Cross-Paradigm Modelling: A Study of Puzznic

Joan Espasa
School of Computer Science
University of St Andrews, UK
https://orcid.org/0000-0002-9021-3047

Ian P. Gent
School of Computer Science
University of St Andrews, UK
https://orcid.org/0000-0002-5604-7006

Ian Miguel
School of Computer Science
University of St Andrews, UK
https://orcid.org/0000-0002-6930-2686

Peter Nightingale
Department of Computer Science
University of York, UK
https://orcid.org/0000-0002-5052-8634

András Z. Salamon
School of Computer Science
University of St Andrews, UK
https://orcid.org/0000-0002-1415-9712

Mateu Villaret
Department of CS, Mathematics and Statistics
University of Girona, Spain
https://orcid.org/0000-0002-8066-3458

***Index Terms*—Planning, Modelling, Constraint Programming**

***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 produce an automated tool for instance generation, parameterised on the grid, the combination of patterned blocks, and the steps required.**

I. Introduction

In *Puzznic* (Taito, 1989, Figure 1) the player manipulates blocks in a grid until they *match*: two or more blocks of the same pattern are adjacent orthogonally, and are removed from play. The goal is to remove all patterned blocks in the grid. 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 blocks are affected by gravity, leading to complex state changes and the possibility of cascaded matches. It is therefore a significant challenge to model, motivating our study. Puzznic is naturally characterised as an AI Planning problem [13]. Given a model of the environment (here the grid, blocks, and their behaviour), a planning problem is to find a valid sequence of actions (block moves) from an initial to a goal state of the environment (all blocks matched). Constraint Programming has been used to solve planning and scheduling problems [1]–[3] and has proven successful for Plotting [9], [10], a puzzle game which also has complex changes of state.

The player has full information about the game state and actions have deterministic effects. This is similar to other single-player games such as Plotting [9], [10], peg solitaire [15] and Black Hole [12]. Instances consist of a grid of cells, which

Fig. 1: Detail from Puzznic (Taito, 1989).

may be empty, a wall, or contain a patterned block. The player selects a single patterned block using the red cursor. The block can be *moved* horizontally if the next cell in that direction is empty. Patterned blocks are affected by gravity, and *fall* until coming to rest above another non-empty block or wall. If the player moves a block over an empty cell, they *immediately* lose control of the block as it falls. If two or more blocks with the same pattern are at rest (not falling) and adjacent orthogonally, they *match* and are removed.

Building upon [7], we present models of Puzznic in the constraints and planning paradigms. Our primary contributions are: (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 generation tool 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. Full models for this paper are available at https://github.com/blind-anon/CP-2024-Submission-Additional-Material.

II. Membership in NP

Some Puzznic levels allow moving wall blocks, which can carry patterned blocks. Herein we only consider a version of the game without moving wall blocks, which is also known as *Cubic*. The solvability of this static variant has been claimed to

be NP-hard [11]. Deciding solvability of the related but more complicated Hanano puzzle is PSPACE-complete [5].

Puzznic is naturally represented as a planning problem, the solvability of which is generally PSPACE-hard [4]. However, the decision problem for the Static variant of Puzznic Solvability is in the complexity class NP (a proof is provided in the workshop version of this paper [8]). We could therefore expect Puzznic to be amenable to constraint programming approaches tailored to solve problems in NP.

III. PDDL Model

Considering Puzznic as a planning problem requires finding a sequence of actions (a *plan*) whose application successively transforms a given initial state into a goal state. A set of finite-domain variables determines the state at each step. An action is applicable at a certain state if the state satisfies its preconditions. The state is then modified according to the effects of the action. The standard Planning Domain Definition Language (PDDL) [14] 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.

PDDL does not support matrices, hence we use a graph-based representation and 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 the different block patterns. Blocks are represented as patterns assigned to locations.

To define the state we use several predicates. The `patterned` predicate states whether a given location has a given pattern. The `next` predicate indicates if two locations are adjacent in a certain direction: we treat the `next` predicate as a declarative specification of the game grid adjacency graph. Since we can never move wall blocks, we exclude walls from the grid, leading to a smaller state space. With the `free` predicate we state whether a certain location is free. 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:

```
(:derived (free ?l) (; a free block has no pattern
 forall (?p - pattern) (not (patterned ?l ?p)) ))
(:derived (falling_flag) ( ; blocks need to fall?
 exists (?l1 ?l2 - location) (and
  (next ?l1 ?l2 down) (not (free ?l1)) (free ?l2)) ))
(:derived (matching_flag) ( ; blocks need to match?
 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 their preconditions and effects which usually constrain the parameters. Preconditions define the requirements a state must satisfy for the action to be applicable. Effects define how actions change the state. Three actions are defined: `move_block`, `fall_block` and `match_blocks`. The solving process must follow the semantics of the game. Therefore, if there are blocks remaining then we have to consider the flags to decide which kind of action is allowed. 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 the game semantics, we check these flags in the actions' preconditions.

For simplicity, we model the matching of an arbitrary set of blocks as an atomic operation:

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

Finally, as the goal is to remove all patterned blocks from the grid, we must 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, needs to fall
    (free ?l2))         ; l2 is free as we're falling on it
  :effect (and ; assign patterns: l1 pattern goes to l2
               (not (patterned ?l1 ?p)) (patterned ?l2 ?p)))
```

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`, where all the falling of a single block would be dealt with in one action. Surprisingly, preliminary experiments showed that compressing fall moves substantially degrades the performance of the planners we considered, in particular for tall instances (e.g. Giraffes and Eagles, see section VI). 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 planner branching factor. We therefore omit the compressed falls model from our experiments.

IV. A Constraint Model of Puzznic

Our constraint model is formulated in Essence Prime [17], [18], 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, allowing it to be applied instantaneously after 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)
$ Patterned block init positions, (row, col)
given initPatternedBlocks :
  matrix [int(1..noPatternedBlocks), int(1..2)] of int(1..)
letting PATTERNEDBLOCKS be domain int(1..noPatternedBlocks)
given patternBands :
  matrix [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. A perimeter of wall blocks is assumed. 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 [9], [10]) 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.

Following a common pattern in constraint models of AI planning problems [16], 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). Since much of the grid state (walls, empty cells) is fixed we maintain only the coordinates of each patterned block. Initial and goal states are stated simply on this viewpoint:

```
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)
$ 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,
```

The model allocates steps to one of three disjoint *modes*: Matching, Progressing and Row Compact.

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

A. *Matching Mode*

Matching mode is triggered by any pair of patterned blocks being adjacent horizontally or vertically at the previous time step. Auxiliary `matchingGrid` is used to detect this state. Matching mode at time step `t` is then forced according to the state of the `matchingGrid` at time step `t-1`, and the matching blocks are removed.

```
find matchingGrid :
 matrix [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]),
```

```
forAll step : STEPSFROM1 .
  (mode[step] = MATCHING_MODE) <->
  (sum(flatten(matchingGrid[step-1,..])) > 0),
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 stay 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 blocks stay in col, above blocks they were above
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 blocks stay above/below wall blocks
$ they were 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
forAll step : STEPSFROM1 .
```

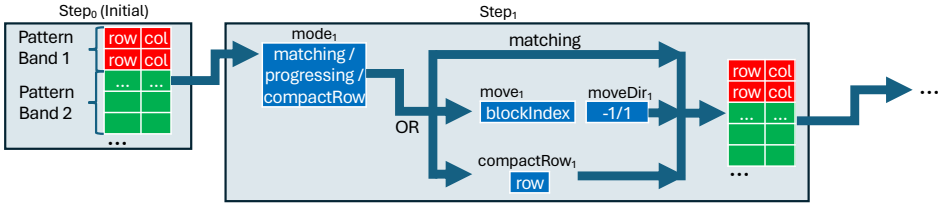


Fig. 2: Constraint model structure: interleaved state, mode variables for flow of control, and action variables, annotated with their domains. Auxiliary variables not depicted for clarity.

```
  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.

B. 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.

```
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 plus a dummy value 0 when in another mode. `moveDir` indicates a left or a right move. The following constraints specify a valid progressing move, transforming state at time `t-1` to time `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:

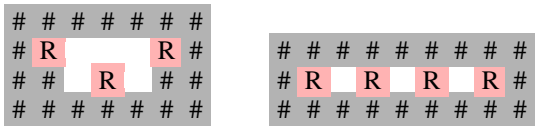


Fig. 3: Illustration of inconsistencies of parallel moves.

```
$ Frame axiom: blocks not in source col stay in 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 under selected stay put.
$ Simplified to all blocks whose row is less than 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 — see repository for full details.

C. 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 [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 trigger a match, and we must avoid initiating falls — see repository for details. Finally, we disallow row compact mode from initiating a match on the same row, and break symmetry 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 is row compact
forAll step : INTERIORSTEPS .
  (mode[step] = ROWCOMPACT_MODE) ->
  (mode[step+1] != MATCHING_MODE),
$ A move leading 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])),
```

D. 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 inactive. 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]),
mode[noSteps] = MATCHING_MODE,$ last step must be a matching
```

V. A Constraint Model for Instance Generation

We modify our constraint model to produce a tool 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)
$ At least a pair of blocks per pattern
given noPatternedBlocks : int(2*noPatterns..)
letting PATTERNEDBLOCKS be int (1..noPatternedBlocks)
given noSteps : int(1..)

find initGrid :
  matrix [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:

```
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.
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 1st 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 generation tool. These are found efficiently with our model encoded into SAT via SAVILE ROW. The largest takes just over 7 minutes to be found on a 2021 MacBook Pro.

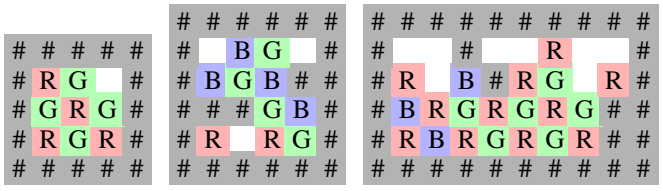


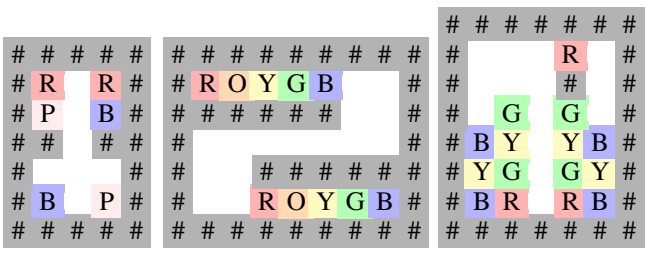
Fig. 4: Instances produced by our instance generation tool.

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

VI. EMPIRICAL EVALUATION

The plans produced by the two models are not directly comparable in terms of length. As discussed above, the Planning model is fine-grained whereas the constraint model implements gravity instantaneously and also permits 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. Table I presents the results of our evaluation across five families of Puzznic instances. Full details of our experimental setup are provided in [8].

We see that the planning and constraint-based approaches have complementary strengths, with neither dominant. Performance varies according to both the instance family and the instance size. *General* instances are from versions of the game and are intended for human players. 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 Table I). 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; more segments lead to more possible moves for a solver to consider. For the caterpillar instances we see the reverse: the planning approach performs well on small instances, but the constraint model scales significantly better.



(a) 5x7 (b) Snake (c) Caterpillar

Fig. 5: Sample instances from different families.

Family	Instance	FD	SAT	Family	Instance	FD	SAT	Family	Instance	FD	SAT
general	9x5-ps1-b22	**0.75**	4.05	snake	9x7-cubic-2	**1.01**	1.52	caterp.	colinsmall	**0.98**	2.49
general	10x5-ps1-b11	**0.87**	22.00	snake	73x7-cubic-2	23.35	**3.28**	caterp.	colinmirrorsm	**1.15**	5.93
general	5x7-ps1-a13	**0.95**	3.70	snake	9x7-cubic-4	**1.00**	6.43	caterp.	colin	236.54	**14.69**
general	10x7-bcl-014-2	**1.06**	6.95	snake	73x7-cubic-4	3571.98	**27.60**	caterp.	colinmirror	TO	**57.87**
general	9x7-ps1-b12	**1.10**	141.83	snake	9x7-cubic-6	**1.87**	37.85	snake	9x7-cubic-8	**3.73**	457.37
general	8x12-ps1-e22	**2.03**	1238.86	snake	28x7-cubic-6	TO	**113.41**	snake	11x7-cubic-8	1880.42	**665.32**
giraffe	5x24-test	**1.93**	6.19	snake	73x7-cubic-6	TO	**281.35**	snake	12x7-cubic-8	TO	**765.32**
giraffe	5x49-test	**6.07**	10.75	snake	9x7-cubic-10	**9.82**	TO	snake	18x7-cubic-8	TO	**852.54**
giraffe	5x99-test	23.66	**20.51**	eagle	5x50-test	**5.61**	5.95	snake	28x7-cubic-8	TO	**1279.29**
giraffe	5x200-test	100.42	**43.06**	eagle	5x100-test	21.57	**9.22**	snake	43x7-cubic-8	TO	**2001.48**

TABLE I: Results for 5 instance families. Times in seconds, 1-hour timeout indicated as TO. FD: Fast Downward, SAT: Kissat via SAVILE ROW. Instance names like _×_ show their dimensions. Snake instances: last part of name is # of patterned blocks.

Eagle and *Giraffe* instances are formed respectively by adding empty rows above, and interpolating empty rows in the middle of, an existing instance (here we have used fig. 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 fig. 5b (taken from [11]) 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 solutions (although plans will require more player moves), and also does not change the difficulty for humans. Our CP approach takes advantage of compact row moves to traverse long horizontal distances, 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 or more.

VII. CONCLUSION

In this work we have presented the static variant of *Puzznic* as a challenging new benchmark, and established its membership in NP. We have presented models for both AI planning and constraint programming, together with a constraint-based instance generation tool. Our empirical results demonstrate that these two approaches are complementary: primacy of one over the other depends on the sub-family of Puzznic instances 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.

ACKNOWLEDGEMENTS

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

REFERENCES

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

[2] R. Barták and D. Toropila, "Reformulating constraint models for classical planning," in *FLAIRS*. AAAI Press, 2008, pp. 525–530.

[3] M. Bofill, J. Coll, J. Suy, and M. Villaret, "Solving the multi-mode resource-constrained project scheduling problem with SMT," in *ICTAI*. IEEE, 2016, pp. 239–246. https://doi.org/10.1109/ICTAI.2016.0045

[4] T. Bylander, "The computational complexity of propositional STRIPS planning," *Artificial Intelligence*, vol. 69, no. 1, pp. 165–204, 1994.

[5] M. C. Chavrimootoo, "Defying gravity and gadget numerosity: The complexity of the Hanano puzzle," in *DCFS*, ser. LNCS, vol. 13918, 2023, pp. 36–50. https://doi.org/10.1007/978-3-031-34326-1_3

[6] J. Espasa, I. P. Gent, R. Hoffmann, C. Jefferson, A. M. Lynch, A. Salamon, and M. J. McIlree, "Using small MUSes to explain how to solve pen and paper puzzles," 2023. https://doi.org/10.48550/arXiv.2104.15040

[7] J. Espasa, I. P. Gent, I. Miguel, P. Nightingale, A. Z. Salamon, and M. Villaret, "Towards a Model of Puzznic," in *ModRef*, 2023. https://doi.org/10.48550/arXiv.2310.01503

[8] ——, "Cross-paradigm modelling: A case study of Puzznic," in *ModRef*, 2024. https://modref.github.io/papers/ModRef2024_CrossParadigmModelling.pdf

[9] J. Espasa, I. Miguel, P. Nightingale, A. Z. Salamon, and M. Villaret, "Plotting: a case study in lifted planning with constraints," *Constraints*, vol. 29, 2024. https://doi.org/10.1007/s10601-024-09370-x

[10] J. Espasa, I. Miguel, and M. Villaret, "Plotting: a planning problem with complex transitions," in *CP*, 2022, pp. 22:1–22:17. https://doi.org/10.4230/LIPIcs.CP.2022.22

[11] E. Friedman, "The game of Cubic is NP-complete," 34th Florida MAA Section Meeting, 2001. https://erich-friedman.github.io/papers/cubic.pdf

[12] I. P. Gent, C. Jefferson, T. Kelsey, I. Lynce, I. Miguel, P. Nightingale, B. M. Smith, and S. A. Tarim, "Search in the patience game 'black hole'," *AI Communications*, vol. 20, no. 3, pp. 211–226, 2007.

[13] M. Ghallab, D. Nau, and P. Traverso, *Automated Planning and Acting*. Cambridge University Press, 2016.

[14] P. Haslum, N. Lipovetzky, D. Magazzeni, and C. Muise, *An Introduction to the Planning Domain Definition Language*. Springer, 2019.

[15] C. Jefferson, A. Miguel, I. Miguel, and A. Tarim, "Modelling and solving English Peg Solitaire," *Comput. Oper. Res.*, vol. 33, no. 10, pp. 2935–2959, 2006. https://doi.org/10.1016/j.cor.2005.01.018

[16] H. Kautz and B. Selman, "Planning as Satisfiability," in *Proceedings of ECAI*, 1992, pp. 359–363.

[17] P. Nightingale, "Savile Row manual," 2021. https://doi.org/10.48550/arXiv.2201.03472

[18] P. Nightingale, Ö. Akgün, I. P. Gent, C. Jefferson, I. Miguel, and P. Spracklen, "Automatically improving constraint models in Savile Row," *Artificial Intelligence*, vol. 251, pp. 35–61, 2017.

[19] H. Simonis, "Sudoku as a constraint problem," in *Workshop on Modeling and Reformulating Constraint Satisfaction Problems*, 2005.