Cross-Paradigm Modelling: A Case Study of Puzznic

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

Ian P. Gent
School of Computer Science, University of St Andrews, UK

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

Peter Nightingale
Department of Computer Science, University of York, UK

András Z. Salamon
School of Computer Science, University of St Andrews, UK

Mateu Villaret
Department of Computer Science, Applied Mathematics and Statistics, University of Girona, Spain

Abstract

Puzznic is a tile-matching video game published by Taito in 1989 and ported to many platforms. The player manipulates blocks in a given grid until they match when two or more blocks of the same pattern are adjacent and are removed from play. The goal is to match all patterned blocks in the grid. Puzznic is rich in structure: levels have internal platforms and the blocks are affected by gravity, leading to complex state changes and the possibility of a cascaded series of matches following each move by the player. The puzzle is therefore a significant challenge to model, motivating our study. We study Puzznic from both constraint modelling and AI Planning perspectives, identifying their complementary strengths and weaknesses for this problem. We further exploit our constraint model to automate instance generation, parameterised on the grid, the combination of patterned blocks, and the steps required for a solution.

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

Keywords and phrases AI Planning, Modelling, Constraint Programming

Supplementary Material Anonymous repository of models and instances:
Model: `https://github.com/blind-anon/CP-2024-Submission-Additional-Material`

Funding *Ian Miguel*: EPSRC grant EP/V027182/1
Peter Nightingale: EPSRC grant EP/W001977/1
Mateu Villaret: Grant PID2021-122274OB-I00 funded by MCIN/AEI/10.13039/501100011033 and by ERDF A way of making Europe

1 Introduction

Puzznic (Taito, 1989) is a puzzle-based video game, ported to many platforms. The player manipulates blocks in a given grid until they *match* when two or more blocks of the same pattern are adjacent horizontally or vertically, and are removed from play. The goal is to remove all patterned blocks in the grid. An illustrative level from the game is presented in Figure 1. Unlike many other puzzle games [6, 19], instances of Puzznic, crafted to challenge human players, are not trivial to solve for automated approaches. Puzznic is rich in structure: levels may have internal platforms and the blocks are affected by gravity, leading to complex state changes and the possibility of a cascaded series of matches following each move by the

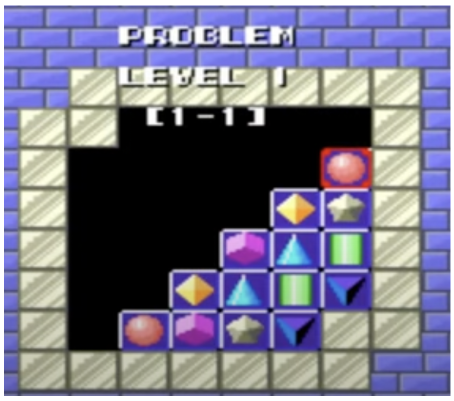


Figure 1 Detail from Puzznic (Taito, 1989). The red cursor that can be seen on the red circle block at the upper right of the image is controlled by the player and used to select and move individual blocks.

player. The puzzle is therefore a significant challenge to model, which motivates our study. Puzznic is naturally characterised as an AI Planning problem [11]. Given a model of the environment (here the grid, blocks, and their behaviour), a planning problem requires finding a sequence of actions (block moves) to progress from an initial state of the environment to a goal state (all blocks matched) while respecting a set of constraints.

Constraint Programming has been used to solve planning problems [1, 2] and has recently proven successful in solving Plotting [8], a puzzle game which, in common with Puzznic, has complex changes of state.

We build upon [7] to present models of Puzznic in the constraints paradigm and in PDDL, the standard modelling language of the AI Planning community. Our primary contributions are as follows: (i) A challenging new benchmark, which we establish for the first time to be in NP, (ii) Models for both AI Planning and constraint modelling paradigms, (iii) An instance generator based on our constraint model, (iv) An empirical comparison between our two models, establishing their complementary strengths and weaknesses across several sub-families of Puzznic instances.

2 Puzznic

Puzznic is a puzzle solitaire game, in which the player has full information about the game state and the effects of each action performed are deterministic. This is representative of tile-matching games such as Plotting [8], as well as board games like peg solitaire [14] and some variants of patience like Black Hole [10]. Each instance of the game consists of a grid of cells similar to that presented in Figure 1. Each cell may be empty, filled with a wall or contain a patterned block. The player controls a red cursor, visible at the top-right of the figure, with which they can select a single patterned block. A selected block can be *moved* horizontally left or right if the cell in the direction chosen is empty. Patterned blocks are affected by gravity, and *fall* until coming to rest above another non-empty block or a wall. If the player moves a block over an empty cell, they *immediately* lose control of the block as it falls. When two or more blocks with the same pattern are adjacent horizontally or vertically, they *match* and are removed from play. The only exception to this is when a patterned block is falling: it cannot match another block until it comes to rest. Via gravity, one match may result in further matches, etc. The goal is to remove all patterned blocks from the grid.

We highlight some challenging aspects of Puzznic.

Measure for Progress Solving other tile-matching puzzles such as Plotting [8] has been shown to benefit from having a notion of monotonic progress. In Puzznic some moves might be required to strategically position blocks for a later move, so it is not immediately clear what constitutes progress towards the goal.

Gravity Due to the effect of gravity, the state update after a player move is sometimes highly localised, but it can also precipitate a ripple effect across large regions of the board, triggering a cascade of falls and matches. Such large-scale changes to the board state creates difficulties for many solution approaches.

2.1 Membership in NP

Some levels of the game also have moving wall blocks, which can carry patterned blocks. In the current work we do not consider this aspect of Puzznic. The version of the game studied here, without moving wall blocks, is also known as *Cubic* and the solvability of this static variant has been claimed to be NP-hard by a reduction from deciding satisfiability of Boolean circuits [9]. The complexity of the solvability of the related but more complicated Hanano puzzle has also been studied [16], and this problem was shown to be PSPACE-complete [5].

It is notable that Static Puzznic Solvability is in the complexity class NP, because Puzznic is naturally represented as a planning problem, and solvability of planning problems is generally PSPACE-hard [4]. We could therefore expect solving Puzznic to be amenable to constraint programming approaches usually tailored to efficiently solve problems in NP.

Theorem. The decision problem for the Static Variant of Puzznic is in NP.

Proof. We sketch a proof here, giving the detailed proof in Appendix A. Verifying that a proposed sequence of moves leads to an empty board takes time that is polynomial in the length of the sequence and the size of the instance. Membership in NP would follow if we could show that any solvable instance has a short enough *nice* solution, of length that is polynomial in the size of the instance. This isn't immediately obvious, because sequences of moves back and forth are sometimes necessary to set up bridges across which blocks must travel to a matching block. Some solvable instances might therefore have no short solutions. However, this is not actually so: if a sequence of moves constitutes a solution (even if it is too long), then there is an equivalent nice sequence of moves, leading to the same matches and block falls, and which has length that is polynomial in the instance size. With each board position we associate an integer, the *fall metric*, by adding up the rows of all blocks in the position, row number 1 being the bottom row. (The fall metric for the position in Figure 1 is $0 * 1 + 4 * 2 + 4 * 3 + 3 * 4 + 2 * 5 + 1 * 6 = 48$.) The key step is to observe that in Static Puzznic Solvability, no block can be required to move in opposite directions without a change in the fall-metric intervening. We prove this by showing that in any hypothetical counterexample, we could always construct a new solution with one less occurrence of such a *switchback*, meaning we can always reduce the number of switchbacks to zero. To achieve this, we consider the last move in one direction by any block which also moves in the opposite direction before the metric changes. The new solution is simply the previous solution with this pair of switchback moves deleted. This doesn't change the position that will result at the end of the subsequence of moves, as long as it doesn't invalidate any move between the two edited-out moves or cause any falls or matches. A case analysis of all ways that such invalidity might occur shows that the new sequence is in fact always valid. Finally, in the Static variant of the puzzle the fall metric never increases and can only decrease a limited number of times, bounded by at most the number of blocks times the height of the grid. Between fall-metric changes the number of blocks moved in a nice solution is at most the number of blocks times the width of the grid. The total number of moves is therefore polynomially bounded by the grid size. These facts yield a polynomial bound for the overall length of a nice solution and membership in NP follows.

3 PDDL Model

Considering Puzznic as a classical planning problem requires finding a sequence of actions (a *plan*) where their application will successively transform a given initial state until a goal state is reached. A set of finite-domain variables determines the state at each step. An action is applicable at a certain state if the state satisfies the preconditions of the action. The state is then modified according to the effects of the action. The Planning Domain Definition Language (PDDL) [12] is the *de facto* planning definition standard. PDDL separates a planning problem into two files: the *domain*, defining 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. In this section we describe our PDDL Puzznic formulation.

Although a possible way of representing the game could be to consider a matrix of cells, PDDL does not support complex types such as matrices. Therefore, we represent the state of the game by means of a graph. We consider objects of the following types: *location* to represent a grid cell location; the `up`, `down`, `left` and `right` *direction*s used both to relate locations and to specify movements; and *pattern* to represent patterns of blocks in the scenario. Note that we do not define a block object in the domain: blocks are instead represented by patterns assigned to locations. The state is defined with the following predicates:

```
  (patterned ?l - location ?p - pattern)
  (next ?from ?to - location ?dir - direction)
  (free ?l - location)
  (falling_flag)
  (matching_flag)
```

The `patterned` predicate states the pattern assigned to a given location. With the `next` predicate we state what location we find following a certain direction from a location. In other words, we treat the `next` predicate as a declarative specification of the game grid adjacency graph. Further, because we can never move wall blocks, we exclude walls from the grid. This choice to only model the non-wall locations in the game grid also leads to a smaller state space. With the `free` predicate we state whether a certain location is free or not. We also use two “flag” predicates to capture the gravity and block matching semantics of the game. The `free`, `falling_flag` and `matching_flag` predicates are *derived predicates*, which are automatically updated after the application of each action:

```
; a block is free if it is not patterned
 (:derived (free ?l) ( forall (?p - pattern)  (not (patterned ?l ?p)) ))
; is there something that needs to fall?
 (:derived (falling_flag) (exists (?l1 ?l2 - location)
                            (and (next ?l1 ?l2 down) (not (free ?l1)) (free ?l2)) ))
; is there something that needs to match?
 (:derived (matching_flag) (exists (?l1 ?l2 - location ?p - pattern ?d - direction)
                             (and (next ?l1 ?l2 ?d) (patterned ?l1 ?p) (patterned ?l2 ?p)) ))
```

Actions are defined by their parameters as well as by their preconditions and effects which usually constrain the parameters. Preconditions define the requirements a state must satisfy in order for the action to be applicable. Effects define how actions change the state once an action has been applied. Three actions are defined: `move_block`, `fall_block` and `match_blocks`. The solving process of the planner needs to follow the semantics of the game, which can be concisely summarized with this solving algorithm: If there are blocks remaining then we have to consider the flags to decide which kind of action can be done. That is, if the `falling_flag` is active we only allow the `fall_block` action, otherwise if the `matching_flag` is active we only allow the `match_blocks` action. Finally, if no flag is active we then only allow the `move_block` action. To enforce that the planner adheres to the game semantics, we check the status of the flags in the actions' preconditions.

Now we present the `match_blocks` action. Modelling the matching of an arbitrary set of blocks as an atomic operation simplifies the representation of the game mechanics: A more granular model using pairs of blocks instead would require intermediate state tracking to take into account matches of more than 2 blocks.

```
(:action match_blocks
 :parameters ()
 :precondition (and (not (falling_flag)) (matching_flag)) ; first things fall, then they match
 :effect (and (forall (?l1 - location ?p - pattern)
                  (when  ; if a patterned locations has some neighbor with the same pattern
                    (exists (?l2 - location ?d - direction)
                      (and (next ?l1 ?l2 ?d) (patterned ?l1 ?p) (patterned ?l2 ?p)))
                    (not (patterned ?l1 ?p)))))) ; remove its pattern
```

Finally, as the goal is to remove all patterned blocks from the grid, we want to reach a state where no location has a pattern, additionally asking for the minimum number of moves.

```
   (:goal  ( forall (?l - location) (not (exists (?p - pattern) (patterned ?l ?p))) ))
   (:metric minimize (total-cost))
```

In contrast with matching, gravity is handled by moving one block at a time. That is, the `fall_block` action moves a single block one position down if it has nothing under it.

```
(:action fall_block
  :parameters (?l1 ?l2 - location ?p - pattern)
  :precondition (and
    (falling_flag)      ; something needs to fall
    (next ?l1 ?l2 down) ; l1 is on top of l2
    (patterned ?l1 ?p)  ; l1 has some pattern and needs to fall
    (free ?l2))         ; l2 is free as we're falling on it
  :effect (and ; the patterns get properly assigned: l1 loses the pattern and l2 gains the pattern l1 had
              (not (patterned ?l1 ?p)) (patterned ?l2 ?p)))
```

Compressing Fall Moves

The PDDL model above produces plans interleaved with long lists of trivial `fall_block` actions. We explored compressing long lists of actions such as these, starting with `fall_block`. More concretely, all the needed falling of a single block would be dealt with in one action. The first step is to define a derived predicate `(below ?x ?y - location)`, which is true when `y` is somewhere below `x` in the same column.

```
 (:derived (below ?x ?y - location) ; ?y is strictly below x (that is, <)
    (or (next ?x ?y down)
        (exists (?z - location)  (and (next ?x ?z down)  (below ?z ?y))))) ;  x > z and z > y
```

Then, a new fall move can be defined where a patterned block at a location `s` falls to another location `t` that is below `s`. It is required that all locations between `s` and `t` are free, and the location directly under `t` is not free, as follows.

```
(:action fall_block
 :parameters (?s ?t - location ?p - pattern)
 :precondition (and
   (falling_flag)    ; something needs to fall
   (below ?s ?t)     ; target is below source
   (free ?t)         ; we're falling somewhere that is free
   (patterned ?s ?p) ; source has really the pattern we have
   ; all the blocks in the falling path (between source and target) are empty
   (forall (?l - location)
      (imply (and (below ?s ?l) ; l is below the source of the fall
                (below ?l ?t))   ; target is below the location
              (free ?l)))
    ; there must be a block or wall under target
    ; forall locations, if its immediately under our target, must not be empty
    (forall (?l - location)
        (imply (next ?t ?l down) (not (free ?l)))))
  :effect  (and ; s loses the pattern and t gains the pattern of s
                (not (patterned ?s ?p)) (patterned ?t ?p)))
```

Surprisingly, preliminary experiments have shown that compressing fall moves substantially degrades the performance of the planners we considered, in particular in instances that are tall (e.g. Giraffes and Eagles, see Section 6). We hypothesise that this is caused by the increase in the number of generated ground actions, as the number of fall actions grows quadratically with height (one step falls, two step falls . . .), and the planner preprocessor cannot discard any of those. Instead, fall actions for each individual step greatly reduce the branching factor for the planner. Consequently, we have not included the compressed falls model in the experiments reported below.

4 A Constraint Model of Puzznic

Our constraint model is formulated in Essence Prime [17], exploiting this richer language to feature a number of abstractions that reduce the number of plan steps, and so decision variables required. This includes how gravity is captured, a key feature of our model, which allows it to be applied instantaneously after either a move or a match. Our model also supports partial parallelism via *compact row* moves, where multiple blocks in the same row may move several grid cells simultaneously in one time step. We begin by describing the model parameters:

```
letting WALL be 0
letting EMPTY be 1
given initGrid : matrix [int(1..gridHeight), int(1..gridWidth)] of int(WALL, EMPTY)
letting GRIDCOLS be domain int(1..gridWidth)
letting INTERIORCOLS be domain int(2..gridWidth-1)
letting GRIDROWS be domain int(1..gridHeight)
letting INTERIORROWS be domain int(2..gridHeight-1)

$ Initial positions of the patterned blocks, in ascending pattern order. Format: row, col
given initPatternedBlocks : matrix [int(1..noPatternedBlocks), int(1..2)] of int(1..)
letting PATTERNEDBLOCKS be domain int(1..noPatternedBlocks)
given patternBands : matrix indexed by [int(1..noPatterns), int(1..2)] of PATTERNEDBLOCKS

given noSteps : int(1..)
letting STEPSFROM1 be domain int(1..noSteps)
letting STEPSFROM0 be domain int(0..noSteps)
letting STEPSEXCEPTLAST be domain int(0..noSteps-1)
letting INTERIORSTEPS be domain int(1..noSteps-1)
```

Parameter `initGrid` gives the locations of walls in the grid. It is assumed that `initGrid` has a perimeter of wall blocks. The coordinates of each patterned block are given in `initPatternedBlocks`, and `patternBands` provides the patterned block types as intervals. For example, if `patternBands` is `[[1,3],[4,6]]` then we have 6 patterned blocks in total, with 3 blocks each of two patterns. The parameters `gridWidth`, `gridHeight`, `noPatternedBlocks`, and `noPatterns` are inferred automatically from the dimensions of the given matrices. In common with many constraint models of planning problems (e.g. Plotting [8]) we solve a sequence of decision problems of increasing `noSteps`. The first such instance for which a solution is found provides the optimal length plan.

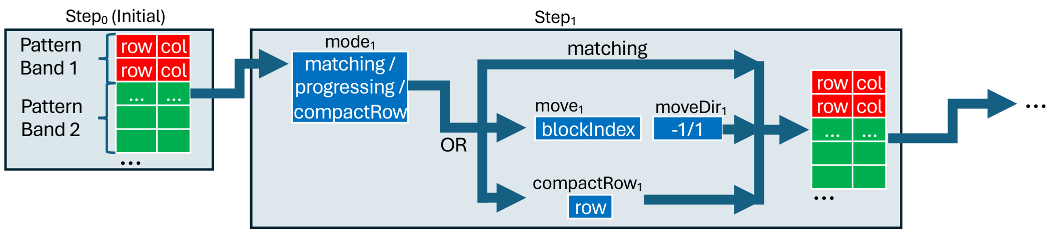


Figure 2 Constraint model structure: interleaved state, mode variables for flow of control, and action variables, annotated with their domains. Auxiliary variables not depicted for clarity.

4.1 Viewpoint, Initial and Goal States

Following a common pattern in constraint models of AI planning problems [15], we employ a time-indexed set of variables, interleaving the state of the puzzle with the action taken to transform the previous state into that following (Figure 2). The simplest state representation is a time-indexed representation of the full grid state. We found this to be too cumbersome to reason about long plans over larger grids. Instead, since much of the grid state (walls, empty cells) is fixed we maintain only the coordinates of each patterned block:

```
letting REMOVED be 0
find patternedBlocksRow : matrix indexed by[STEPSFROM0, PATTERNEDBLOCKS]
  of INTERIORROWS union int(REMOVED)
find patternedBlocksCol : matrix indexed by[STEPSFROM0, PATTERNEDBLOCKS]
  of INTERIORCOLS union int(REMOVED)
```

The initial and goal states are stated simply on this viewpoint:

```
$ Initial state:
forAll b : PATTERNEDBLOCKS . patternedBlocksRow[0, b] = initPatternedBlocks[b,1],
forAll b : PATTERNEDBLOCKS . patternedBlocksCol[0, b] = initPatternedBlocks[b,2],

$ Goal state:
forAll b : PATTERNEDBLOCKS . patternedBlocksRow[noSteps, b] = REMOVED,
forAll b : PATTERNEDBLOCKS . patternedBlocksCol[noSteps, b] = REMOVED,
```

4.2 Matching Mode

The model operates in three disjoint *modes*:

```
letting MATCHING_MODE be 0
letting PROGRESSING_MODE be 1
letting ROWCOMPACT_MODE be 2
letting MODES be domain int(MATCHING_MODE, ROWCOMPACT_MODE, PROGRESSING_MODE)
find mode : matrix indexed by[STEPSFROM1] of MODES
```

The first of these is matching mode, and is triggered by any pair of patterned blocks being adjacent horizontally or vertically at the previous time step. We introduce auxiliary Booleans `matchingGrid` to detect this state:

```
find matchingGrid : matrix indexed by[STEPSEXCEPTLAST, PATTERNEDBLOCKS] of bool

forAll step : STEPSEXCEPTLAST .
  forAll p : PATTERNS .
    forAll i : int(patternBands[p,1]..patternBands[p,2]) .
      (exists j : int(patternBands[p,1]..patternBands[p,2]) .
        ((j != i) /\
         (((patternedBlocksRow[step,i] = patternedBlocksRow[step,j]) /\
           (patternedBlocksCol[step,i] - patternedBlocksCol[step,j] = 1)) \/
          ((patternedBlocksCol[step,i] = patternedBlocksCol[step,j]) /\
           (patternedBlocksRow[step,i] - patternedBlocksRow[step,j] = 1)))))
      <->
      (matchingGrid[step,i]),
```

Matching mode at time step `t` is then forced according to the state of the `matchingGrid` at time step `t-1`:

```
forAll step : STEPSFROM1 .
  (mode[step] = MATCHING_MODE) <-> (sum(flatten(matchingGrid[step-1,..])) > 0),
```

The matching blocks are then removed:

```
forAll step : STEPSFROM1 .
  (mode[step] = MATCHING_MODE) ->
  (forAll b : PATTERNEDBLOCKS .
     (matchingGrid[step-1,b]) ->
     (patternedBlocksRow[step,b] = REMOVED) /\ (patternedBlocksCol[step,b] = REMOVED)),
```

As a consequence of these matches and removals, we must capture the effects of gravity, as well as ensure that unaffected blocks are unchanged. Rather than attempting to calculate the precise positions of the blocks, we model gravity elegantly via a declarative description of the blocks' behaviour:

```
$ Unmatched, unremoved blocks must stay on the grid, and in the same column
forAll step : STEPSFROM1 .
  (mode[step] = MATCHING_MODE) ->
  (forAll b : PATTERNEDBLOCKS .
    (!(matchingGrid[step-1,b]) /\ (patternedBlocksRow[step-1,b] != REMOVED)) ->
    ((patternedBlocksRow[step,b] != REMOVED) /\
     (patternedBlocksCol[step,b] = patternedBlocksCol[step-1,b]))),

$ Unmatched block must stay above: All unmatched blocks in same col, which it was above before.
forAll step : STEPSFROM1 .
  (mode[step] = MATCHING_MODE) ->
  (forAll b : PATTERNEDBLOCKS .
    (!(matchingGrid[step-1,b])) ->
    (forAll b2 : PATTERNEDBLOCKS .
       ((b2 != b) /\
        (!(matchingGrid[step-1,b2])) /\
        (patternedBlocksCol[step-1,b2] = patternedBlocksCol[step-1,b]) /\
        (patternedBlocksRow[step-1,b2] > patternedBlocksRow[step-1,b]))
       ->
       (patternedBlocksRow[step,b2] > patternedBlocksRow[step,b]))),

$ Unmatched block must stay above/below wall blocks it was above/below before
forAll step : STEPSFROM1 .
  (mode[step] = MATCHING_MODE) ->
  (forAll b : PATTERNEDBLOCKS .
    (!(matchingGrid[step-1,b])) ->
    (forAll row : INTERIORROWS .
      (initGrid[row, patternedBlocksCol[step-1,b]] = WALL) ->
      ((row < patternedBlocksRow[step-1,b]) -> (row < patternedBlocksRow[step,b])) /\
      ((row > patternedBlocksRow[step-1,b]) -> (row > patternedBlocksRow[step,b])))),

$ Common to all modes: No floating blocks - either wall there or there exists another block.
forAll step : STEPSFROM1 .
  forAll b : PATTERNEDBLOCKS .
    (patternedBlocksRow[step,b] != REMOVED) ->
    ((initGrid[patternedBlocksRow[step,b]-1, patternedBlocksCol[step,b]] = WALL) \/
     (exists b2 : PATTERNEDBLOCKS .
       (b != b2) /\
       (patternedBlocksRow[step,b2] = patternedBlocksRow[step,b] - 1) /\
       (patternedBlocksCol[step,b2] = patternedBlocksCol[step,b]))),
```

The above sets of constraints simply require unmatched blocks in a column to maintain their relative ordering, both with each other and the wall cells in the grid, and disallows any block from floating above an empty cell. Constraint propagation then ensures that a column where blocks have been removed 'settles' according to the effects of gravity.

4.3 Progressing Mode

In progressing mode a committal player action is taken: a block is selected and moved so as to cause it to fall or to cause a match at the next time step. We introduce variables to capture this choice:

```
find move : matrix [STEPSFROM1] of PATTERNEDBLOCKS union int(0)
find moveDir : matrix [STEPSFROM1] of int(-1,1)
```

The domain of `move` is the indices of the patterned blocks. A dummy value 0 is added for when in another mode. `moveDir` indicates a left or a right move. The following set of constraints specify a valid progressing move, transforming the state at time step `t-1` to time step `t`:

```
$ Select only valid blocks
forAll step : STEPSFROM1 .
  (mode[step] = PROGRESSING_MODE) -> (patternedBlocksRow[step-1,move[step]] != REMOVED),

$ destination column defined via moveDir
forAll step : STEPSFROM1 .
  (mode[step] = PROGRESSING_MODE) ->
  (patternedBlocksCol[step,move[step]] = patternedBlocksCol[step-1,move[step]]+moveDir[step]),

$ destination row must be at or below moveRow
forAll step : STEPSFROM1 .
  (mode[step] = PROGRESSING_MODE) ->
  (patternedBlocksRow[step,move[step]] <= patternedBlocksRow[step-1,move[step]]),

$ in destination column, everything from source row to destination row must be empty.
forAll step : STEPSFROM1 .
  (mode[step] = PROGRESSING_MODE) ->
  (forAll row : INTERIORROWS .
     ((row <= patternedBlocksRow[step-1,move[step]]) /\ (row >= patternedBlocksRow[step,move[step]])) ->
     ((initGrid[row, patternedBlocksCol[step,move[step]]] = EMPTY) /\
       (forAll b : PATTERNEDBLOCKS .
         ((patternedBlocksRow[step-1, b] != row) \/
          (patternedBlocksCol[step-1, b] != patternedBlocksCol[step,move[step]]))))),
```

Frame axioms fix unaffected blocks in place:

```
$ Frame axiom: blocks not in source col stay in the same place.
forAll step : STEPSFROM1 .
  (mode[step] = PROGRESSING_MODE) ->
  (forAll b : PATTERNEDBLOCKS .
     (patternedBlocksCol[step-1, b] != patternedBlocksCol[step-1, move[step]]) ->
     ((patternedBlocksCol[step-1, b] = patternedBlocksCol[step, b]) /\
      (patternedBlocksRow[step-1, b] = patternedBlocksRow[step, b]))),

$ Frame axiom: blocks in source col underneath that selected stay in the same place.
$ Simplified to all blocks whose row is less than that selected.
forAll step : STEPSFROM1 .
  (mode[step] = PROGRESSING_MODE) ->
  (forAll b : PATTERNEDBLOCKS .
     (patternedBlocksRow[step-1, b] < patternedBlocksRow[step-1, move[step]]) ->
     ((patternedBlocksCol[step-1, b] = patternedBlocksCol[step, b]) /\
      (patternedBlocksRow[step-1, b] = patternedBlocksRow[step, b]))),
```

Gravity is handled similarly to matching mode:

```
$ Gravity: if above selected block and no wall in the way, fall one cell.
forAll step : STEPSFROM1 .
  (mode[step] = PROGRESSING_MODE) ->
  (forAll b : PATTERNEDBLOCKS .
     ((patternedBlocksRow[step-1, b] > patternedBlocksRow[step-1, move[step]]) /\
      (patternedBlocksCol[step-1, b] = patternedBlocksCol[step-1, move[step]]) /\
      (forAll row : INTERIORROWS .
         ((row < patternedBlocksRow[step-1, b]) /\
          (row > patternedBlocksRow[step-1, move[step]])) ->
         (initGrid[row, patternedBlocksCol[step-1, move[step]]] = EMPTY)))
     ->
     ((patternedBlocksCol[step, b] = patternedBlocksCol[step-1, b]) /\
      (patternedBlocksRow[step, b] = patternedBlocksRow[step-1, b] - 1))),

$ Gravity: if above selected block and wall in the way, stay in the same location.
forAll step : STEPSFROM1 .
  (mode[step] = PROGRESSING_MODE) ->
  (forAll b : PATTERNEDBLOCKS .
     ((patternedBlocksRow[step-1, b] > patternedBlocksRow[step-1, move[step]]) /\
      (patternedBlocksCol[step-1, b] = patternedBlocksCol[step-1, move[step]]) /\
      (exists row : INTERIORROWS .
```

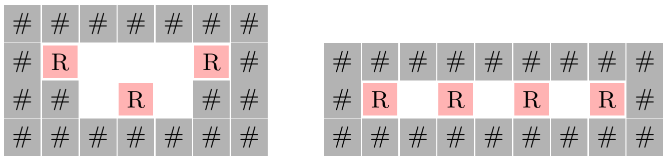


Figure 3 Illustrating why parallel falls and matches must be disallowed. The left instance has no solution. The right instance has a solution, but compact row moves to initiate matches at each end of the grid would give a solution that is not possible to physically realise in the game.

```
         ((row < patternedBlocksRow[step-1, b]) /\
          (row > patternedBlocksRow[step-1, move[step]]) /\
          (initGrid[row, patternedBlocksCol[step-1, move[step]]] = WALL))))
     ->
     ((patternedBlocksCol[step, b] = patternedBlocksCol[step-1, b]) /\
      (patternedBlocksRow[step, b] = patternedBlocksRow[step-1, b]))),
```

4.4 Row Compact Mode

In row compact mode, the blocks of a selected row are moved horizontally, while remaining in the same row and not triggering a match at the next step. We introduce variables to capture the row selection (again, a dummy value of 0 is added for when in another mode):

```
find compactRow : matrix indexed by[STEPSFROM1] of INTERIORROWS union int(0)
```

This last mode allows significant parallelism in the plan, but note that it is necessary (rather than a choice) to disallow it from creating either falling blocks or trigger matches. In the former case, it would then be possible to create blocks falling in parallel in a way that is not possible for a human player (see Figure 3, left). In the latter, it would be possible to create parallel matches (e.g. at two ends of a row) that are again not possible in the game itself (Figure 3, right). In both cases, the result could be the generation of invalid solutions.

Modelling row compact moves resembles our approach to gravity: a declarative description of the rules that the blocks in a selected row must respect, leaving search and propagation to decide the details. First we maintain the relative order among patterned and wall blocks:

```
$ Stay on the same side of all wall blocks on the same row.
forAll step : STEPSFROM1 .
  (mode[step] = ROWCOMPACT_MODE) ->
  (forAll col : INTERIORCOLS .
     (initGrid[compactRow[step], col] = WALL) ->
     (forAll block : PATTERNEDBLOCKS .
        (patternedBlocksRow[step-1, block] = compactRow[step]) ->
        (((patternedBlocksCol[step-1, block] < col) -> (patternedBlocksCol[step, block] < col)) /\
         ((patternedBlocksCol[step-1, block] > col) -> (patternedBlocksCol[step, block] > col))))),

$ Maintain order on the blocks in the chosen row.
forAll step : STEPSFROM1 .
  (mode[step] = ROWCOMPACT_MODE) ->
  (forAll block : PATTERNEDBLOCKS .
     (patternedBlocksRow[step-1, block] = compactRow[step]) ->
     (forAll block2 : int(block + 1 .. noPatternedBlocks) .
        (patternedBlocksRow[step-1, block2] = compactRow[step]) ->
        (((patternedBlocksCol[step-1, block] < patternedBlocksCol[step-1, block2]) ->
          (patternedBlocksCol[step, block] < patternedBlocksCol[step, block2])) /\
         ((patternedBlocksCol[step-1, block] > patternedBlocksCol[step-1, block2]) ->
          (patternedBlocksCol[step, block] > patternedBlocksCol[step, block2]))))),
```

We disallow movement over a block of the same pattern, which would have triggered a match.

```
$ We exploit the pattern bands to compare only blocks of like pattern.
forAll step : STEPSFROM1 .
  (mode[step] = ROWCOMPACT_MODE) ->
  (forAll pattern : PATTERNS .
     forAll block : int(patternBands[pattern,1]..patternBands[pattern,2]) .
       (patternedBlocksRow[step-1, block] = compactRow[step]) ->
       (forAll block2 : int(patternBands[pattern,1]..patternBands[pattern,2]) .
          (patternedBlocksRow[step-1, block2] = compactRow[step]-1) ->
          (((patternedBlocksCol[step-1, block] < patternedBlocksCol[step-1, block2]) ->
            (patternedBlocksCol[step, block] < patternedBlocksCol[step, block2])) /\
           ((patternedBlocksCol[step-1, block] > patternedBlocksCol[step-1, block2]) ->
            (patternedBlocksCol[step, block] > patternedBlocksCol[step, block2]))))),
```

We must avoid initiating falls:

```
$ We cannot move a block onto or over a gap that would cause it to fall.
forAll step : STEPSFROM1 .
  (mode[step] = ROWCOMPACT_MODE) ->
  (forAll block : PATTERNEDBLOCKS .
     (patternedBlocksRow[step-1, block] = compactRow[step]) ->
     (forAll col : INTERIORCOLS .
        ((initGrid[compactRow[step]-1, col] = EMPTY) /\
         (forAll block2 : PATTERNEDBLOCKS .
            (patternedBlocksRow[step-1,block2] != compactRow[step]-1) \/
            (patternedBlocksCol[step-1,block2] != col)))
        ->
        (((col < patternedBlocksCol[step-1, block]) -> (col < patternedBlocksCol[step, block])) /\
         ((col > patternedBlocksCol[step-1, block]) -> (col > patternedBlocksCol[step, block]))))),
$ Anything covered by another block stays where it is.
$ Don't need to guard with compactRow because it is true irrespective.
forAll step : STEPSFROM1 .
  (mode[step] = ROWCOMPACT_MODE) ->
  (forAll block : PATTERNEDBLOCKS .
     (exists block2 : PATTERNEDBLOCKS .
       ((patternedBlocksCol[step-1, block] = patternedBlocksCol[step-1, block2]) /\
        (patternedBlocksRow[step-1, block2] = patternedBlocksRow[step-1, block] + 1))
     ->
     (patternedBlocksCol[step, block] = patternedBlocksCol[step-1, block])),
```

Finally, we disallow row compact mode from initiating a match on the same row, and break a symmetry in the model by ensuring that even single-block moves that could have been captured by the progressing mode infrastructure are labelled as row compact if they do not lead to a match:

```
$ A move that does not lead to a match should be labelled row compact
forAll step : INTERIORSTEPS . (mode[step] = ROWCOMPACT_MODE) -> (mode[step+1] != MATCHING_MODE),
$ A move that leads to a match should be labelled progressing
forAll step : INTERIORSTEPS .
  (mode[step] = PROGRESSING_MODE) ->
  ((mode[step+1] = MATCHING_MODE) \/
   (exists b : PATTERNEDBLOCKS. patternedBlocksRow[step,b] < patternedBlocksRow[step-1,b])),
```

4.5 Symmetry and Dominance Breaking, Implied Constraints

To complete our model, we add symmetry, dominance-breaking, and implied constraints. A simple dominance condition that we can exploit is to disallow the solver from returning to exactly the same state as at a previous time step, since a plan with only the first occurrence of that state must be at least as short. Given our compact representation of state in terms of the patterned block coordinates, this can be achieved simply by requiring the coordinate of at least one patterned block to be different between all pairs of states in the plan.

```
forAll step : STEPSFROM0 .
  forAll step2 : int(step+1..noSteps) .
    exists block : PATTERNEDBLOCKS .
      ((patternedBlocksRow[step, block] != patternedBlocksRow[step2, block]) \/
       (patternedBlocksCol[step, block] != patternedBlocksCol[step2, block])),
```

In addition to the symmetry in the modes described in the previous subsection, there is the potential for conditional symmetry among the values of the action variables when they are not active. We fix them to their dummy values to avoid this:

```
$ Pin rowCompact and movement variables to break symmetry
forAll step : STEPSFROM1 .
  (mode[step] = MATCHING_MODE) ->
   ((move[step] = 0) /\ (moveDir[step] = -1) /\  (compactRow[step] = 0)),

$ Pin rowCompact variable to break symmetry
forAll step : STEPSFROM1 .
  (mode[step] = PROGRESSING_MODE) -> (compactRow[step] = 0),
```

```
$ Pin movement variables to break symmetry
forAll step : STEPSFROM1 .
  (mode[step] = ROWCOMPACT_MODE) -> ((move[step] = 0) /\ (moveDir[step] = -1)),
```

There is no mechanism by which a block can move upwards, which can be added as an implied constraint by insisting that the row coordinate of each patterned block decreases monotonically. The final step of the plan returned must be in matching mode, leading to the removal of the final remaining blocks. This is captured with a simple unary constraint.

```
forAll step : STEPSFROM1 .
  forAll block : PATTERNEDBLOCKS .
    (patternedBlocksRow[step, block] <= patternedBlocksRow[step-1, block]),
$ The final step must be in matching mode, to derive the last match and a clear grid.
mode[noSteps] = MATCHING_MODE,
```

5 A Constraint Model of Puzznic Instance Generation

We modify our constraint model to generate Puzznic instances, to aid in level design to challenge human players, for benchmarking or to train an algorithm selection approach in future. Rather than giving an initial grid as a parameter, the model is modified to *find* an initial state such that a plan of a specified length exists. The parameters are the grid dimensions and the number of patterns and patterned blocks. The initial grid then becomes a set of decision variables, along with the pattern bands. Additional variables are not needed for the initial coordinates of the patterned blocks, since the model already has these for time step 0. One use case is to increase `noSteps` iteratively to search for an instance of the specified grid dimensions and patterned blocks with a proven minimum solution length. Alternatively, we can search for a configuration that admits a k-step plan, with the caveat that a solution shorter than k steps may be possible.

```
given gridWidth, gridHeight : int(1..)
given noPatterns : int (1..)
letting PATTERNS be domain int(1..noPatterns)
given noPatternedBlocks : int(2*noPatterns..)              $ At least a pair of blocks per pattern
letting PATTERNEDBLOCKS be int (1..noPatternedBlocks)
given noSteps : int(1..)

find initGrid : matrix indexed by[int(1..gridHeight), int(1..gridWidth)] of int(WALL, EMPTY)
find patternBands : matrix indexed by [PATTERNS, int(1..2)] of PATTERNEDBLOCKS
```

Constraints are added over the initial state in order to find valid instances. First, we must ensure that the initial positions of the patterned blocks are on empty cells:

```
$ Connect initGrid to patternedBlocks at step 0
forAll block:PATTERNEDBLOCKS .
  initGrid[patternedBlocksRow[0,block], patternedBlocksCol[0,block]] = EMPTY,
```

We insist on a perimeter wall around the grid, and that the pattern bands are valid:

```
$ perimeter wall$
forAll row : GRIDROWS .
  (initGrid[row, 1] = WALL) /\ (initGrid[row, gridWidth] = WALL),
forAll col : GRIDCOLS .
  (initGrid[1, col] = WALL) /\ (initGrid[gridHeight, col] = WALL),
$ Start and end of pattern bancds are fixed
patternBands[1,1] = 1,
patternBands[noPatterns,2] = noPatternedBlocks,
$ Pattern band entries are ordered
forAll p : PATTERNS . patternBands[p,1] < patternBands[p,2],
$ Pattern bands must have at least two blocks
forAll p : int(1..noPatterns-1) . patternBands[p,2] = patternBands[p+1,1] - 1,
```

We require that the initial state does not trigger matching mode at the first step, to avoid trivial instances. Although not required, we disallow interior rows and columns from being entirely wall blocks to promote the use of the whole grid as specified:

```
mode[1] != MATCHING_MODE,
forAll row : INTERIORROWS . sum(initGrid[row,..]) > 0,
forAll col : INTERIORCOLS . sum(initGrid[..,col]) > 0,
```

We remove trivially equivalent instances by disallowing "walled in" empty spaces, and breaking symmetry among the patterns and in the list of initial coordinates for each pattern:

```
$ No walled in empty spaces - removes trivially equivalent solutions.
forAll row : INTERIORROWS .
  forAll col : INTERIORCOLS .
    (initGrid[row,col] >= EMPTY) ->
    ((initGrid[row+1,col] != WALL) \/ (initGrid[row-1,col] != WALL) \/
     (initGrid[row,col+1] != WALL) \/ (initGrid[row,col-1] != WALL)),
$ Symmetry Breaking: lex order within pattern bands.
forAll p : PATTERNS .
  forAll b1 : PATTERNEDBLOCKS .
    forAll b2 : int(b1+1..noPatternedBlocks) .
      ((b1 >= patternBands[p,1]) /\ (b2 <= patternBands[p,2])) ->
      ([patternedBlocksRow[0,b1],patternedBlocksCol[0,b1]] <=lex
       [patternedBlocksRow[0,b2],patternedBlocksCol[0,b2]]),
$ Symmetry Breaking: order first element of each pattern band
forAll p1 : PATTERNS .
  forAll p2 : int(p1+1..noPatterns) .
    forAll b1 : PATTERNEDBLOCKS .
      forAll b2 : int(b1+1..noPatternedBlocks) .
        ((b1 = patternBands[p1,1]) /\ (b2 = patternBands[p2,2])) ->
        ([patternedBlocksRow[0,b1],patternedBlocksCol[0,b1]] <=lex
         [patternedBlocksRow[0,b2],patternedBlocksCol[0,b2]]),
```

Figure 4 presents illustrative instances produced by our generator model. These are found efficiently with our model encoded into SAT via Savile Row. The largest of the three takes just over 7 minutes to be found on a 2021 MacBook Pro (M1, 32GB RAM).

Despite the fact that the initial grid is almost completely full, the first has a solution of 6 steps (3 player moves, 3 matching steps). The second is a more intricate design with three patterns that admits a simple solution: move the right red block left to initiate two cascaded matches. For the third we added the constraint that there must be a wall block on the interior of each row, demonstrating the flexibility of the constraint-based instance generation method. The 20-block instance produced has an 8-step solution (4 progressing moves, one compact row move, and three matching steps). Our instance generator could

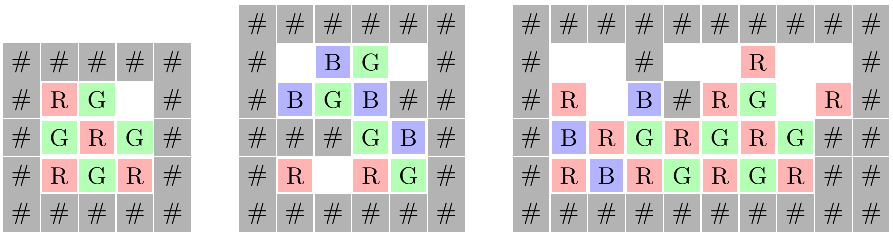


Figure 4 Illustrative instances produced by our instance generator model.

straightforwardly be adapted for further flexibility, for instance to generate patterned block positions only in a given grid.

6 Empirical Evaluation

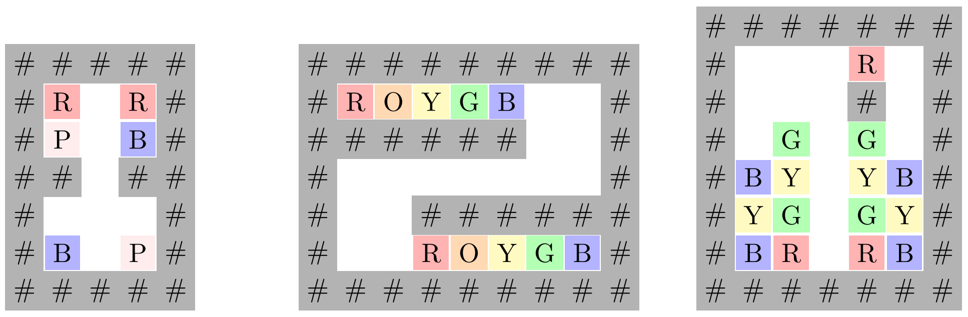
We present an empirical evaluation of our two models.The plans produced by the two models are not directly comparable in terms of length. As we have discussed, the AI Planning model is fine-grained whereas the constraint model implements gravity instantaneously and also permits some parallelism (leading to plans with fewer steps). We can, however, observe the time taken by each approach to produce the optimal plan from its own perspective.

Experiments were run on an Intel i5 quad-core processor with 2GHz maximum clock speed and 32GB RAM, under MacOS 14.3.1. For PDDL, we used Fast Downward 23.06+ [13] as our planner, modified to always approximate negative axioms and using A* search with a blind heuristic, with a 1 hour timeout. For the Essence Prime model we used the natively compiled version of Savile Row 1.10.0 [18] with options `-run-solver -O0 -deletevars -identical-cse -sat -sat-polarity`, a timeout of 600s per time step, a 1 hour overall timeout, and Kissat 3.1.1 [3] as the backend solver. For each Puzznic instance, Savile Row is called repeatedly with an increasing number of steps in a solution until a solution is found.

Table 1 presents the results of our evaluation across five families of Puzznic instances. Overall, we observe that the AI planning and constraint-based approaches have complementary strengths, with neither dominant. In particular, performance varies according to the instance family and with instance size within a family.

General instances are from versions of the game and are intended for human players. We observe that the planning approach performs well on these, and some of the more difficult instances are challenging for our CP approach. On some of the most difficult such instances both approaches time out (not shown in the table). However, consider now the *Caterpillar* instances (see Figure 5c for reference). Each has many possible moves, but requires reasoning about a complex chain of matches to see a one-move solution triggering a cascade of falls and matches. Caterpillars also show that arbitrarily many blocks can match at any one time. Segments can be added to the body of the caterpillar by duplicating the two middle rows of the instance. Instances with more segments have more possible moves for a solver to consider. For the caterpillar instances we observe the reverse pattern: although the planning approach performs well on small caterpillar instances, the constraint model scales significantly better.

Our remaining experiments are on families generated by modifying existing instances in order to gain insight into how instance features affect performance. The *Eagle* and *Giraffe*



(a) Eagle/giraffe 5x7-ps1-a13 **(b)** Snake 9x7-cubic-10 **(c)** Caterpillar colinmirrormini

Figure 5 Some sample instances from different families.

Family	Instance	FD	SAT	Family	Instance	FD	SAT
general	9x5-ps1-b22	**0.75**	4.05	snake	9x7-cubic-2	**1.01**	1.52
general	10x5-ps1-b11	**0.87**	22.00	snake	73x7-cubic-2	23.35	**3.28**
general	5x7-ps1-a13	**0.95**	3.70	snake	9x7-cubic-4	**1.00**	6.43
general	10x7-bcl-014-2	**1.06**	6.95	snake	73x7-cubic-4	3571.98	**27.60**
general	9x7-ps1-b12	**1.10**	141.83	snake	9x7-cubic-6	**1.87**	37.85
general	8x12-ps1-e22	**2.03**	1238.86	snake	28x7-cubic-6	TO	**113.41**
caterpillar	colinsmall	**0.98**	2.49	snake	73x7-cubic-6	TO	**281.35**
caterpillar	colinmirrorsm	**1.15**	5.93	snake	9x7-cubic-8	**3.73**	457.37
caterpillar	colin	236.54	**14.69**	snake	11x7-cubic-8	1880.42	**665.32**
caterpillar	colinmirror	TO	**57.87**	snake	12x7-cubic-8	TO	**765.32**
giraffe	5x24-test	**1.93**	6.19	snake	43x7-cubic-8	TO	**2001.48**
giraffe	5x49-test	**6.07**	10.75	snake	9x7-cubic-10	**9.82**	TO
giraffe	5x99-test	23.66	**20.51**	eagle	5x50-test	**5.61**	5.95
giraffe	5x200-test	100.42	**43.06**	eagle	5x100-test	21.57	**9.22**

Table 1 Empirical results across five families of instances. Times in seconds with a 1-hour timeout (indicated by "TO"). FD refers to the Fast Downward planner, while SAT refers to Kissat via Savile Row. Names of the instances starting with $\_ \times \_$ refer to their dimensions. The number of blocks in a snake instance is the final component of its name.

instances are formed respectively by adding empty rows above, and interpolating empty rows in the middle of, an existing instance (here we have used Figure 5a as the base instance). We see that the CP approach continues scaling roughly linearly, while the planning approach degrades to approximately quadratic scaling. The *Snake* instances are versions of Figure 5b (taken from [9]) with the horizontal ledges stretched. Instance difficulty can also be varied by changing the number of blocks at the head and tail of the snake. The original instance (with 10 blocks) is challenging, and although the planning approach does produce a solution reasonably quickly, our CP approach does not complete within the timeout. Increasing width does not change the essential nature of the solutions (although plans will require more player moves), and also does not change the difficulty for humans. Our CP approach is able to take advantage of compact row moves to deal with the long horizontal distances that blocks need to traverse, and scales roughly linearly with the width. In contrast, the planning approach scales poorly, timing out when the original width of 9 is increased to 12 or more on the 8-block version, and times out on 6-block snakes with width 28.

7 Conclusion

In this work we have provided a challenging new benchmark, *Puzznic* and established its membership in NP. We have presented models for both AI planning and Constraint Programming, together with a constraint-based instance generator. Our empirical results demonstrate that these two approaches are complementary: primacy of one over the other depends on the sub-family of Puzznic instance considered, and we have established several such families. In future work, we will develop both of our approaches further. A static reachability analysis, for example, would yield information that both the constraint and planning models could exploit. In the context of the constraint model, we could recognise when the grid is in a symmetric state and exploit that situation to reduce search. Similarly, we could develop more dominance-breaking constraints to improve performance.

References

1 Roman Barták, Miguel A Salido, and Francesca Rossi. Constraint satisfaction techniques in planning and scheduling. *Journal of Intelligent Manufacturing*, 21(1):5–15, 2010. `doi:10.1007/s10845-008-0203-4`.

2 Roman Barták and Daniel Toropila. Reformulating constraint models for classical planning. In *FLAIRS*, pages 525–530. AAAI Press, 2008. URL: `https://cdn.aaai.org/FLAIRS/2008/FLAIRS08-121.pdf`.

3 Armin Biere, Katalin Fazekas, Mathias Fleury, and Maximillian Heisinger. CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020. In *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions*, volume B-2020-1, pages 51–53. University of Helsinki, 2020. URL: `http://hdl.handle.net/10138/318754`.

4 Tom Bylander. The computational complexity of propositional strips planning. *Artificial Intelligence*, 69(1):165–204, 1994. `doi:10.1016/0004-3702(94)90081-7`.

5 Michael C. Chavrimootoo. Defying gravity: The complexity of the Hanano puzzle. arXiv, 2022. `doi:10.48550/arXiv.2205.03400`.

6 Joan Espasa, Ian P. Gent, Ruth Hoffmann, Christopher Jefferson, Matthew J. McIlree, and Alice M. Lynch. Explaining Pen and Paper Puzzles with MUSes? In *Proceedings of the SICSA eXplainable Artifical Intelligence Workshop 2021*, volume 2894 of *CEUR Workshop Proceedings*, pages 56–63. CEUR-WS.org, 2021. URL: `http://ceur-ws.org/Vol-2894/short8.pdf`.

7 Joan Espasa, Ian P. Gent, Ian Miguel, Peter Nightingale, András Z. Salamon, and Mateu Villaret. Towards a Model of Puzznic. In *ModRef*, 2023. `doi:10.48550/arXiv.2310.01503`.

8 Joan Espasa, Ian Miguel, and Mateu Villaret. Plotting: a planning problem with complex transitions. In *28th International Conference on Principles and Practice of Constraint Programming (CP 2022)*, pages 22:1–22:17, 2022. `doi:10.4230/LIPIcs.CP.2022.22`.

9 Erich Friedman. The game of Cubic is NP-complete. 34th Annual Florida MAA Section Meeting, 2001. URL: `https://erich-friedman.github.io/papers/cubic.pdf`.

10 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. `https://content.iospress.com/articles/ai-communications/aic405`.

11 Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning: theory and practice*. Elsevier, 2004.

12 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. Springer, 2019. `doi:10.2200/S00900ED2V01Y201902AIM042`.

13 Malte Helmert. The fast downward planning system. *J. Artif. Intell. Res.*, 26:191–246, 2006. `doi:10.1613/JAIR.1705`.

14 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`.

15 H Kautz and B Selman. Planning as Satisfiability. In *Proceedings of ECAI*, pages 359–363, 1992. URL: `https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=0f8e74761d1c1638295d8c436efa189ad74e343b`.

16 Ziwen Liu and Chao Yang. Hanano puzzle is NP-hard. *Information Processing Letters*, 145:6–10, 2019. `doi:10.1016/j.ipl.2019.01.003`.

17 Peter Nightingale. Savile Row manual, 2021. `doi:10.48550/arXiv.2201.03472`.

18 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. `doi:10.1016/j.artint.2017.07.001`.

19 Helmut Simonis. Sudoku as a constraint problem. In *ModRef 2005: Workshop on Modeling and Reformulating Constraint Satisfaction Problems*, 2005. URL: `https://citeseerx.ist.psu.edu/document?doi=4f069d85116ab6b4c4e6dd5f4776ad7a6170faaf`.

A Proof of Complexity Claim

Claim

If a Static Puzznic instance has a solution, it has a solution where no block moves in two different directions in any sequence of moves where the fall-metric does not change.

Proof

Imagine a solution where there is a block which does a 'switchback' while the fall-metric does not change. It is enough to find another solution which contains one less pair of moves in which a block moves in both directions. This can be repeated until we have a solution with no switchbacks.

Consider a solution with any subsequence of moves not containing any falls or matches but containing a block moving in two directions. The pair of moves we eliminate will be the last pair of opposite moves. Specifically we look for the last move in the sequence of any block that subsequently moves in the opposite direction in the sequence. Call the block that moves and later returns block X. Without loss of generality, we assume that the pair of moves we eliminate switchback involves a block moving right-then-left. (If it only involved a block moving left then right, then simply mirror both the grid and solution.)

We will eliminate the pair of moves of X to give a new potential solution. So now consider the sequence of moves which is exactly the same as the original solution, except that we remove the final right hand move that X makes and the next left hand move X makes. (There might be many left hand moves that X makes but one of them must be first.) This clearly doesn't change the position that will result at the end of the subsequence of moves, as long as it doesn't invalidate any move between the two edited-out moves or cause any falls or matches. As a reminder, we know that none of the moves in the subsequence including the now-deleted moves can have involved a fall or match, and that all blocks in the remainder of the subsequence each move in only one direction.

The following is a useful lemma as it is necessary twice in the proof.

Lemma: Both locations underneath X must be occupied continuously while X does its switchback in the original sequence - i.e. the position under X's original position and the location one to the right.

Proof of Lemma: The position to the right clearly needs to be occupied to support X in the original sequence, until X returns back to the left. Less obviously, the position underneath X's original position must remain filled. If a block, say Z, moved from there it would have to move left. But we know that when X returns the location must be filled to support X, and Z is the only block that can be moved back to support Z. But this is impossible because we have already said that no block moves in two directions after X's move right.

Now, we consider anything that might happen that would make some move invalid in the new sequence of sideways moves, or cause a fall or match.

- No fall or match can be invalidated since none happens in this sequence.
- The removal of the two moves could conceivably stop some other block moving by the X's original space being occupied instead of empty. But this can't happen because in the original sequence, we were able to reverse the move of X to move it back to its original position. If a block Y had moved into X's position it would have had to move from left to right after X moved. Following that, Y can't have moved further right as that is blocked by X. Also Y can't move back left because it now only moves in one direction and has moved right since X moved. So the situation is impossible.
- The removal of X's moves could conceivably cause some match or fall to happen which

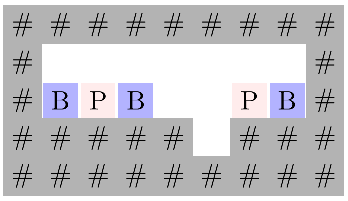
would otherwise not have happened when X doesn't move, i.e. in the new sequence we might get a match or fall that did not happen in the original sequence. Let's consider a fall first. There's two ways a fall can be created.

  - The first way this can happen is if X originally moved to the right, then some block Z moved on top of X. In the new sequence this would become a fall (because X is not there) when it was not originally a fall. But this is impossible: the block Z would still fall in the original sequence when X moved back left. If Z did not fall, then it would have had to move out of the way, but it cannot have moved either left or right. Z cannot move left because then it would move to X's original location which must have been empty as X moved back to it. It would have then fallen but we know it did not. Z cannot have moved right. To see this note first that Z moved leftwards to go on top of X after X had moved: the square to X's left was the space left by X so could not have supported Z. But to then move right, Z would have reversed direction since X moved.
  - The second way a fall can be created is that X is sitting on top of another block, say U. Originally X moved to the right and then U moved, but in the revised sequence the move of U causes X to fall from its original position. But this is impossible by the lemma, which states that the block U can't have moved.

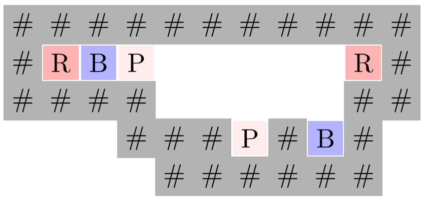
- The final case to consider is the possibility that a match occurs in the new sequence which did not in the original. First, note that the new match must involve the block X and a duplicate block. The reason is that X is the only block in the new sequence which is ever in a different position compared to the original sequence: thus any non-X match in the new sequence must happen in the old one, and none did.
  If the potential match was to X's right, we are safer now, because the block didn't move there. If the potential match was to X's left, then another X block must have moved from left to right, placing it immediately to the left of X's starting and final position. But the second X to the left can't have moved out of the way in the original sequence, as it would have to move back left, thereby causing a switchback which didn't happen. Could any block move on top of the unmoved X causing a match? No, because if it did so while X was originally to the right, then it would have fallen in the original sequence, which it didn't. Could any block move to be underneath X's original position, which was unoccupied in the original sequence? No, because of the lemma which states that the position was occupied continuously.

This completes the case analysis. We have shown the revised sequence is a valid sequence of moves which are all sideways moves, and which starts and ends with all blocks in the same places. If any switchbacks still occur we can just repeat this process until there are none. QED.

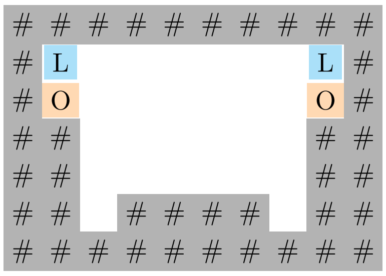
B Selected instances



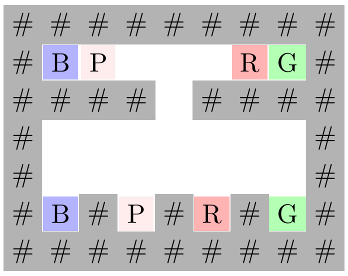
9x5-ps1-b22



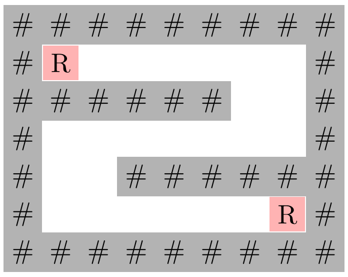
10x5-ps1-b11



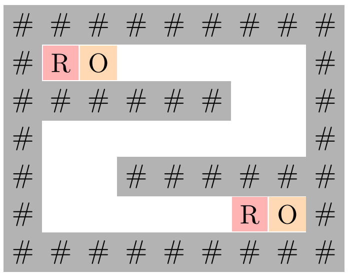
10x7-bcl-014-2



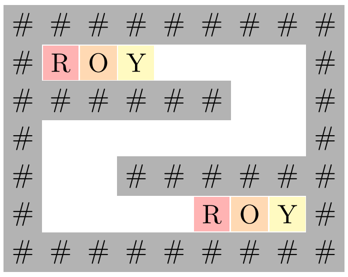
9x7-ps1-b12



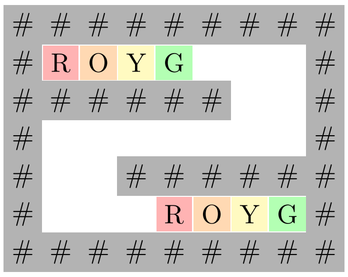
Snake 9x7-cubic-2



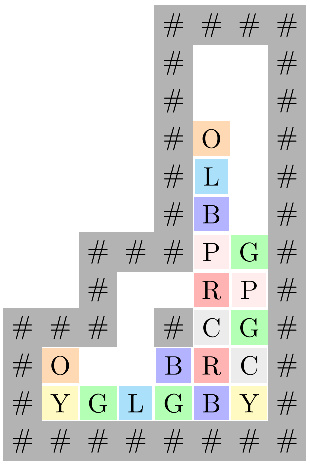
Snake 9x7-cubic-4

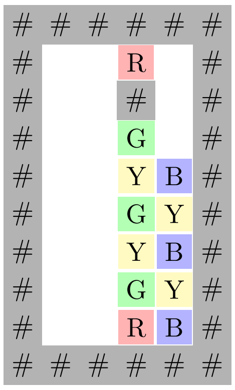


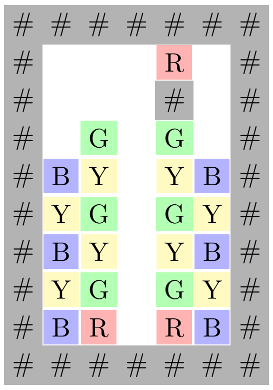
Snake 9x7-cubic-6

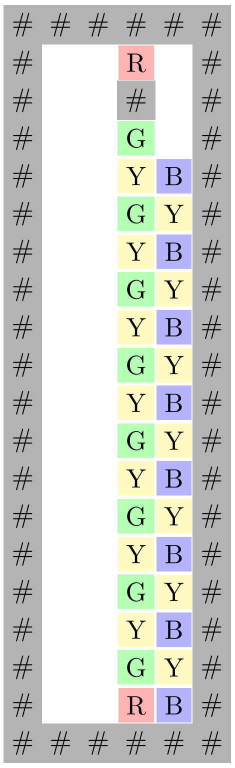


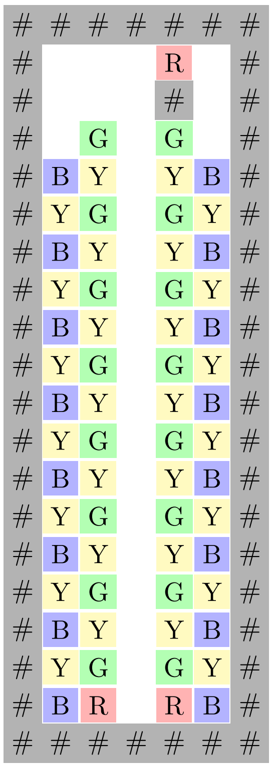
Snake 9x7-cubic-8

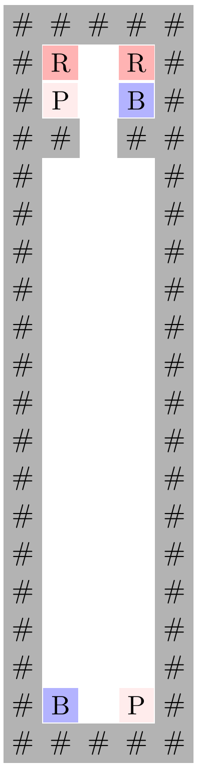

8x12-ps1-e22

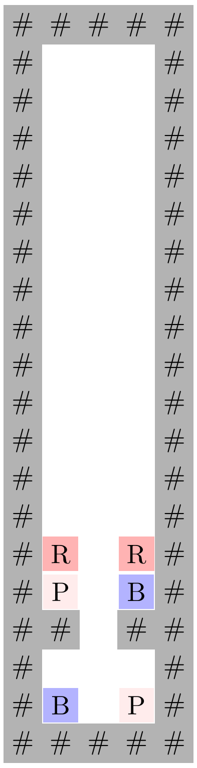

Caterpillar colinsmall


Caterpillar colinmirrorsm


Caterpillar colin


Caterpillar colinmirror


Giraffe 5x20-test


Eagle 5x20-test2