without additional constraints. The plot clearly splits the solvers in four performance profiles. SAT solves most instances,

	#instances					PAR2 Score				
	none	de	em	mo	all	none	de	em	mo	all
SAT+S	174	0	0	0	0	248764	-428	+1129	+1093	+3714
Chuffed+S	139	+5	+5	+1	+4	493458	-34174	-42729	-15553	-28534
CPLEX+S	93	+7	+6	+5	+5	788953	-36517	-32037	-25446	-26264
SAT+A	176	0	-1	-1	0	213866	+1674	+7078	+6875	+3994
Chuffed+A	154	-15	-12	-10	-4	371833	+96809	+76535	+63611	+28808
CPLEX+A	107	+1	+4	-1	-3	719877	-8118	-15288	+10127	+29473

Table 1 Number of instances solved and PAR2 score per solver and model. Column *none* is performance without the extra constraints. Columns *de*, *em* and *mo* show the differences in performance with the dead end implied constraint, the empty column and monochrome symmetry breaking constraints 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. For example, CPLEX+A solves more instances when separately adding the *de* and *em* constraints to the base model, but solves less instances when adding *mo* or *all* of them in combination. The PAR2 score summarises how this affects solving times in all instances.

followed by Chuffed, CPLEX and finally Fast Downward. Comparing models S and A, we see three different behaviours. With SAT, the number of solved instances converges regardless of the model, with model A slightly faster. For Chuffed, there is a clear performance gap between them throughout. CPLEX seems to work better with model S until around the 1500 second mark, where model A overtakes it. Overall, model A performs consistently better.

Table 1 summarises performance with and without the extra constraints. The PAR2 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. Considering the PAR2 scores, the extra constraints are generally slightly harmful for SAT, with only one exception: the dead end implied constraint when using SAT+S. Chuffed and CPLEX show a notable difference between models: Adding additional constraints to the S model consistently help, while if we do the same for model A it generally hinders solving efficiency.

Breaking symmetries in the PDDL model would require even more involved preconditions. For instance, we must state that when shooting a monochrome column there is no (same-coloured) monochrome column in a precedent position. Unfortunately, preprocessing time in the planner is critical in comparison to solving time. Therefore we have not implemented symmetry breaking in PDDL. The native way of handling these is using the `constraints` PDDL3.0 extension [13], sadly with no support among state-of-the-art planners.

7 Conclusions and Further Work

Although Plotting is a planning problem, we have shown that automated planners cannot deal efficiently with a natural PDDL model. The lack of support for some crucial PDDL features such as multi-valued variables, functional symbols and numeric reasoning makes the modelling of problems with complex transitions a cumbersome and error-prone process.

We have presented alternative models in Essence Prime and, in an extensive empirical analysis supported by a new instance generator, experimentally validated that this approach is efficient using a variety of solving technologies. Although both planning and constraint models are quite involved, since Essence Prime is a more expressive language most key points in the model are easier to encode. Native constructs for Essence Prime to express planning-specific primitives would further aid the encoding of planning problems.

References

1. Accenture. The Global Gaming Industry Value Now Exceeds $300 Billion, New Accenture Report Finds. https://newsroom.accenture.com/news/global-gaming-industry-value-now-exceeds-300-billion-new-accenture-report-finds.htm, 2021. [Online; accessed 2-Feb-2022].
2. Özgür Akgün, Ian P. Gent, Christopher Jefferson, Ian Miguel, Peter Nightingale, and András Z. Salamon. Automatic discovery and exploitation of promising subproblems for tabulation. In *Principles and Practice of Constraint Programming - 24th International Conference, CP*, volume 11008, pages 3–12, 2018. doi:10.1007/978-3-319-98334-9_1.
3. Behrouz Babaki, Gilles Pesant, and Claude-Guy Quimper. Solving classical AI planning problems using planning-independent CP modeling and search. In Daniel Harabor and Mauro Vallati, editors, *Proceedings of the Thirteenth International Symposium on Combinatorial Search, SOCS*, pages 2–10. AAAI Press, 2020.
4. Christer Bäckström and Bernhard Nebel. Complexity Results for SAS+ Planning. *Comput. Intell.*, 11:625–656, 1995. doi:10.1111/j.1467-8640.1995.tb00052.x.
5. Roman Barták, Miguel A Salido, and Francesca Rossi. Constraint satisfaction techniques in planning and scheduling. *Journal of Intelligent Manufacturing*, 21(1):5–15, 2010.
6. Roman Barták and Daniel Toropila. Reformulating constraint models for classical planning. In David Wilson and H. Chad Lane, editors, *Proceedings of the Twenty-First International Florida Artificial Intelligence Research Society Conference, May 15-17, 2008*, pages 525–530. AAAI Press, 2008.
7. Armin Biere, Katalin Fazekas, Mathias Fleury, and Maximillian Heisinger. CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020. In Tomas Balyo, Nils Froleyks, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda, editors, *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions*, volume B-2020-1 of *Department of Computer Science Report Series B*, pages 51–53. University of Helsinki, 2020.
8. Geoffrey Chu, Peter J. Stuckey, Andreas Schutt, Thorsten Ehlers, Graeme Gange, and Kathryn Francis. Chuffed 0.10.4. https://github.com/chuffed/chuffed, 2019 (accessed 03-05-2022).
9. Joan Espasa, Ian Miguel, Jordi Coll, and Mateu Villaret. Towards lifted encodings for numeric planning in essence prime. *Proceedings of the 18th International Workshop on Constraint Modelling and Reformulation (ModRef)*, 2019.
10. Joan Espasa Arxer, Ian P Gent, Ruth Hoffmann, Christopher Jefferson, Matthew J McIlree, and Alice M Lynch. Towards generic explanations for pen and paper puzzles with MUSes. In *Proceedings of the SICSA eXplainable Artifical Intelligence Workshop*, 2021.
11. Maria Fox and Derek Long. PDDL2.1: an extension to PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence Research*, 20:61–124, 2003. doi:10.1613/jair.1129.
12. Ian P Gent, Chris Jefferson, Tom Kelsey, Inês Lynce, Ian Miguel, Peter Nightingale, Barbara M Smith, and S Armagan Tarim. Search in the patience game 'black hole'. *AI Communications*, 20(3):211–226, 2007.
13. Alfonso Gerevini and Derek Long. Plan constraints and Preferences in PDDL3. Technical report, Technical Report 2005-08-07, Department of Electronics for Automation, University of Brescia, Brescia, Italy, 2005, 2005.
14. Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning: theory and practice*. Elsevier, 2004.
15. Nina Ghanbari Ghooshchi, Majid Namazi, M. A. Hakim Newton, and Abdul Sattar. Encoding domain transitions for constraint-based planning. *Journal of Artificial Intelligence Research*, 58:905–966, 2017. doi:10.1613/jair.5378.
16. Gaël Glorian, Adrien Debesson, Sylvain Yvon-Paliot, and Laurent Simon. The dungeon variations problem using constraint programming. In Laurent D. Michel, editor, *27th International Conference on Principles and Practice of Constraint Programming, CP 2021, Montpellier, France*, volume 210 of *LIPIcs*, pages 27:1–27:16. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.CP.2021.27.

17 Patrik Haslum, Nir Lipovetzky, Daniele Magazzeni, and Christian Muise. *An Introduction to the Planning Domain Definition Language.* Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2019. `doi:10.2200/S00900ED2V01Y201902AIM042`.

18 Malte Helmert. The Fast Downward Planning System. *J. Artif. Intell. Res.*, 26:191–246, 2006. `doi:10.1613/jair.1705`.

19 Christopher Jefferson, Angela Miguel, Ian Miguel, and Armagan Tarim. Modelling and solving english peg solitaire. *Comput. Oper. Res.*, 33(10):2935–2959, 2006. `doi:10.1016/j.cor.2005.01.018`.

20 Christopher Jefferson, Wendy Moncur, and Karen E Petrie. Combination: Automated generation of puzzles with constraints. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, pages 907–912, 2011.

21 Henry A. Kautz and Bart Selman. Planning as Satisfiability. In *ECAI*, pages 359–363, 1992.

22 Derek Long. Drilling down: Planning in the field. Invited Talk, Twenty-Ninth International Conference on Automated Planning and Scheduling, (ICAPS), Berkeley, CA, USA, 2019.

23 Arman Masoumi, Megan Antoniazzi, and Mikhail Soutchanski. Modeling Organic Chemistry and Planning Organic Synthesis. In *Global Conference on Artificial Intelligence (GCAI)*, pages 176–195, 2015.

24 Ian Miguel, Peter Jarvis, and Qiang Shen. Flexible graphplan. In *ECAI*, pages 506–510, 2000.

25 Tim Niemueller, Erez Karpas, Tiago Vaquero, and Eric Timmons. Planning competition for logistics robots in simulation. In *Workshop on Planning and Robotics (PlanRob) at International Conference on Automated Planning and Scheduling (ICAPS)*, 2016.

26 Peter Nightingale, Özgür Akgün, Ian P. Gent, Christopher Jefferson, Ian Miguel, and Patrick Spracklen. Automatically improving constraint models in Savile Row. *Artificial Intelligence*, 251:35–61, 2017.

27 Peter Nightingale and Andrea Rendl. Essence' description. *CoRR*, abs/1601.02865, 2016. `arXiv:1601.02865`.

28 Andrea Rendl, Ian Miguel, Ian P. Gent, and Peter Gregory. Common subexpressions in constraint models of planning problems. In *Eighth Symposium on Abstraction, Reformulation, and Approximation, SARA*. AAAI, 2009.

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

30 Peter van Beek and Xinguang Chen. CPlan: A Constraint Programming Approach to Planning. In *Sixteenth National Conference on AI and Eleventh Conference on Innovative Applications of AI*, pages 585–590, 1999.

31 Vincent Vidal and Héctor Geffner. Branching and pruning: An optimal temporal POCL planner based on constraint programming. *Artificial Intelligence*, 170(3):298–335, 2006.