INFORMS Transactions on Education

Publication details, including instructions for authors and subscription information:
http://pubsonline.informs.org

Puzzle—The Fillomino Puzzle

Robin H. Pearce, Michael A. Forbes

To cite this article:
Robin H. Pearce, Michael A. Forbes (2017) Puzzle—The Fillomino Puzzle. INFORMS Transactions on Education 17(2):85-89.
https://doi.org/10.1287/ited.2016.0166

informs
http://pubsonline.informs.org/journal/ited/

INFORMS TRANSACTIONS ON EDUCATION
Vol. 17, No. 2, January 2017, pp. 85–89
ISSN 1532-0545 (online)

Puzzle

The Fillomino Puzzle

Robin H. Pearce,[a] **Michael A. Forbes** [a]

[a] School of Mathematics and Physics, University of Queensland, Brisbane, St. Lucia, Austrilia QLD 4072
Contact: r.pearce2@uq.edu.au (RHP), m.forbes@uq.edu.au (MAF)

Received: June 2, 2016
Accepted: July 14, 2016
Published Online: January 13, 2017

https://doi.org/10.1287/ited.2016.0166

Abstract. Logic puzzles form an excellent set of problems for the teaching of advanced solution techniques in operations research. They are an opportunity for students to test their modelling skills on a different style of problem, and some puzzles even require advanced techniques to become tractable. Fillomino is a puzzle in which the player must enter integers into a grid to satisfy certain rules. This puzzle is a good exercise in using lazy constraints and composite variables to solve difficult problems.

Supplemental Material: Supplemental material is available at https://doi.org/10.1287/ited.2016.0166.

Keywords: puzzle • lazy constraints • composite variables

Introduction

Logic puzzles form an excellent set of problems for the teaching of advanced solution techniques in operations research. They are an opportunity for students to test their modelling skills on a different style of problem, and some puzzles even require advanced techniques to become tractable. Puzzles are also typically modelled as integer programs (IP), for example, crossword construction (Wilson 1989), Su Doku and the Log Pile puzzle (Chlond 2005), Rummikub (den Hertog and Hulshof 2006), the Battleship problem (Meuffels and den Hertog 2010) and more.

The use of composite variables can make the solution to some problems much easier to obtain, however they are not widely used. The same can be said for lazy constraints: they are an extremely powerful technique and can yield impressive results for difficult integer and mixed-integer programs, however there are few publications demonstrating the use of lazy constraints. This is perhaps because they are not widely known techniques, and as such should be taught more often in advanced undergraduate operations research courses.

Fillomino is a puzzle whose creation is credited to Nikoli Co., Ltd. (2008). The player must enter integers into a grid to satisfy certain rules. Some cells in the grid have preset values which cannot change. If two cells that share an edge have the same number, they become a tile. If a cell is neighbouring a tile of the same value, it joins the tile. The grid must be filled with numbers such that every cell is assigned a number, and every tile filled with k's has k cells belonging to it. Two tiles of the same number cannot share an edge, since they would merge into one tile which has too many cells. One last assumption that we make is every tile must contain at least one preset value, however there are versions of the puzzle where this is not the case.

The solution to each puzzle is a unique layout of polyominoes "(shapes made by combining individual squares)." An example of a starting grid and its unique solution can be seen in Figure 1. Every puzzle can be solved logically, and an efficient algorithm exists for solving it in this way (Yen et al. 2011). Constraint programming could also be applied to this problem, however we are interested in solving it using integer programming, since this puzzle is an excellent example of the usefulness of composite variables and lazy constraints. First, we will give a brief description of these two methods.

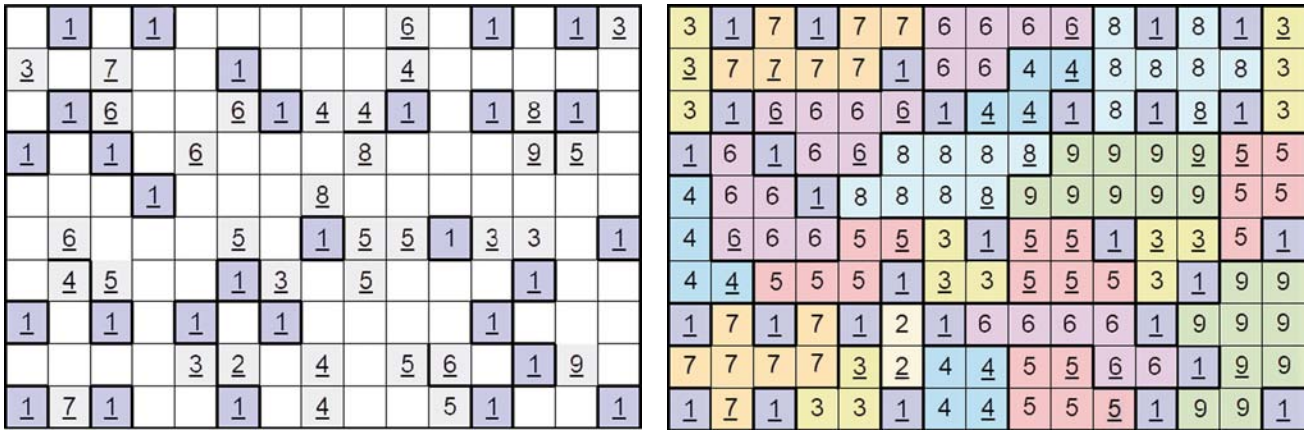
Lazy Constraints

There are a number of problems which can be formulated using very many constraints, most of which may not be required (or active) for solving a specific instance of the problem. Specialist branch-and-cut procedures have been used to solve some of these problems (Applegate et al. 2001). However, since 2012, commercial MIP solvers have allowed these constraints to be added as required during the exploration of the branch-and-bound tree. The advent of such "lazy constraints" has made the solution of these problems much easier in some cases.

The general approach to formulating a problem using lazy constraints is to remove a complicating part of the problem. Whenever the solver finds a feasible integer solution, that solution is checked against the constraints that were removed. If the solution is actually infeasible, a new constraint is added to the problem to cut off that (and possibly other) solutions.

Another common use for lazy constraints is to implement subtour elimination constraints in problems

Figure 1. Example of Fillomino Starting Grid and Corresponding Solution



Notes. Underlined numbers are preset values.

like the Travelling Salesman Problem (Gurobi Optimization Inc. 2015). Instead of adding all possible subtour elimination constraints, the problem is formulated only with the degree-2 constraint (every city must be connected to exactly two other cities). Every time a candidate solution is found, if a subtour exists, a constraint eliminating that specific subtour is added and the branch-and-bound process continues. This can have a large impact on the solution time for difficult problems.

The main reason we use lazy constraints for this problem is to enforce the size of the tiles. It is very difficult to formulate an integer program which manages to generate tiles of the correct size, however with the inclusion of lazy constraints the problem becomes much more simple.

Composite Variables

We use the term composite variables to refer to variables which represent a collection of decisions in the underlying model. In this sense, using composite variables is a column generation technique. While delayed column generation is a common form of column generation (Barnhart et al. 1998), the method we present here involves generating all possible columns initially. Modern solvers are able to solve even very large composite variable formulations extremely quickly, and so the majority of time is usually spent generating the columns. For this problem, the solver is able to solve the problem in the preprocessing stage, and the generation of all possible tiles takes only a few seconds, making composite variables an efficient way of solving this problem.

Integer Programming Formulation

We can formulate this puzzle as an integer program and apply lazy constraints to it to find the unique solution to each grid. We require variables representing the entries in each cell of the grid. We also use variables recording how many tiles of each type there are, which we can use to tighten the formulation by placing upper and lower bounds on the number of cells of each type. Since there is a unique solution, we do not require an objective function. The formulation we present is as follows:

Sets

N, M: the rows and columns of the grid;
K: the range of valid cell entries;
$Neigh_{ij}$: the set of cells which share an edge with (i, j).

Data

$Preset_{ij}$: the given value of cell (i, j). 0 implies cell (i, j) is empty.

Variables

x_{ijk}: is 1 if cell (i, j) is of type k, 0 otherwise;
y_k: is the number of tiles of type k.

Constraints

$$\sum_{k \in K} x_{ijk} = 1, \quad \forall\, i, j \in N \times M, \tag{1}$$

$$x_{ijPreset_{ij}} = 1, \quad \forall\, i, j \in N \times M \mid Preset_{ij} \neq 0, \tag{2}$$

$$x_{ij1} = 0, \quad \forall\, i, j \in N \times M \mid Preset_{ij} \neq 1, \tag{3}$$

$$x_{ijk} \leq \sum_{(a,b) \in Neigh_{ij}} x_{abk}, \quad \forall\, i, j \in N \times M, \quad \forall\, k \in K \mid k > 1, \tag{4}$$

$$\sum_{(a,b) \in Neigh_{ij}} x_{ab2} \leq 1 + (|Neigh_{ij}| - 1)(1 - x_{ij2}), \quad \forall\, i, j \in N \times M, \tag{5}$$

$$\sum_{(i,j) \in N \times M} x_{ijk} = k y_k, \quad \forall\, k \in K. \tag{6}$$

Constraint (1) ensures every cell has exactly one value assigned to it. The next two constraints (2–3) fix the preset values, and make sure no extra 1's are added. Constraint (4) says that a cell can only have a value k if

at least one neighbouring cell also has a value k. Since there is only one unique domino (a tile with two cells), we can add a constraint for the case of 2's that says that if a cell is a 2, it has exactly one neighbour which is of type 2, however if it is not a 2 then there are no restrictions on how many neighbouring 2's there are. This is enforced by constraint (5). Finally, we know that the number of cells of type k is k times the number of tiles of type k (6).

This is an incomplete formulation. These are the base constraints to generate example solutions to the grid, however there is nothing to prevent tiles of sizes other than k from occurring. We find it near impossible to add constraints that enforce the correct size of each tile and that avoid two tiles of the same type touching or overlapping.

Use of Lazy Constraints to Enforce Tile Size

When a potential solution is found by the above implementation, we must check to make sure all tiles are the correct size. To do this, we measure the size of each tile, and if it is not correct, we add one of two lazy constraints.

Bounding Tile Size from Above

Once we have a potential solution, we check the size of each tile. If any tiles are too big, then at least one of the cells in this tile must change its value. Let T be the set of cells in this tile, which are all numbered k^* with $|T| > k^*$. We then add the lazy constraint:

$$\sum_{(i,j)\in T} x_{ijk} \leq |T| - 1. \tag{7}$$

We cannot say that the number must be equal to k^*, since this one tile may in fact be two tiles which are joined by one incorrectly numbered cell. Enforcing an equality constraint would make the unique solution invalid and the model would become infeasible. This constraint forbids the current configuration of tiles, which forces the model to try something different. Eventually there will be no more tiles which are larger than they are meant to be. This does not, however, stop them from being smaller than they need to be.

Bounding Tile Size from Below

If there are no tiles that are larger than they are allowed to be in a solution, we then check for tiles that are smaller than required. For each such tile, let T be the set of cells in this tile, which are all numbered k^* and $|T| < k^*$. Also let TN be the set of cells which are a neighbour of at least one cell in T, but are not in T, and whose preset value is 0. If their preset value was k^*, they would already be part of this tile. We then add the lazy constraint:

$$\sum_{(i,j)\in T} x_{ijk^*} + \sum_{(a,b)\in TN} \sum_{\substack{k'\in K \\ k'\neq k^*}} x_{abk'} \leq |T| + |TN| - 1. \tag{8}$$

This constraint says that either the tile needs to get smaller and perhaps disappear, or some of the neighbours have to change their value to k^*. In other words, it will either remove the tile or pull at least one neighbour into it. This will eventually bound all tiles from below. These two lazy constraints, together with the integer programming formulation above, will find the unique solution to each puzzle, however it may take a long time for the larger grids. We can speed it up using a number of preprocessing techniques.

Preprocessing Techniques

Upper Bound on Number of Tiles

Since every tile has to cover at least one preset value, we can calculate an upper bound on the number of tiles of each type. The simplest way would be to count the number of preset cells of type k, which provides an upper bound for y_k. There is, however, the possibility that there may be multiple preset cells of the same type connected to each other, which will be part of the same tile. Thus, for every value $k \in K$, we count the number of connected groups of cells of type k and call this number $\mathscr{C}_k$. If $k < 3$, then the number of tiles is exactly equal to $\mathscr{C}_k$, otherwise we add constraints of the form:

$$y_k \leq \mathscr{C}_k, \quad \forall\, k \in K,\, k > 2. \tag{9}$$

Lower Bound on Number of Tiles

We can also work out a lower bound on the number of tiles of each type since every preset value has to be covered. Because the upper bound on the number of tiles is an equality for types 1 and 2, we only need to consider every value $k \in K$ greater than 2. For each of these, we can calculate the geodesic distance of each cell from the nearest preset value of k. The geodesic distance is the length of the shortest valid path of cells from a preset value to the current cell.

For each value $k > 2$, we calculate the geodesic distance from each empty cell to the nearest cell with preset value k. This distance represents the minimum number of cells of type k that must be added to include each cell in a tile of type k. If it is possible for two cells with preset values k to be part of the same tile, there will be two paths, one from each, of length at most $\lfloor (k-1)/2 \rfloor$ which will touch. By removing any cells whose geodesic distance is greater than $\lfloor (k-1)/2 \rfloor$, we can count the number of connected components remaining, which gives us the minimum number of tiles needed to cover all cells with preset values of k. The constraint is of the same form as above, except it will be a greater than or equal to constraint.

Restriction on Location of Cells

Because a cell can only have a value k if it is part of a tile of type k, and each tile must cover at least one preset value, if a cell has a geodesic distance of k or greater from the nearest preset cell of type k, it can not possibly be part of a tile of type k. We can add constraints to reflect this by following a similar procedure to the other preprocessing techniques. For each value of $k \in K$ greater than 2, we calculate the geodesic distance from each preset value. Where n preset values of the same type are connected, we know that all cells with a geodesic distance of at most $k-n$ could possibly take the value k. For all cells (i, j) which are not within $k-n$ of any preset value, we add a constraint $x_{ijk} = 0$, which removes many unnecessary variables.

Neighbours Around Preset Values

Finally, we know that each preset value belongs to a tile, so if we consider the set of all cells with geodesic distance at most $k-1$ from a particular preset cell of type k, there must be at least k cells of type k in this set. For a connected group of l preset cells of type k, we find the set S of cells with geodesic distance at most $k-l$ from any cell of the connected group, and add the constraint

$$\sum_{(i,j)\in S} x_{ijk} \geq k. \tag{10}$$

Composite Variables Formulation

Another way of formulating this problem is to consider it as a tiling problem. If we can find every possible placement of tiles of every type, we can choose which combination of tiles gives us the unique solution. The method of composite variables is a column generation approach, but instead of generating columns during the solution process, all columns (in this case, valid placements of tiles) are generated first. We now present our composite variables formulation.

Sets

N, M: the rows and columns of the grid,
K: the range of valid cell entries,
P: the set of all possible tiles that can be placed in the grid,
$P_k \subseteq P$: the set of all possible tiles of type k that can be placed in the grid,
$Neigh_{ij}$: the set of cells which share an edge with (i, j),
T_{ij}: the set of tuples (k, p) representing all tiles $p \in P$ which cover cell (i, j).

Data

$Preset_{ij}$: the given value of cell (i, j). 0 implies cell (i, j) is empty.

Variables

x_{kp}: is 1 if tile $p \in P_k$ is used, 0 otherwise.

Constraints

$$\sum_{(k,p)\in T_{ij}} x_{kp} = 1, \quad \forall (i, j) \in N \times M. \tag{11}$$

There is one constraint for every cell of the grid which says that it must be covered by exactly one tile, with the exception of cells that have a 1 in them. Each tile is represented as a (k, p) pair, describing which value of k it covers and which $p \in P_k$ it is. If all possible tile placements are known, this will yield the unique solution to the problem. To find all possible tile placements, we start with the preset cells of each type and grow them outwards by adding neighbours one by one, until they are the correct size, being careful to remove duplicates as we go.

The runtime of this implementation is highly dependent on how quickly one can find all possible tile placements, as the integer program itself solves in a fraction of a second. Another possible concern is that two tiles of the same type may touch in a solution. If this is the case, we can use a modified version of the lazy constraints described above. By following the same procedure, checking each tile to see if any one is larger than k cells, we look at all the cells in this oversized tile and note which tiles $p \in P$ they belong to in the current solution. Where T is the set of tiles in this violation, we add the constraint

$$\sum_{(k,p)\in T} x_{kp} \leq |T| - 1. \tag{12}$$

This will ensure we find the unique solution.

Results

We tested both implementations using Python 2.7 and the Gurobi 6.5 (2016) solver package. We sourced four randomly-generated puzzles from the Puzzle Baron website (2016), all of which are 20×20 and follow our assumptions. Table 1 shows that, when implemented efficiently, the composite variables implementation is significantly faster. The majority of time is spent generating the tiles, and once completed, the IP solves in a fraction of a second. For the lazy constraints implementation, less than one second is spent preprocessing the grid and adding initial constraints.

Table 2 shows the number of variables and constraints for both implementations, as well as the number of nodes explored and lazy cuts generated for the lazy constraints implementation. For the composite variables implementation, the number of variables reflects the number of potential tiles which can be legally placed in the grid, and the number of constraints is 400 minus the number of squares that contain 1's. This is because there are no options for placing 1's outside the preset locations, so no tiles will

Table 1. Comparison of Runtimes for the Composite Variables and Lazy Constraints Implementations

	Composite variables		Lazy constraints	
Instance	IP	Total	IP	Total
1	0.09	2.28	25.38	25.73
2	0.11	2.41	11.42	11.76
3	0.13	1.64	8.37	8.73
4	0.13	2.08	4.05	4.41

Notes. All times are in seconds. The times spent solving the IP and the whole problem are reported separately.

Table 2. Comparison Between the Composite Variables and Lazy Constraints Implementations

	Composite variables		Lazy constraints			
Instance	Variables	Constraints	Variables	Constraints	Lazy cuts	Nodes
1	5,219	325	755	862	244, 535	48,932
2	6,979	324	536	618	136, 393	26,269
3	4,348	326	612	676	153, 322	28,428
4	4,842	329	648	702	43, 341	10,301

Notes. The number of variables and constraints used in solving the IP, number of lazy constraints added and number of nodes explored in the branch-and-bound tree are shown. The Lazy Cuts column is separated into (upper, lower) cuts.

cover those squares and as such they do not require constraints.

For the lazy constraints implementation, the number of variables reported is the number left after Gurobi's presolve stage. Initially, the number is always 3,609: $20 \times 20 \times 9$ for the x_{ijk} variables, and 9 for the y_k variables. With our preprocessing, most of those variables will be fixed to 0 (or 1 in the case of cells whose values have been preset), so the number remaining reflects the actual number of possibilities for cell values. Since we know that there are around 325 cells which are not 1, and there are usually 600 variables remaining, this suggests that on average a blank cell only has two choices for which number it could be.

The number of lazy constraints added is also interesting. The number of times tiles have to be bounded from below is always higher than, and usually at least double, the number of times they are bounded from above. This may be because both constraints can be satisfied by moving a cell of the tile to a neighbouring blank cell, thus maintaining the same size of the tile in a different location, however it is much easier for this to occur with tiles that are smaller than needed compared to those which are larger.

Conclusion

This problem is an excellent demonstration of how lazy constraints and composite variables can be used to solve problems which are difficult to implement as straight MIPs, or where the naive implementation is intractable. There are many other puzzles and industrial problems which can benefit from similar approaches.

References

Applegate D, Bixby R, Chvátal V, Cook W (2001) *TSP Cuts Which Do Not Conform to the Template Paradigm* (Springer, Berlin), 261–303.

Barnhart C, Johnson E, Nemhauser G, Savelsbergh M (1998) Branch-and-price: Column generation for solving huge integer programs. *Oper. Res.* 46(3):316–329.

Chlond M (2005) Classroom exercises in ip modeling: Su doku and the log pile. *INFORMS Trans. Ed.* 5(2):77–79.

den Hertog D, Hulshof P (2006) Solving rummikub problems by integer linear programming. *Comput. J.* 49(6):665–669.

Gurobi Optimization Inc. (2015) The travelling salesman problem with integer programming and Gurobi. http://examples.gurobi.com/traveling-salesman-problem/.

Gurobi Optimization Inc. (2016) Gurobi optimizer reference manual. http://www.gurobi.com.

Meuffels W, den Hertog D (2010) Solving the battleship puzzle as an integer programming problem. *INFORMS Trans. Ed.* 10(3): 156–162.

Nikoli Co Ltd. (2008) Database of Japanese puzzles.

Puzzle Baron (2016) Fillomino by Puzzle Baron. Accessed April 2016, http://fillomino.puzzlebaron.com.

Wilson J (1989) Crossword compilation using integer programming. *Comput. J.* 32:273–275.

Yen SJ, Su TC, Hsu SC (2011) An efficient algorithm for solving fillomino. *2011 IEEE Internat. Conf. Fuzzy Systems (FUZZ)* (IEEE, Piscataway, NJ), 190–194.