Challenges in Modelling and Solving Plotting with PDDL *

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

[1]School of Computer Science, University of St Andrews, UK
[2]Department of Computer Science, University of York, UK
[3]Departament d'Informàtica, Matemàtica Aplicada i Estadística, Universitat de Girona, Spain
{jea20,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 game is to remove a target number of coloured blocks from a grid by sequentially shooting blocks into the grid. Plotting features complex transitions after every shot: various blocks are affected directly, while others can be indirectly affected by gravity. We highlight the challenges of modelling Plotting with PDDL and of solving it with a grounding-based state-of-the-art planner.

Introduction

We consider 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. The game 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 (Espasa, Miguel, and Villaret 2022a), aiming to find a sequence of firing positions such that enough blocks are removed to beat the scenario objective. The complexity of state transitions after every shot makes this problem interesting: some blocks are affected directly, while others can be indirectly affected by gravity, as explained in the next section. Modelling the game dynamics in PDDL (Haslum et al. 2019) is difficult, as we will demonstrate.

The resulting complexity of the model severely hinders the ability of current planning systems to produce a valid plan. Most state-of-the-art AI planners rely on grounding, instantiating every action schema for all meaningful combinations of parameters. As we will show, the complexity of Plotting is too much for this grounding process. Problems with grounding are now attracting attention in the planning community (Matloob and Soutchanski 2016; Corrêa et al. 2020), with suggestions to avoid grounding *lifted* representations as far as possible. A lifted representation succinctly
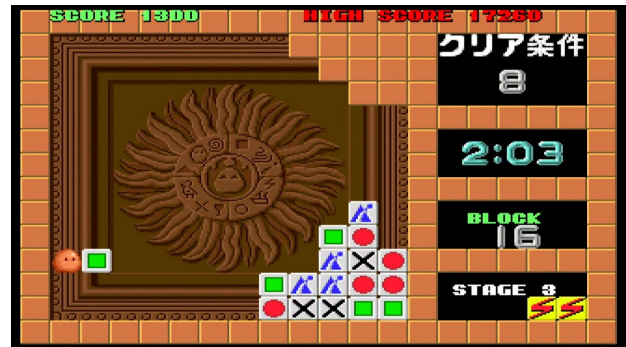
Figure 1: Plotting (Taito, 1989). The avatar is seen on the left, holding a green block. The objective is to reduce the number of blocks in the middle pile. In this particular case there are 16 left (see center-right of the image), and the goal is 8 or less (see top-right of image).

defines actions by grouping them with their preconditions and effects using action schemas with parameters.

Constraint modelling languages can be used to express planning problems (Barták, Salido, and Rossi 2010; Babaki, Pesant, and Quimper 2020; Barták and Toropila 2008; Espasa et al. 2019; van Beek and Chen 1999). These languages are more expressive than PDDL and permit a succinct lifted representation of Plotting, providing access to lifted solving approaches that don't need exhaustive grounding. However, they are not a panacea and require significant human effort to be put into modelling. In this work we describe Plotting and provide a working model, an instance generator and a set of benchmark instances. We also highlight the challenges of modelling and solving Plotting with PDDL.

Plotting

Plotting is played by one agent with full information of the game 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 (Espasa et al. 2021), some variants of patience like Black Hole (Gent et al. 2007), and board games such as peg solitaire (Jefferson et al. 2006) or the knight's tour (Akgün et al. 2018). 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. Note that we consider the topmost row as the first row and the leftmost column as the first column. 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. If the block 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: this is 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 complex shot is depicted in Figure 2, 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 green block replaces the final red block, which rebounds to the avatar's hand. 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.

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. In the model we abstract the avatar's movement to consider the key decisions: the rows or columns chosen at which to shoot the held block. 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.

Limitations of Planning Approaches

Tools to solve problems such as Plotting should ideally support natural ways of expressing elements such as 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. PDDL 2.1 (Fox and Long 2003) added support for numeric and temporal features, extending the expressivity of the language. Still, such an extension is insufficient for efficiently modelling and solving Plotting. Fox and Long (2003) states:

> 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
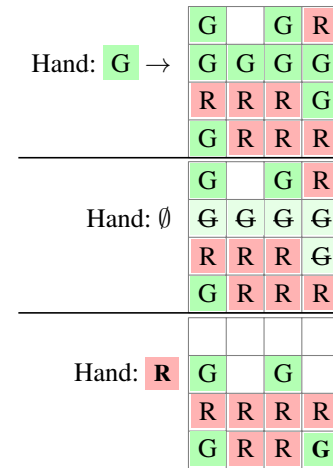

Figure 2: A diagram of a shot where the firing block reaches the end and goes downwards. The top right red block has to fall a variable number of positions (two in this case), depending on the state of the board and the colour of the shot.

> 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.

In other words, 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 use 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. 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 the next section, we are forced to define predicates to simulate some basic arithmetic operations on indices.

Functional STRIPS (Geffner 2000) or Planning Modulo Theories (Gregory et al. 2012) would alleviate the expressivity problems faced with Plotting. On one hand, with Functional STRIPS extensions such as those of Francès and Geffner (2016) 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, the Planning Modulo Theories paradigm could support a theory with both matrices and arithmetic.

Unfortunately, these 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 (Francès and Geffner 2016) has dependencies on software which we have not been able to find and the Planning Modulo Theories planner of Gregory et al. (2012) was not released. Therefore, it would require a significant engineering effort to either reproduce or re-engineer them. Considering available and well supported planners, we are limited to using classical planners.

Modelling Plotting in PDDL

We now 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 each cell is the colour of the block it contains, or `null` if empty. Therefore, the state 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 both 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 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 (p_1, p_2) between 1 and 5 such that $p_1 < p_2$. Figure 3 is an excerpt of the action consisting of partially removing blocks of colour `?c` in row `?r` until column `?t`, not reaching the last column. One of the principal difficulties is in identifying successors and predecessors of particular rows or columns (e.g. Lines 8, 14, 22, 32), which could have been eased by support for arithmetic 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 28-29). 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 22, and the values of `?nextcolumn` and `?nextcolour` are discovered in lines 23-25 as a condition for the effect to take place.

If we could use function symbols and arithmetic, we could remove variables `?nextcolumn` and `?nextcolour`,

```
1  (:action shoot-partial-row
2    ;; ?r - what row we are shooting at
3    ;; ?t - the end cell where the shot ends
4    ;; ?c - the colour we are removing
5    :parameters (?r - number ?t - number ?c - colour)
6    :precondition (and
7      ;; ?col is successor of ?t coloured differently to ?c
8      (exists (?col - number)
9        (and (succ ?col ?t)
10            (not (coloured ?r ?col ?c))
11            (not (coloured ?r ?col null))))
12     ...
13     ;; all blocks up to ?t are either colour ?c or null
14     (forall (?col - number)
15       (or  (lt ?t ?col)
16            (and (= ?col ?t) (coloured ?r ?t ?c))
17            (or (coloured ?r ?col ?c)
18                (coloured ?r ?col null)))))
19   :effect (and
20     ;; Change hand colour
21     ;; next cell that we cannot remove gets hand colour
22     (forall (?nextcolumn - number ?nextcolour - colour)
23       (when
24         (and (succ ?nextcolumn ?t)
25              (coloured ?r ?nextcolumn ?nextcolour))
26         (and (not (coloured ?r ?nextcolumn ?nextcolour))
27              (coloured ?r ?nextcolumn ?c)
28              (hand ?nextcolour)
29              (not (hand ?c)))))
30   ;; Move everything downwards.
31   ;; 2 cases: base case (top row), general case (rest)
32   (forall (?currentrow ?nextrow ?currentcol - number)
33     (and ;; the general case: any row except the top
34       (forall (?currentcolor ?nextcolor - colour)
35         (when
36           (and
37             (lt ?currentrow ?r)
38             (succ ?nextrow ?currentrow)
39             (or (lt ?currentcol ?t)
40                 (= ?currentcol ?t))
41             ;; ensure cells have the pertaining colours
42             (coloured ?currentrow ?currentcol ?currentcolor)
43             (coloured ?nextrow ?currentcol ?nextcolor)
44             ;; avoid a contradiction:
45             (not (= ?currentcolor ?nextcolor)))
46           (and
47             (not (coloured ?nextrow ?currentcol ?nextcolor))
48             (coloured ?nextrow ?currentcol ?currentcolor)
49         ))))))); Then, case of firing on the top row.
50       ...))
```

Figure 3: Fragment of the *shoot-partial-row* action.

changing the `coloured` symbol to a function that, given a row and column, maps to the colour in that cell. Overall, lines 22-29 could theoretically be simplified to:

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

Unfortunately, functions can not have numeric expressions as parameters. 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`. E.g. requiring at most 2 non-`null` cells creates:

```
(: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 simpler to state in a constraint language with, for example, an `atleast` constraint.

Empirical Evaluation

We improved our previous instance generator (Espasa, Miguel, and Villaret 2022b) to avoid generating symmetric grids, ensuring more interesting instances. A new set of 522 instances was then created with a range of difficulties[1]. These use from 8 to 49 blocks, and from 2 to 6 colours.

We considered the best planners in the 2018 International Planning Competition. From all planners, 9 claimed to support the features required. Of those, 7 were based on the Fast Downward preprocessor and the rest crashed when given the instances. We therefore present results for only Fast Downward (Helmert 2006) 22.12 because the pre-processing for all planners based on Fast Downward is the same, and for the successfully pre-processed instances the search time was very small. The integrated Stone Soup portfolio showed that only the blind heuristic supported the features in the model.

We also used a Planning as Satisfiability (Kautz and Selman 1992) approach and translating the problem to the constraint programming language Essence Prime. Then, Savile Row (Nightingale et al. 2017) 1.9.1 was used to solve the problem with three different solvers: kissat 3.0.0, Chuffed 0.10.4, and OR-Tools 9.5. As a brief summary of the approach, a planning problem is encoded to a Boolean formula (or constraint satisfaction problem), with the property that any model of this formula will correspond to a valid plan. Since the length of a valid plan is not known a priori, we encode the existence of a plan of T steps with a formula $f(T)$. Then, the method for finding the shortest plan consists in iteratively checking the satisfiability of $f(T)$ for $T = 0, 1, 2, \ldots$ until a satisfiable formula is found. We observed 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. Combined with the fact that each shot should remove at least one block from the grid, for each instance we can consider a sequence of decision problems from 1 up to

---

[1]Model and instances are available in `https://github.com/stacs-cp/Plotting-Journal`

$(width \times height) - \max(goal_b, colours)$ 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. As expected, the hardest queries for each instance are the last unsatisfiable one (with the greatest queried time horizon among the unsatisfiable ones) and the first satisfiable query.

Experiments were executed on a cluster of compute nodes with two 2.1 GHz 18-core Intel Xeon processors each. Each process was limited to 8GB of memory and 1 hour. In the table we compare instances solved within resource limits and the PAR2 score for each solver.

Solver	Instances Solved	PAR2 Score (s)
Chuffed	**510**	**162537**
Kissat	508	176207
OR-Tools	488	329176
FD	78	3233535

The three CP solvers all perform reasonably well, being able to solve most instances within the given time. In contrast, Fast Downward exhibits a significantly worse performance scaling, and also only solves the smallest instances, getting stuck during the grounding process. When using the best constraint model, neither SAT, OR-Tools nor Chuffed ran out of memory. In contrast, out of the 522 instances, Fast Downward runs out of memory for 249 and times out for 195. Further, when Fast Downward exceeds resource bounds, it always does so during grounding.

Conclusions

As we have shown, classical planning can solve Plotting despite the highlighted PDDL limitations. Over the last few decades, the prevalent method of solving classical planning problems has been heuristic search. In such approaches, a grounded representation of the problem is generally needed to be able to then compute heuristic values that guide the search. The grounding component in most of the planners struggles when presented with the PDDL model. 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 appears to be a *hard-to-ground problem* (Corrêa et al. 2020), and we therefore intend to investigate models that are easier to ground.

Problems with grounding could be mitigated by using a more expressive language, like those proposed by Gregory et al. (2012) or Francès and Geffner (2016), allowing more concise and efficient problem representation. However, one may also need to deal with the grounding of this richer language to apply similar solving methods. Corrêa et al. (2020) 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. Improvements to grounding, such as partial grounding (Gnad et al. 2019) or pruning-during-grounding, may enable automated planning systems to leapfrog CP and SAT solvers for this hard-to-ground problem.

Acknowledgements

References

Akgün, Ö.; Gent, I. P.; Jefferson, C.; Miguel, I.; Nightingale, P.; and Salamon, A. Z. 2018. Automatic Discovery and Exploitation of Promising Subproblems for Tabulation. In *CP*, volume 11008 of *LNCS*, 3–12. Springer. https://doi.org/10.1007/978-3-319-98334-9_1.

Babaki, B.; Pesant, G.; and Quimper, C. 2020. Solving Classical AI Planning Problems Using Planning-Independent CP Modeling and Search. In *SOCS*, 2–10. https://doi.org/10.1609/socs.v11i1.18529.

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

Barták, R.; and Toropila, D. 2008. Reformulating Constraint Models for Classical Planning. In *FLAIRS*, 525–530. https://web.archive.org/web/20220202110048/https://kti.mff.cuni.cz/~bartak/downloads/FLAIRS2008.pdf.

Corrêa, A. B.; Pommerening, F.; Helmert, M.; and Francès, G. 2020. Lifted Successor Generation Using Query Optimization Techniques. In *ICAPS*, 80–89. https://doi.org/10.1609/icaps.v30i1.6648.

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

Espasa, J.; Miguel, I.; Coll, J.; and Villaret, M. 2019. Towards Lifted Encodings for Numeric Planning in Essence Prime. *Proceedings of the 18th International Workshop on Constraint Modelling and Reformulation (ModRef)*. https://modref.github.io/papers/ModRef2019_Towards%20Lifted%20Encodings%20for%20Numeric%20Planning%20in%20Essence%20Prime.pdf.

Espasa, J.; Miguel, I.; and Villaret, M. 2022a. CSPLib Problem 088: Plotting. http://www.csplib.org/Problems/prob088.

Espasa, J.; Miguel, I. J.; and Villaret, M. 2022b. Plotting: a planning problem with complex transitions. In *CP*, volume 235 of *LIPIcs*. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. https://doi.org/10.4230/LIPIcs.CP.2022.22.

Fox, M.; and Long, D. 2003. PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains. *Journal of Artificial Intelligence Research*, 20: 61–124. https://doi.org/10.1613/jair.1129.

Francès, G.; and Geffner, H. 2016. Effective Planning with More Expressive Languages. In *IJCAI*, 4155–4159. https://www.ijcai.org/Abstract/16/621.

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

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

Gnad, D.; Torralba, Á.; Domínguez, M. A.; Areces, C.; and Bustos, F. 2019. Learning How to Ground a Plan - Partial Grounding in Classical Planning. In *AAAI*, 7602–7609. https://doi.org/10.1609/aaai.v33i01.33017602.

Gregory, P.; Long, D.; Fox, M.; and Beck, J. C. 2012. Planning Modulo Theories: Extending the Planning Paradigm. In *ICAPS*, 65–73. https://doi.org/10.1609/icaps.v22i1.13505.

Haslum, P.; Lipovetzky, N.; Magazzeni, D.; and Muise, C. 2019. *An Introduction to the Planning Domain Definition Language*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Springer. https://doi.org/10.2200/S00900ED2V01Y201902AIM042.

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

Jefferson, C.; Miguel, A.; Miguel, I.; and Tarim, A. 2006. Modelling and solving English Peg Solitaire. *Comput. Oper. Res.*, 33(10): 2935–2959. https://doi.org/10.1016/j.cor.2005.01.018.

Kautz, H. A.; and Selman, B. 1992. Planning as Satisfiability. In *ECAI*, 359–363. https://web.archive.org/web/20230209175344/https://henrykautz.com/papers/satplan.pdf.

Matloob, R.; and Soutchanski, M. 2016. Exploring Organic Synthesis With State-of-the-Art Planners. In *Proceedings of Scheduling and Planning Applications woRKshop (SPARK)*, 52–61. https://icaps16.icaps-conference.org/proceedings/spark16.pdf.

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

van Beek, P.; and Chen, X. 1999. CPlan: A Constraint Programming Approach to Planning. In *AAAI*, 585–590. https://web.archive.org/web/20221125213531/https://www.aaai.org/Library/AAAI/1999/aaai99-083.php.