Modeling Irregular Shape Placement Problems with Regular Constraints

Mikael Z. Lagerkvist[1] and Gilles Pesant[2]

[1] ECS, ICT, KTH - Royal Institute of Technology, Sweden, zayenz@kth.se
[2] École Polytechnique de Montréal, Montreal, Canada, Gilles.Pesant@polymtl.ca

Abstract. The `regular` constraint [5] is a powerful global constraint with many possible uses. It was introduced to model certain common constraints in rostering problems. In this paper we show how to use it to model placement problems with irregular shapes. The technique is illustrated by solving Pentominoes and Solitaire Battleships. The efficiency of the basic model is improved by projecting out irrelevant information. Experimental results on Pentomino packing benchmark instances indicate that this simple approach can be competitive with specialized algorithms. From the applications we can identify some areas for improvement in the implementation of regular constraints.

1 Introduction

Constraint programming is a very powerful technique for solving combinatorial problems. A key factor in harnessing this power is creating a model of the problem that is both clear and efficient. Creating such models is a hard process, and often the goals of clarity and efficiency are in conflict. In this paper, we show how the `regular` constraint [5] can be used to model placement problems with potentially very irregular shapes.

Placement problems have been studied extensively, both in general and in the context of constraint programming. The Pentominoes problem was among the first solved using backtracking algorithms [6, 4]. In the context of constraint programming, several specialized constraints have been proposed. The well-known `diffn` constraint [3] specifies the placement of n-dimensional boxes. To generate connected shapes of a given fixed size, the `polyomino` constraint [2] can be used. For placing specific irregular shapes, a generalization of `diffn` called `geost` has been proposed recently [1].

Contributions. The main contribution of this paper is the natural way irregular shape placement problems can be modeled using regular expressions. The paper also contributes two natural and clear models for problems that are hard to model without special purpose constraints.

Plan of paper. The next section gives background on regular expressions and the `regular` constraint. Section 3 describes how shape placement can be encoded using regular expressions. The following two sections describe solving the Pentominoes and Solitaire Battleships problems. Section 6 concludes the paper.

2 Background

This section introduces the notions and concepts needed for this paper.

Regular languages. A *regular expression* (RE) is an expression $R := R\,R \mid R^* \mid R|R \mid (R) \mid \epsilon \mid X$, representing concatenation, repetition, disjunction, grouping, the empty language, and simple symbols respectively. The expression R^n is a short-hand for the expression R concatenated n times.

A *deterministic finite automaton* (DFA) is a tuple $\langle S, Val, T, q, A\rangle$, where S is a set of states, $T : S \times Val \to S$ is a transition function, $q \in S$ is the start state, and $A \subseteq S$ is a set of accepting states. A DFA matches a string $x_1, \ldots, x_n$ iff there is a sequence of states $q_1, \ldots, q_{n+1}$ such that $q_1 = q$, $q_{i+1} = T(q_i, x_i)$ for $i = 1, \ldots, n$, and $q_{n+1} \in A$.

The Regular constraint. Regular languages can be used as a specification language for the `regular` constraint introduced by Pesant [5]. Specifying a regular language can be done using both REs and DFAs, the choice depends on the particular constraint that is to be specified. If an RE is used, then it is translated into a DFA for propagation. For efficiency, the DFA can be minimized.

The propagation algorithm for `regular` uses a data structure called a *layered graph* (LG). The LG for a `regular` constraint over variables $x_1, \ldots, x_n$ is constructed by unfolding the DFA $\langle S, Val, T, q, A\rangle$ into a graph $\left\langle \cup_{i=1}^{n+1} N^i, E\right\rangle$, where layer N^i contains states $q_1^i, \ldots, q_m^i$, and edges are between consecutive layers following the DFA transitions, $E = \{\left\langle q_a^i, v, q_b^{i+1}\right\rangle \mid q_b \in T(q_a, v), v \in Val\}$. Propagation proceeds by finding the union of paths from q^1 to any q_f^{n+1} where $q_f \in A$. The set of paths has an edge between layers i and $i+1$ if and only if the edge value is valid for x_i.

3 Shape Placement Encoded as a Regular Expression

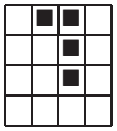
Consider placing the shape in Figure 1(a) in the 4 by 4 grid shown in Figure 1(b). In order to encode its placement as a string, we cut the grid into horizontal strips corresponding to its rows and concatenate them (we could equivalently have made vertical strips). Each square of the resulting sequence ABC... P takes value 1 if the shape overlaps it and value 0 otherwise. For example, placing the shape on squares B, C, G, and K (Figure 1(c)) results in the string 0110001000100000. This string and all other strings corresponding to placing that shape on the grid belong to the language described by regular expression $0^*110^310^310^*$: first comes some number of 0's, then two 1's in a row (covering squares B and C in our example placement), then come exactly three 0's (not covering places D, E, and F), and so on. The variable number of 0's at the beginning and at the end makes the expression match any placement of the shape in the grid.

There is one problem however, since it allows "placing" the shape on squares D, E, I, and M (Figure 1(d)). That is, the shape may wrap around the grid. To prohibit this, we add a new dummy-column to the grid, so that it looks

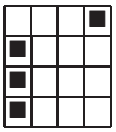
(a) Shape to place

A	B	C	D
E	F	G	H
I	J	K	L
M	N	O	P

(b) Grid to place in



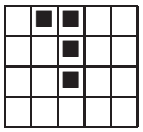
(c) Possible placement of shape.



(d) Erroneous placement of shape.

A	B	C	D	Q
E	F	G	H	R
I	J	K	L	S
M	N	O	P	T

(e) Modified grid.



(f) Shape positioned on modified grid

Fig. 1: Placing a shape in a grid.

like the grid in Figure 1(e). The new squares Q to T are fixed to zero, and the regular expression for the placement of the shape is modified to include the extra column, resulting in $0^*110^410^410^*$ matching, for example, the placement in Figure 1(f). The language of that regular expression is now precisely the strings corresponding to possible placements of the given shape. From now on, we will assume that grids have an extra column on the right.

Shape placement using regular expressions is not limited to connected shapes. The only issue is that the extra column must be given a unique value that is not allowed to occur in inter-piece spaces, so that wrap-around is still prohibited.

The complexity of using the expression as a constraint is decided by the size of the layered graph. Consider placing a $l \times l$ piece on a grid of size $m \times n$, where $n < m$. The size of the LG is in $O(mn^2l)$ since the size of the regular expression grows as $O(nl)$.

4 Pentominoes

Pentominoes are the shapes constructed from five equal-sized squares where the squares form a connected component. This leads to twelve possible shapes. The pieces can be rotated and mirrored to obtain symmetrical variants.

There is a long history of using the pentomino pieces as a puzzle. For example, the pieces can be fitted together to form rectangles of sizes 6×10, 5×12, 4×15, and 3×20, as well as 8×8 with the middle 2×2 squares empty. Other sizes of pieces can also be used, and are then called *polyominoes*. In the following, we will consider placing a general set of polyominoes on a board of fixed size. The pieces to be placed can be rotated and mirrored.

4.1 Basic Model

We associate a distinct variable to each square on the board, whose value is the number of the piece which is placed over it. For each piece, we define one `regular` constraint for placing it on the board.

Fig. 2: Rotations of the piece in Figure 1.

Rotating pieces. In contrast to placing a shape as in the previous section, a pentomino piece may be rotated and mirrored. To handle that, we can use disjunctions of regular expressions. Consider the symmetric versions of the piece from Figure 1(a) as shown in Figure 2. The regular expression for placing the second piece in Figure 2 on a 4×4 grid is $0^*10^2110^*$. To combine the regular expressions for the first two pieces, we can simply use disjunction of regular expressions, arriving at the expression $0^*110^410^410^*|0^*10^2110^*$. There are 8 symmetries for the pieces in general. The 8 disjuncts for a particular piece might contain less than 8 distinct expressions. This redundancy is removed when the automaton for the expression is computed, since it is minimized.

Using disjunctions is naturally not limited to placing symmetrical pieces, it can also be used for placing alternative shapes of a piece or alternative pieces.

Placing several pieces. To generalize the above model to several pieces, we let the variables range from 0 to n, where n is the number of pieces to place. Given that we place three pieces, and that the above shown piece is number one, we will replace each 0-expression with the expression $\neg 1$, indicating all values other than 1. Thus, the original regular expression becomes $(\neg 1)^*11(\neg 1)^41(\neg 1)^41(\neg 1)^*$.

4.2 Improvements to the model

While the above model is clear and follows the original statement closely, it is not very efficient. The main problem here is that each constraint talks about every value in every variable (through the use of complemented values), even though it constrains only one of the values. To improve the performance, we must abstract away the dependence on the placement of the other values, and only focus on a specific value v for each constraint. Assume that the variables x describe a $m \times n$ board with k pieces to place. In order to abstract the values, we define additional 0-1 variables y_{ij}:

$$y_{ij} = 1 \Leftrightarrow x_i = j \quad 0 \leq i < mn,\ 1 \leq j \leq k$$

To connect the x and y variables, a channeling propagator can be used for each x_i and corresponding $\langle y_{i1}, \ldots, y_{ik} \rangle$. The `regular` constraints can now be defined using 0-1 expression on the y variables instead. This extended model has many more propagators than the original model, $mn + k$ compared to just k, but the extra propagators are very cheap. We gain in efficiency by reducing the number of times the `regular` constraints have to propagate, as well as by reducing the size of the layered graphs used.

Size	1st, `regular` failures	time	1st, `geost` failures	time	all, `regular` failures	time	all, `geost` failures	time
20×3	25 129	6 695	1 434	1 840	35 680	9 320	47 381	49 740
15×4	4 700	1 015	290	560	649 068	147 210	888 060	939 060
12×5	541	126	1 594	1 850	2 478 035	576 270	3 994 455	4 112 870
10×6	893	237	111	260	5 998 165	1 517 150	9 688 985	10 726 810

Table 1: Packing pentominoes. Time is given in milliseconds.

4.3 Evaluation

To evaluate the model four classic pentomino instances are tested: packing squares of sizes 20×3, 15×4, 12×5, and 10×6 with the twelve distinct pentominoes. The models are solved both for finding one and all solutions. The branching strategy is to instantiate the x variables in row-major order, trying to place "harder" pieces first. The ordering of the pieces is LFTWYIZNPUXV using the standard letter names for Pentomino pieces. The experiments were run using Gecode 2.1.1 as the CP system on an Athlon 64 3500+ with 2GB of RAM. The results are compared against the results from [1], were the `geost` constraint is presented. These experiments were run using SICStus Prolog 4 on a 3GHz Pentium IV with 1MB of cache.

The results are shown in Table 1. While no direct comparison can be made since different computers, constraint solvers, and branching schemes are used, the results show that using the `regular` encoding for placement is clearly competitive with the specialized constraint, at least on tightly constrained instances.

5 Solitaire Battleships

Solitaire Battleships (problem number 14 in CSPLib) is a puzzle derived from the two-person game Battleship. The objective of the puzzle is to figure out where the ships are located. The hints given for an instance are the number of occupied squares in each column and row. Also, some positions are revealed as being water, submarine, ship end (with orientation) or ship middle. Ships may not be placed adjacent to each other (not even diagonally). The standard size of the puzzle uses 10 by 10 grids, with 4 submarines (one unit long), 3 destroyers (2 units long), 2 cruisers (3 units long), and 1 battleship (4 units long). The size of the puzzle may, naturally, be varied in general.

Smith discusses several models for Solitaire Battleships, involving up to three types of variables to channel between, sum constraints, reified constraints, global cardinality constraints, and regular constraints [7]. The latter are used to restrict the configurations of ship segments on a given row or column. Global cardinality constraints ensure that the total number of segments of each type of ship is correct. We can replace them by ten placement constraints (one per ship), each encoded as a `regular` constraint.

Here the difficulty is that ships must be separated by water. By enlarging each ship with water on two adjacent sides (and similarly for the grid), the resulting

shapes simply need to avoid any overlap, a familiar situation. Rotations are again allowed, given that the sides where water is added to the ship is invariant. For the placement constraints there is no need to encode ship parts differently according to their location (end/middle) or orientation (horizontal/vertical) as long as water is added on the appropriate sides.

6 Conclusions and Future Work

We have shown how to use the `regular` constraint for writing clear, simple, and efficient models for small placement problems. These models use basic understanding on how to model strings of symbols as `regular` constraints, which many programmers have. The model obtained for Pentominoes is competitive with other approaches.

It would be interesting to do a more detailed comparison to the `geost` constraint for placement problems with regards to complexity and pruning efficiency. This includes the generalization of the proposed model to higher dimensional problems. The simplification of the model to use 0-1 variables for the constraints is important, and thus it would be interesting to find a way to do this transformation automatically.

Acknowledgements.. Mikael Lagerkvist is partially funded by the Swedish Research Council (VR) under grant 621-2004-4953. Gilles Pesant is partially funded by the Canadian Natural Sciences and Engineering Research Council (NSERC) under grant OGP0218028. The authors would like to thank Christian Schulte and Guido Tack for interesting discussions about the usage of the `regular` constraint.

References

1. N. Beldiceanu, M. Carlsson, E. Poder, R. Sadek, and C. Truchet. A generic geometrical constraint kernel in space and time for handling polymorphic k-dimensional objects. In C. Bessiere, editor, *CP*, volume 4741 of *Lecture Notes in Computer Science*, pages 180–194. Springer, 2007.
2. N. Beldiceanu, M. Carlsson, and J.-X. Rampon. Global constraint catalog. `www.emn.fr/x-info/sdemasse/gccat`, 2008.
3. N. Beldiceanu and E. Contjean. Introducing global constraints in CHIP. *Mathematical and Computer Modelling*, 12:97–123, 1994.
4. J. G. Fletcher. A program to solve the pentomino problem by the recursive use of macros. *Commun. ACM*, 8(10):621–623, 1965.
5. G. Pesant. A regular language membership constraint for finite sequences of variables. In M. Wallace, editor, *CP*, volume 3258 of *Lecture Notes in Computer Science*, pages 482–495. Springer, 2004.
6. D. S. Scott. Programming a combinatorial puzzle. Technical Report Technical Report No. 1, Department of Electrical Engineering, Princeton University, 1958.
7. B. M. Smith. Constraint programming models for solitaire battleships. Technical report, CPPod-20-2006, November 2006.