A Constraint-based Approach to Enigma 1225

Hadrien Cambazard and Barry O'Sullivan

Cork Constraint Computation Centre
Department of Computer Science, University College Cork, Ireland

Barbara M. Smith

School of Computing, University of Leeds, U.K.

Abstract

This paper presents a constraint programming approach to the Enigma 1225, a mathematical puzzle published in the New Scientist magazine in February 2003. An approach based on Prolog was published recently. In this paper we give a constraint programming perspective on the problem, highlighting the differences between the two methodologies. We show how problem-specific knowledge can be easily incorporated into a constraint-based approach, giving an efficient constraint model for the generalized version of the puzzle. From the constraint programming point of view, the Enigma 1225 puzzle exhibits interesting symmetries, that can be eliminated using only a small number of constraints added to the model. Furthermore, properties of the puzzle can be used to derive a strong constraint propagation scheme that limits the search once an optimal solution has been found.

Key words: Constraint Programming, Modelling, Symmetry Breaking, Puzzles.

1 Introduction

A puzzle is published each week in the New Scientist magazine [1], called Enigma; each puzzle is given a unique number to identify it. The Enigma 1225 problem was published in the February 8th 2003 issue [1]. It is an interesting mathematical puzzle, which serves as a nice pedagogical example for

Email addresses: h.cambazard@4c.ucc.ie (Hadrien Cambazard), b.osullivan@4c.ucc.ie (Barry O'Sullivan), bms@comp.leeds.ac.uk (Barbara M. Smith).

[1] http://www.newscientist.com

demonstrating the expressiveness of constraint programs. The statement of the puzzle, as it appeared in the magazine, is as follows:

> *First draw a chessboard. Now number the horizontal rows* $1, 2, \ldots 8$ *from top to bottom and number the vertical columns* $1, 2, \ldots 8$ *from left to right. You have to put a whole number in each of the sixty-four squares, subject to the following:*
> *(1) No two rows are exactly the same.*
> *(2) Each row is equal to one of the columns, but not to the column with the same index as the row.*
> *(3) If* N *is the largest number you write on the chessboard then you must also write* $1, 2, \ldots, N-1$ *on the chessboard.*
> *The sum of the sixty-four numbers you write on the chessboard is called total. What is the largest total you can obtain?*

The puzzle can clearly be generalized to a square chessboard of any size, and we deal with the generalized puzzle in this paper. Figure 1 gives three examples of feasible solutions to the puzzle for boards of size 4, 6 and 7. The solution for the puzzle of size 4, with $total = 40$, is optimal. These examples will be used throughout the paper.

	C_1	C_2	C_3	C_4
R_1	1	1	2	3
R_2	1	1	3	2
R_3	3	2	4	4
R_4	2	3	4	4

	C_1	C_2	C_3	C_4	C_5	C_6
R_1	1	2	1	3	4	5
R_2	1	1	2	5	3	4
R_3	2	1	1	4	5	3
R_4	5	4	3	6	7	6
R_5	3	5	4	6	6	7
R_6	4	3	5	7	6	6

	C_1	C_2	C_3	C_4	C_5	C_6	C_7
R_1	1	**2**	**2**	2	2	1	3
R_2	2	4	5	5	**4**	2	2
R_3	2	4	4	5	5	**2**	2
R_4	2	5	4	4	5	2	2
R_5	2	5	5	**4**	4	2	**2**
R_6	3	**2**	2	2	2	1	1
R_7	1	2	**2**	2	2	3	1

Fig. 1. Three examples of feasible solutions to the puzzle for different sized boards.

Csenki [3] has described a Prolog-based approach to modelling and solving the generalized puzzle. In any feasible solution to the puzzle, the rows of the chessboard are a permutation of the columns, such that no column index is mapped to the same row index. (Permutations with this restriction are called *derangements*.) Given any such permutation of $1, 2, ..., n$, Csenki shows how to find the solution with the highest value of *total* for that permutation, for an $n \times n$ chessboard. This reduces the problem to finding the permutation of $1, 2, ..., n$ that will yield the highest *total*. Csenki further shows that there are classes of permutations that are equivalent, in that all members of the class give solutions with the same maximum value of *total*, so that only one representative of each class need be considered. Each class corresponds to a partition of n, and Csenki uses a recursive algorithm from discrete mathematics for enumerating partitions of integers. The approach is strongly based on Prolog's logic programming paradigm, specifically taking advantage of unification.

In this paper we give an alternative approach based on the constraint programming paradigm, leading to a complementary analysis of the puzzle. While constraint programs are often solved using a form of backtrack search, efficient inference about local consistencies through the use of generic constraint propagation algorithms and problem-specific global constraints allow us to easily exploit problem-specific knowledge in a constraint model. Rather than explicitly enumerating the partitions of n, as Csenki does, we write constraints that implicitly describe the representative permutations. We also design an upper bound on the value of *total* that can be easily incorporated into a constraint approach. The combination of the upper bound and the implicit enumeration of representative permutations avoids the inherently exponential behaviour of a *generate-and-test* approach that lists all feasible solutions to prove optimality. Using constraint propagation we can considerably reduce the number of feasible solutions we need to explore.

This paper is organized as follows. Section 2 presents several observations and theoretical results on the problem, highlighting their importance from a constraint solving point of view. In Section 3 we briefly introduce constraint programming and present two models of the Enigma 1225 puzzle: a simple one and a more advanced model. We present some empirical results summarising the behaviour of each of our models in Section 5, showing the power of exploiting problem-specific knowledge in a constraint model. We conclude in Section 6.

2 An Analysis of the Enigma 1225 Puzzle

2.1 From Permutations to Integer Partitions

Using a standard notation for permutations, we can denote a column-to-row permutation of the board using a matrix in which the first row represents the columns of the board and the second row represent the rows of the board. For example, the solutions given Figure 1 are based on the three permutations π_1, π_2, π_3 defined as follows:

$$\pi_1 = \begin{bmatrix} 1\ 2\ 3\ 4 \\ 2\ 1\ 4\ 3 \end{bmatrix} \quad \pi_2 = \begin{bmatrix} 1\ 2\ 3\ 4\ 5\ 6 \\ 2\ 3\ 1\ 5\ 6\ 4 \end{bmatrix} \quad \pi_3 = \begin{bmatrix} 1\ 2\ 3\ 4\ 5\ 6\ 7 \\ 7\ 3\ 4\ 5\ 2\ 1\ 6 \end{bmatrix}. \tag{1}$$

The interpretation of π_1 is that column 1 is equal to row 2, column 2 = row 1, column 3 = row 4 and column 4 = row 3. Any feasible solution to the puzzle is defined by such a permutation of $1, \ldots, n$ due to Rule 2 of the puzzle statement. In [3] it was noted that some sets of permutations give rise

to solutions whose overall totals are equivalent and one need only consider a representative of each set of equivalent permutations. We briefly recall this property since it allows us to greatly reduce the number of permutations we need to consider when searching for an optimal solution to the puzzle.

A permutation can be represented alternatively using cycle notation, showing its decomposition into disjoint cycles. For example, π_3 is composed of a 3-cycle and a 4-cycle and can be written as: $\pi_3 = (1\ 7\ 6)(2\ 3\ 4\ 5)$. One can read this cycle representation of the permutation as follows: for the first cycle, $1 \rightarrow 7$, $7 \rightarrow 6$ and $6 \rightarrow 1$, closing the cycle, and for the second cycle, $2 \rightarrow 3 \rightarrow 4 \rightarrow 5 \rightarrow 2$.

The cycle representation of π_3 is associated with the *integer partition* $\{3, 4\}$ of 7 such that $7 = 3 + 4$. The main result of [3] was, essentially, to show that all permutations of $1, \ldots, n$ whose cycle representation is associated with the same integer partition of n are equivalent, in the sense that they lead to solutions of the puzzle with the same overall value of *total*. For example, π_3 and ρ_3 yield the same value of *total*, where:

$$\rho_3 = (1\ 3\ 4)(2\ 5\ 6\ 7). \tag{2}$$

To demonstrate the equivalence, we can show how a solution derived from π_3 can be transformed into a solution derived from ρ_3 by applying a permutation to the columns and rows. A suitable permutation can be found by writing the cycle decomposition of each permutation in two rows, in such a way that cycles of equal length are aligned. Define σ to be the permutation that takes a value in the first row to the corresponding element in the second row. We then have, for example:

$$\sigma = \begin{bmatrix} 7\ 6\ 1\ 3\ 4\ 5\ 2 \\ 1\ 3\ 4\ 2\ 5\ 6\ 7 \end{bmatrix} = (1\ 4\ 5\ 6\ 3\ 2\ 7). \tag{3}$$

Clearly the required permutation σ is not uniquely defined: we could instead write the two permutations as $\pi_3 = (1\ 7\ 6)(2\ 3\ 4\ 5)$ and $\rho_3 = (1\ 3\ 4)(2\ 5\ 6\ 7)$ and get $\sigma = (1)(2)(3\ 5\ 7)(4\ 6)$.

The ρ_3 solution is built by applying σ to the columns of the solution based on π_3 and then to the rows of the resulting matrix (or v.v.). The two steps of the transformation are shown in Figure 2.

One can check that the permutation associated with the solution on the right of Figure 2 is indeed ρ_3.

Thus, rather than considering the permutations of n we need only consider

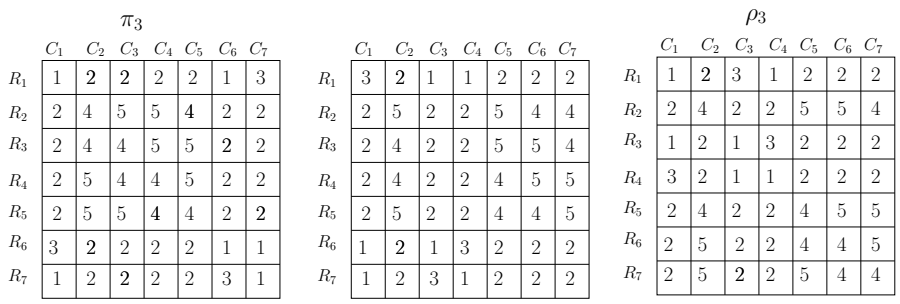

Fig. 2. Transformation of a solution based on π_3 into a solution based on ρ_3.

the integer partitions of n. Moreover, no unit cycle is allowed, to fulfil the second part of Rule 2, and hence partitions containing a 1 can be ignored. We will, therefore, focus on integer partitions in the remainder of the paper, and specifically on the link between a given partition (giving the permutation linking the rows and columns) and the objective function, i.e. *total*.

2.2 Inferring Strong Upper Bounds on the Objective

The objective associated with the Enigma 1225 puzzle is to maximise the sum of the values in the cells of the board. It is, therefore, critical to design upper bounds that can be used during search to avoid unpromising solutions. Optimization is successfully addressed in constraint programming when propagation is able to prune the search space based on inferences derived from the value of the objective function. In this section we address this issue in detail and show that strong bounds can be obtained on the objective function of this problem. Rule 2 of the puzzle can be used to infer bounds on the minimal number of occurrences of values related to a cycle composition. We will use this information in a constraint model later in the paper.

Suppose that a solution to the puzzle is derived from a permutation π. We denote by (i, j) the cell of the board located in row i and column j. Observe that a cell (i, j) will take the same value as a cycle of cells linked by the permutation π as follows:

$$(i,j) \rightarrow (\pi(j), i) \rightarrow (\pi(i), \pi(j)) \rightarrow (\pi^2(j), \pi(i)) \rightarrow (\pi^2(i), \pi^2(j))$$

$$\rightarrow (\pi^3(j), \pi^2(i)) \rightarrow \ldots \rightarrow (i,j)$$

where $\pi^m(i)$ denotes the composition of π, m times, i.e.:

$$\pi^m(i) =_{def} \overbrace{\pi(\pi(\ldots\pi(i)))}^{m}.$$

We denote by $eqc_\pi(i, j)$ the set of cells that take the same value as (i, j) due to π, including (i, j) itself. For example, $eqc_{\pi_3}(6, 1) = \{(6, 1), (7, 6), (1, 7)\}$ because $(6, 1) \rightarrow (7, 6) \rightarrow (1, 7) \rightarrow (6, 1)$. In the solution derived from π_3

shown in Figure 1, these three cells have the value 3. Moreover, we say that a cell (i, j) is a cell of a cycle τ if $i \in \tau$ or $j \in \tau$. A value belongs to a cycle if it occurs in at least one cell of the cycle. For example, the values $\{1, 2, 3\}$ belong to cycle $\tau_1 = (1\ 7\ 6)$ of π_3.

Property 1 (Value Occurrence in Even-length Cycles.) *Let π be a permutation of $1, 2, ..., n$ and τ a cycle of size k of π, where k is even. Every value belonging to τ occurs at least $2k$ times on the board.*

Proof. We need to show that for a given arbitrary cell (i, j) of τ, $|eqc_\pi(i, j)| \geq 2k$. Observe that in the cycle of cells with the same value listed earlier, two kinds of cell occur alternately: $(\pi^s(i), \pi^s(j))$ and $(\pi^s(j), \pi^{s-1}(i))$. Therefore, the cycle of cells of equal value can only end if:

$$\begin{cases} \pi^s(i) = i \\ \pi^s(j) = j \end{cases} \quad \text{or} \quad \begin{cases} \pi^s(j) = i \\ \pi^{s-1}(i) = j \end{cases} \tag{4}$$

In the first case, we can only have $\pi^s(i) = i$ if s is a multiple of k, from the definition of the cycle. The smallest possible value of s is k, and in that case there are $2k$ cells in the cycle.

In the second case, combining the two equations gives $\pi^s(\pi^{s-1}(i)) = i$. Again, this can only happen if $2s-1$ is a multiple of k, but since k is even this cannot happen.

Hence, any value involved in the cycle occurs at least $2k$ times. ■

Property 2 (Value Occurrence in Odd-length Cycles.) *Let π be a permutation of n and τ a cycle of size k of π, where k is odd. One value belonging to τ occurs exactly k times and all others at least $2k$ times.*

Proof. It follows from the previous proof of Property 1 that if k is odd, the minimum number of occurrences of a value is at least k, because we can then have $s + s - 1 = k$, and k cells are required in the chain. In fact, only one sequence of cells can be exactly of size k. Indeed the second condition of Equation 4, which is needed to have exactly k equal cells, implies that both i and j are in the cycle ($i, j \in \{1, \dots k\}$) and at the same time $\pi^{(k+1)/2}(j) = i$ and $\pi^{(k-1)/2}(i) = j$. For any i among $\{1 \dots k\}$, j is determined uniquely by these equations. Consider for example the cycle $(1\ 4\ 2\ 3\ 5)$ with $k = 5$. For $i = 1$, the single value of j such that $\pi^{(k+1)/2}(j) = \pi^3(j) = 1$ and $\pi^{(k-1)/2}(i) = \pi^2(1) = j$ is 2. As j is determined for any i in $\{1 \dots, k\}$ by the second part of Equation 4, there can only be k cells that can fulfil this condition. In other words, there is only one set of exactly k cells that takes the same value. All other values occur at least $2k$ times, to fulfil the first condition of Equation 4. ■

For example, π_2 has a cycle of length 3 and in the corresponding solution, the value 2 occurs exactly three times (in the cycle of equal cells $(3,1) \to (2,3) \to (1,2) \to (3,1)$) and the value 1 occurs six times.

Properties 1 and 2 show that optimal solutions will tend to come from a permutation consisting of small cycles, to allow for more distinct values to be included on the board. This can be used to derive upper bounds on the maximum number of distinct values used on the board, and that can be translated into bounds on the objective function itself.

Property 3 (Number of Distinct Values.) *The cells of a cycle of size k in a board of size n contain at most $nval(n,k)$ distinct values where:*

$$nval(n,k) = \begin{cases} n - k/2 & \text{if } k \text{ is even,} \\ n - (k-1)/2 & \text{otherwise.} \end{cases}$$

Proof. A cycle of size k involves k rows, containing kn cells, and k columns, also containing kn cells. There are k^2 cells in the block where these rows and columns intersect. Hence, the number of cells in the cycle is $2kn - k^2$.

The number of distinct values in these cells is maximized if each value is used as few times as possible. So in the even case, we have:

$$nval(n,k) = (2kn - k^2)/2k = n - k/2$$

because in the best case every value occurs exactly $2k$ times (see Property 1). In the odd case, we have:

$$nval(n,k) = (2kn - k^2 - k)/2k + 1 = n - (k-1)/2$$

because in the best case one value occurs k times and all others $2k$ times. ■

This immediately gives us a simple upper bound on the number of distinct values in an optimal solution of the puzzle.

Property 4 (Upper Bound on the Number of Distinct Values.) *Let $C = \{c_1, \ldots, c_k\}$ be a non-increasing partition of n, i.e. $c_i \geq c_{i+1}$, for $i = 1, 2, ..., k-1$. Define* $\textsc{ubNVal}(n, \{c_1, \ldots, c_k\})$ *recursively as follows:*

$$\begin{cases} \textsc{ubNVal}(0, \{\emptyset\}) = 0 \\ \textsc{ubNVal}(n_i, \{c_i, \ldots, c_k\}) = nval(n_i, c_i) + \textsc{ubNVal}(n_i - c_i, \{c_{i+1}, \ldots, c_k\}) \end{cases} \tag{5}$$

where $n_i = c_i + c_{i+1} + ... + c_k$.

$\textsc{ubNVal}(n, \{c_1, \ldots, c_k\})$ *is an upper bound on the number of distinct values in a solution to the Enigma 1225 puzzle based on a permutation corresponding*

to the partition $C = \{c_1, c_2, ..., c_k\}$ *of* n.

Proof. In constructing this upper bound, the elements of the partition are considered independently. For each one, the maximum possible number of values involved in the corresponding cycle (see Property 3) is added to the bound. This can be visualized in Figure 3. The first cycle, of length c_1, acts on the dashed area at the top and left of the matrix and at most $nval(n, c_1)$ different values can occur in this area. The board is then reduced to a size of $n - c_1 \times n - c_1$ and the process continues with c_2. In this way, the total area of the board is shared between the cycles. The resulting $\text{UBNVAL}(n, \{c_1, \ldots, c_k\})$ is an upper bound, because we are ignoring any interaction between the cycles: for instance, the cycle of length c_2 does act on the area of the board allocated to the first cycle in Figure 3, although we assume that it does not, and this could reduce the number of distinct values in the area allocated to c_1.

Any ordering of the partition would thus give a valid upper bound. By sorting the values in the partition to be non-increasing, the largest cycles are applied to the largest area of the board, thus giving the tightest upper bound.

For example, $\text{UBNVAL}(6, \{4, 2\}) \leq \text{UBNVAL}(6, \{2, 4\})$, since in the first case we have $\text{UBNVAL}(6, \{4, 2\}) = nval(6, 4) + nval(2, 2) = (6 - 2) + (2 - 1) = 5$, and in the second case, $\text{UBNVAL}(6, \{2, 4\}) = nval(6, 2) + nval(4, 4) = (6 - 1) + (4 - 2) = 7$. ■

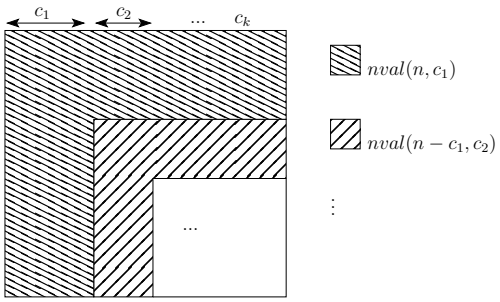

Fig. 3. Illustration of Property 4.

We need to go a step further to obtain an upper bound that can yield useful information even if the partition is not completely known. Notice, to this end, that $\text{UBNVAL}(n, C)$ is maximized by $C = \{2, \ldots, 2\}$ when n is even and by $C = \{3, 2, \ldots, 2\}$ when n is odd. Smaller cycles increase the bound on the number of values. Indeed, for n_1, n_2 odd:

$$nval(n_1, 2) > nval(n_2, k) \qquad \forall n_1 \geq n_2 \text{ and } k > 2$$

since $n_1 - 1 > n_2 - k/2$ for $n_1 \geq n_2$ and $k > 2$, and similarly if n_1, n_2 are even:

$$nval(n_1, 3) > nval(n_2, k) \qquad \forall n_1 \geq n_2 \text{ and } k > 3.$$

A partial partition C' can, therefore, be extended to a complete partition C by adding $2, \ldots, 2$ or $3, 2, \ldots, 2$ to reach n, depending on whether the remaining size is even or odd, and C maximizes the upper bound $\text{UBNVAL}(n, C)$ amongst all partitions that contain C'. Given this complete partition C, an upper bound on the objective value, S, can be immediately obtained by considering the value $\text{UBNVAL}(n, C)$ placed in all cells. The objective is at most $n^2 \times \text{UBNVAL}(n, C)$, which is an ideal, but illegal, case where the value $\text{UBNVAL}(n, C)$ is placed in every cell in the grid. Although this bound could be improved further, it is sufficient to greatly reduce the search space.

We will now show how the previous results can be embedded in a simple constraint model, and then into a better model giving a very efficient solver for the generalised puzzle, i.e. for arbitrary board sizes.

3 A Constraint Programming Model

Constraint Programming [8] is a well-known paradigm for solving combinatorial problems. The problem to be solved is stated as a constraint satisfaction problem (CSP), described in terms of a set of variables, each with a finite set of possible values, its *domain*, together with a set of constraints. Each constraint specifies assignments that are not allowed for some subset of the variables. A solution to the CSP is an assignment to every variable of a value from its domain, in such a way that all the constraints are satisfied. This framework can be extended to optimization problems by adding an objective function and requiring the solution that maximizes or minimizes this function, as appropriate.

CSPs are normally solved using depth-first backtrack search interleaved with constraint propagation. The search typically proceeds by a series of binary choices, $var = val$ versus $var \neq val$, where var is a CSP variable that has not yet been assigned a value, and val is a value in its domain. The aim of constraint propagation is reduce the search space by making logical inferences from the constraints of the problem, including the constraints added at choice points during the search, since $var = val$ and $var \neq val$ may be considered as constraints. Although other kinds of logical inference are possible, constraint solvers usually restrict constraint propagation to removing those values from the domains of variables that can be seen to conflict with the constraints. Backtracking is triggered when some variable is found to have an empty domain as a result of constraint propagation.

Global constraints [5] give constraint solvers significant power by capturing some recurrent and well-known subproblem that can be solved efficiently using a propagation algorithm specific to the global constraint.

We present an initial simple constraint programming model in Figure 4. While this model is a correct formulation of the puzzle, we will show later in the paper that it is quite inefficient. We will present a much more efficient model based on the problem-specific knowledge we have gained in the previous section about the puzzle.

Objective: $maximize(S)$, the sum of the entries on the board.
Variables

S	$\in \{1, \ldots, ub(S)\}$	
N	$\in \{1, \ldots, ub(N)\}$	
p_i	$\in \{1, \ldots, n\}$	$\forall\, i \leq n$
x_{ij}	$\in \{1, \ldots, ub(N)\}$	$\forall\, i, j \leq n$
o_i	$\in \{0, \ldots, n^2\}$	$\forall\, i \leq ub(N)$

Constraints

C_1 :	$\forall i, j \leq n, i \neq j$	$x_{i1} \neq x_{j1} \vee \ldots \vee x_{in} \neq x_{jn}$
C_2 :	$\forall i \leq n$	$p_i = j \Leftrightarrow x_{1i} = x_{j1} \wedge \ldots \wedge x_{ni} = x_{jn}$
C_3 :		$gcc(\{x_{ij} \mid i, j \leq n\}, [o_1, \ldots, o_{ub(N)}])$
C_4 :	$\forall i \leq ub(N)$	$o_i = 0 \Leftrightarrow N < i$
C_5 :		$alldifferent(\{p_1, \ldots, p_n\})$
C_6 :	$\forall i \leq n$	$p_i \neq i$
C_7 :		$N = max(\{x_{ij} \mid i, j \leq n\})$
C_8 :		$S = \sum_i o_i \times i$

Fig. 4. A simple constraint model for the Enigma 1225.

The specification of the simple constraint model in Figure 4 is written in a notation quite close to standard declarative constraint modelling languages such as ILOG OPL [2]. However, we have used Choco [3], an open-source constraint programming system in this work.

S is the variable that represents the objective function, which is the sum of all cells of the board; the objective is to maximize S. N is the variable corresponding to the number of distinct values used in the grid. The set of possible values of N, i.e. its *domain*, is between 1 and $ub(N)$, an upper bound for N. As described in the last section, $\textsc{ubNVal}(n, \{2, 2, \ldots, 2\})$ (n even) or $\textsc{ubNVal}(n, \{3, 2, \ldots, 2\}))$ (n odd) are used for $ub(N)$. For the upper bound on S, $ub(S)$, we use $n^2 \times ub(N)$. (Note that apart from its use in calculating the initial bound on N, we make no use of the upper bound on the number of distinct values in this model.)

The p_i variables encode the relationship between the rows and columns: $p_i = k$ means that column i is equal to row k. The x_{ij} variables correspond to the cells of the board, where $x_{ij} = k$ means that value k will be placed in the

[2] `http://www.ilog.com`
[3] `http://choco.sourceforge.net/`

square row i and column j, and their domain ranges from 1 to $ub(N)$. Finally, o_i expresses the number of occurrences of value i in the grid.

Constraint C_1 expresses Rule 1 of the puzzle (no two rows can be exactly the same). Constraint C_2 expresses that $p_i = j$ if and only if row i is equal to column j. No column can be equal to the row of the same index because of constraint C_6. Constraint C_5 combined with the domains of the p_i variables ensure that the values assigned to them must be a permutation of $1, 2, ..., n$: there are n variables with n values between them, and they must all have different values. C_1 and C_2 could be expressed as stated in Figure 4 using the Boolean connectors *and*, *or*, etc., that are usually available in constraint solvers. However, for efficiency reasons, we implemented them in Choco using dedicated constraints.

The fact that all numbers between 1 and N should appear on the board is stated by a global cardinality constraint [7]: $gcc(\{v_1, \ldots, v_n\}, [o_1, \ldots, o_N])$ that forces value i to occur exactly o_i times amongst the set of variables $\{v_1, \ldots, v_n\}$. C_7 defines N to be the largest number appearing on the board, and constraint C_4 says that every number up to and including N must appear on the board. Constraint C_8 defines S to be the sum of all the cells on the board, via the occurrence variables $o_1, ..., o_{ub(N)}$.

For this basic model, the decision variables are the p_i and x_{ij} variables. The search is performed first on the p_i variables and then switches to the x_{ij} variables. Searching first on the x_{ij} variables gives much worse performance: it results in assignments to the x_{ij} variables that cannot satisfy C_2 and thus leads to wasted search effort.

For both sets of variables, the next variable chosen for instantiation is the one with smallest remaining domain (MinDomain), breaking ties by choosing the variable with smallest index. This MinDomain variable ordering heuristic is a standard search heuristic in constraint programming that often gives good results. It is a dynamic heuristic, i.e. the variable order is not fixed in advance of the search, since it takes into account the effect of constraint propagation on the domains of the variables during the search. Having chosen a variable, the smallest value in its domain is assigned to it.

As mentioned earlier, we implemented the basic constraint model using the latest version of the Choco Constraint Programming system [6], running on a dual core 2Ghz MacBook with 2Gb of Ram. We present our results in Table 1. We report the optimal value found, the time (in seconds) required to find and optimal solution and prove optimality, the number of nodes in the search tree, the number of backtracks and the optimal partition found.

It should be noted that the basic model is not sufficient to solve even small instances of the puzzle in a reasonable amount of time. With a time-limit of

Table 1
Empirical results using the basic model.

n	Objective value	Time (sec.)	Nodes	Back-tracks	Optimal partition
3	15	0.01	8	10	{3}
4	40	0.43	249	588	{2, 2}
5	55	0.88	2407	5824	{5}

one hour, when $n = 6$, the optimal solution can be found, but its optimality cannot be proven.

4 An Improved Model

The basic model is much worse than the faster of the two Prolog implementations presented by Csenki [3], who reported a "near instantaneous response" for $n = 14$. Our basic model has two main disadvantages compared with the Prolog approach. Firstly, once a permutation has been found (i.e. an assignment to the p_i variables), it still needs to search for the optimal assignment to the x_{ij} variables for that permutation. Secondly, it does not take any advantage of Csenki's observation that only one representative permutation for each partition of n need be considered. Our second model will address these two drawbacks. The new model is given Figure 5 and its improvements are detailed in the following two sections.

4.1 Improving Constraint Propagation

Properties 1 and 2 can be used to reason about the minimal number of occurrences of values. For example, they show that every value $\leq N$ must occur at least 3 times, so that the domain of o_i can be restricted to $\{0, 3, \ldots, n^2\}$ rather than $\{0, \ldots, n^2\}$. This allows the *gcc* constraint to remove more values from the domains of the variables.

We have shown in Section 2.2 that given a partition C of n, these properties can be further used to provide an upper bound on the number of distinct values N, and hence a bound on the objective function, S. As the main computational task is the maximization of S, a simple constraint (C_{12}) can be written to propagate that upper bound as follows.

First of all, the exact value of S can be calculated if the permutation is known (i.e. p_i instantiated but x_{ij} unknown, $\forall 1 \leq i, j \leq n$) by computing the exact

cardinality of all sets $eqc_p(i,j)$, the set of cells that take the same value as (i,j), as defined in Section 2.2. Algorithm 1 runs in $O(n^2)$ time, and computes the sets of equal cells implied by a permutation π. Moreover, the calculation can be extended to assign the x_{ij} variables to their optimal values for this permutation.

Algorithm 1 computeSFromPermutation(permutation π)

```
1: eqcSet ← ∅; {set of sets of equal cells}
2: for all cells (i, j) not marked do
3:    set ← ∅;
4:    (a, b) ← (i, j);
5:    repeat
6:       set ← set ∪ (a, b);
7:       mark (a, b);
8:       (a, b) ← (π(b), a);
9:    until (a, b) = (i, j)
10:   eqcSet ← eqcSet ∪ {set};
11: end for
12: sort the sets in eqcSet in increasing order of size;
13: c ← |eqcSet|;
14: for all set_k ∈ eqcSet from the smallest (set_1) to the biggest (set_c) do
15:    for all cells (i, j) ∈ set_k do
16:       instantiate x_ij to k;
17:    end for
18: end for
19: return Σ_k |eqcSet.get(k)| × k;
```

Secondly, consider a partial (possibly empty) assignment of the variables in P, such that the cycle composition (integer partition) consistent with this partial assignment is not completely defined. In Section 2.2 we explained how to compute the best possible partition that maximizes the upper bound. Algorithm 2 summarizes the computation of the upper bound on S. If the permutation variables are not ground then we build the partial partition implied by p_i and complete it as explained in Section 2.2 to derive the bound.

Algorithm 2 upperBoundOnS(n, $\pi = \{p_1, \ldots, p_n\}$)

```
1: if all p_i are instantiated then
2:    return computeSFromPermutation(π);
3: else
4:    part ← partial partition build by closing all open cycles defined by the
      partial assignment of p_i.
5:    complete part by 2, ..., 2 or 3, 2, ..., 2 depending on whether the re-
      maining size is even or odd, respectively.
6:    return ubNVal(n, part) × n^2.
7: end if
```

Algorithm 3 gives the details of the computation of UBNVAL(n, *partition* p), the upper bound presented in Section 2.2.

Algorithm 3 UBNVAL(n,partition p)

```
1: ubNVal ← 0;
2: for all k in the partition (from the biggest to the smallest) do
3:    if k is even, add n − k/2 to ubNVal;
4:    else add n − (k − 1)/2 to ubNVal;
5:    n ← n − k;
6: end for
7: return ubNVal;
```

Consider, for example, $n = 11$ and the following partial assignment:

$$\begin{bmatrix} p_1 & p_2 & p_3 & p_4 & p_5 & p_6 & p_7 & p_8 & p_9 & p_{10} & p_{11} \\ 2 & 3 & ? & ? & 1 & ? & ? & 9 & 10 & ? & ? \end{bmatrix}. \tag{6}$$

The best extension of this partial permutation has two cycles of lengths 4 and 3: (1 2 3 5) and (8 9 10). The best partition extending the assignment is, therefore, $\{4, 3, 2, 2\}$; UBNVAL$(11, \{4, 3, 2, 2\}) = 9 + 6 + 3 + 1 = 19$ and $ub(S) = 19 \times 11^2$.

Once a complete assignment to the p_i variables has been found, the best possible values for the x_{ij} variables are calculated by Algorithm 1 and the best possible value of a solution based on this assignment is then known; this either gives a new best solution, or, if the value calculated is worse than the best solution already found, triggers backtracking to look for a better assignment.

Hence, having a tight bound on the value of S allows many assignments to the p_i variables to be abandoned; others will fail immediately when completed, once the optimal solution has been found. Algorithm 2 is used as the propagation algorithm of constraint C_{12} of Figure 5 that only updates the upper bound of variable S. It is triggered at every change of the state of variables p_i.

4.2 Equivalent Solutions

As already described, Csenki shows that permutations that have the same cycle decomposition are equivalent in the sense that they lead to solutions to the puzzle that have the same value of *total*. In searching for the optimal solution it is therefore important to consider only one representative permutation for each possible cycle decomposition; considering more than one such permutation is clearly a waste of effort.

We can consider that permutations with the same cycle decomposition are symmetrically equivalent. For any given permutation, the optimal solution for that permutation is equivalent to the optimal solution for any other permutation with the same cycle decomposition; or for cycle decompositions containing a 1-cycle, the permutations are equivalent in having no feasible solution to the puzzle.

Symmetry in constraint programming is an established research area [2,4]. Suppose the symmetry group of a CSP is G. Any $g \in G$ applied to an assignment $v_i = j$ is another assignment $g(v_i = j)$. If $V_1 = \{v_1 = l_1, v_2 = l_2, ..., v_k = l_k\}$ is a complete assignment, then $\{g(v_1 = l_1), g(v_2 = l_2), ..., g(v_k = l_k)$, for any $g \in G$, is also a complete assignment (i.e., it assigns exactly one value to each variable, and is a solution to the CSP if and only if V_1 is a solution. The effect of a symmetry g on an assignment depends on the symmetry: symmetries may affect only the values, so that $g(v_i = l_i)$ is $v_i = g(l_i)$, whereas other symmetries permute the variables, so that $g(v_i = l_i)$ is $v_{g(i)} = l_i$.

In this case, given any assignment to the variables p_i and the variables x_{ij}, applying any permutation σ of $1, 2, ..., n$ to both the rows and the columns gives an equivalent assignment. Hence, σ is a symmetry of the problem, where $\sigma(p_i = j) \rightarrow (p_{\sigma(i)} = \sigma(j))$ and $\sigma(x_{ij} = k) \rightarrow (x_{\sigma(i)\sigma(j)})$.

In dealing with symmetry in constraint programming, it is unusual to be able to explicitly characterise the symmetry equivalence classes. In this case, we can: two assignments to the p_i variables are equivalent if the permutations that they represent have the same cycle decomposion, and hence each symmetry equivalence class corresponds to a partition of n.

Rather than enumerating the partitions of n, as in [3], the approach which we adopt here is to design a set of constraints that will only allow one permutation for each partition. We introduce a set of new variables $Q = \{q_1, \ldots, q_n\}$ denoting the *inverse* of $P = \{p_1, \ldots, p_n\}$, such that $p_i = j \Leftrightarrow q_j = i$. The *inverse* constraint (constraint C_{11} in Figure 5) enforces such channelling between the two sets of variables and is provided in most standard constraint solvers such as ILOG Solver and Choco [6].

Given a partition of n, the permutation representing that partition is chosen to be the one in which each cycle involves consecutive elements and the cycles are in non-decreasing order of size. For example, if the partition is $\{3, 4\}$, the representative permutation is (1 2 3)(4 5 6 7). Constraint C_9 in Figure 5 enforces the first condition (the elements of each cycle are consecutive).

If $p_i < i$, then element i is the end of a cycle which started with the element p_i, and the first element of the next cycle is $i + 1$. In that case, we have to ensure that the next cycle is at least as long as the cycle just ending. The next cycle will consist of the elements $i + 1, i + 2, ..., k$, where $p_k = i + 1$, and

the length of the cycle is $k-i$, which can be expressed in terms of the inverse variables as $q_{i+1}-i$. The cycle just ending starts with p_i and ends with i, so its length is $i-p_i+1$. The necessary condition to ensure non-decreasing cycles is $q_{i+1}-i \geq i-p_i+1$, or $q_{i+1} \geq 2i-p_i+1$. This is stated in constraint C_{10}.

Eliminating the symmetry in the Enigma 1225 puzzle requires only $2n$ additional constraints. Adding constraints to a model in order to eliminate symmetric assignments is a common approach in constraint programming. However, it is often impractical to eliminate the symmetry completely when the symmetry group is large, because too many constraints would be required. A few special cases are known in which the symmetry group may be large but only a small number of constraints are required to eliminate all the symmetry. The Enigma 1225 puzzle exhibits a type of symmetry that has not been met before in constraint programming, and thus adds to the repertoire of known special cases where symmetry elimination is tractable.

Putting together the improved bounds and elimination of symmetric equivalents gives the improved CP model shown in Figure 5.

Objective: $maximize(S)$, the sum of the entries on the board.
Variables

$$
\begin{array}{lll}
S & \in \{1,\ldots,ub(S)\} & \\
N & \in \{1,\ldots,ub(N)\} & \\
p_i & \in \{1,\ldots,n\} & \forall\, i \leq n \\
q_i & \in \{1,\ldots,n\} & \forall\, i \leq n \\
x_{ij} & \in \{1,\ldots,ub(N)\} & \forall\, i,j \leq n \\
o_i & \in \{0,3,\ldots,n^2\} & \forall\, i \leq ub(N)
\end{array}
$$

Constraints

$$
\begin{array}{lll}
C_1: & \forall i,j \leq n, i \neq j & x_{i1} \neq x_{j1} \vee \ldots \vee x_{in} \neq x_{jn} \\
C_2: & \forall i \leq n & p_i = j \Leftrightarrow x_{1i} = x_{j1} \wedge \ldots \wedge x_{ni} = x_{jn} \\
C_3: & & gcc(\{x_{ij} | i,j \leq n\}, [o_1,\ldots,o_{ub(N)}]) \\
C_4: & \forall i \leq ub(N) & N < i \Leftrightarrow o_i = 0 \\
C_5: & & alldifferent(\{p_1,\ldots,p_n\}) \\
C_6: & \forall i \leq n & p_i \neq i \\
C_7: & & N = max(\{x_{ij} | i,j \leq n\}) \\
C_8: & & S = \textstyle\sum_i o_i \times i \\
C_9: & \forall i \leq n & p_i \leq i+1 \\
C_{10}: & \forall i < n & p_i < i \Rightarrow q_{i+1} \geq 2i+1-p_i \\
C_{11}: & & inverse(P,Q) \\
C_{12}: & & S \leq \textsc{upperBoundOnS}(n,p_1,\ldots,p_n)
\end{array}
$$

Fig. 5. The complete constraint model for the Enigma1225 puzzle.

For the advanced model, the decision variables are just the p_i variables; once these variables are instantiated, i.e. the permutation defining the matrix is known, the x_{ij} variables can be assigned their best possible values for this permutation using Algorithm 3, as already described. Indeed, once a permutation is known, a sharp upper bound on N is propagated and, during the search on P, partial assignments of P will be pruned by the upper-bound constraint.

As before, the search heuristic used for the p_i variables is to choose the variable with smallest remaining domain, breaking ties by choosing the variable with smallest index, and to assign it its smallest possible value. However, the heuristic results in different behaviour in this model, because of the additional filtering of the domains of the p_i variables from constraints C_9 to C_{11}. Here, after initial propagation of the constraints, p_1 has only one possible value (2), since 1-cycles are not allowed. This assignment is made as a result of constraint propagation, since there is no choice involved. The heuristic then chooses p_2, which can be either 1 or 3. The first choice is $p_2 = 1$ (giving an initial 2-cycle); the choice $p_2 = 3$ is made if the search backtracks to this node. If $p_2 = 1$, the domain of p_3 is reduced to just $\{4\}$, and then p_4 can be 3 or 5. Choosing 3 first gives another 2-cycle. Proceeding in this fashion, the search constructs the partition $\{2, 2, 2, 2, 2...\}$ first (with the final cycle being of length 2 or 3 depending on n). Hence, the first permutation constructed leads to the optimal solution (except when $n = 5$).

Once the optimal solution has been found, the search continues to consider other choices for the p_i variables in order to prove optimality. The bounds derived earlier are sufficiently strong that not all possible partitions of n need be considered. Whenever a complete permutation is constructed during the search the calculation of the upper bound on S using Algorithm 1 shows that it is worse than the solution already found, and the search backtracks.

5 Experimental results

The improved constraint model is also implemented in Choco, and the results are given in Table 2. As before, we give the optimal value found, the run time (in seconds), the number of nodes, the number of backtracks and the optimal partition. We also give the number of complete partitions of n that have been explored and the total number of partitions that do not give a cycle of size one. The difference between these counts measures the propagation achieved by the upper bound on S: the larger the difference, the more search reduction

we are getting from polynomial-time constraint propagation.

The advanced model is a significant improvement on the basic model: it solves $n = 20$ more quickly and with much less search than the basic model requires when $n = 5$.

While no direct comparison with the Prolog approach presented in [3] has been made, we can say that the constraint programming approach presented here is more scalable. The Prolog approach generates a representative permutation for every partition of n, in order to find the one that gives the maximum *total*, and the number of partitions increases exponentially with n. Our results show that the proportion of complete partitions explored by the advanced model decreases as n increases, so that when $n = 55$, fewer than 5% of the partitions need be considered.

Except when $n = 5$, the optimal partition found has always the form $\{2, 2, 2, ..., 2\}$ or $\{2, 2, ..., 2, 3\}$, depending on whether n is even or odd, which is consistent with the theoretical results we obtained (Properties 2 and 3), suggesting that small cycles allow us to get more distinct values and a higher objective total. The exception when $n = 5$ is because the partition $\{2, 3\}$ does not allow a solution that satisfies C_1 and C_2. It could undoubtedly be proved that, for $n > 5$, the optimal partition is always $\{2, 2, 2, ..., 2\}$ or $\{2, 2, ..., 2, 3\}$, thus showing the problem to be polynomial. This is, however, not the focus of this paper, where we use the puzzle as a modelling exercise to illustrate an alternative approach to that based on Prolog.

6 Conclusion

A constraint programming approach has been presented for the Enigma 1225 puzzle, giving a complementary view to that presented in [3]. In particular, properties presented about permutations and partitions in [3] have been incorporated in a constraint model with few constraints, and without the need to enumerate partitions of integers. Other interesting properties have been exhibited and used to go a step further than [3] on the optimization side, taking advantage of the strength of constraint propagation.

With our improved model, large instances of the problem can be solved quickly; the search strategy finds the optimal solution immediately, and propagation of a strong upper bound on the objective allows optimality to be proved while considering only a small proportion of the possible partitions of n.

Finally, we believe that puzzles such as this are very interesting for teaching. The basic CP model exhibits very poor performance and successfully solving

Table 2
Empirical results using the advanced model.

n	Objective value	Time (sec.)	Nodes	Back-tracks	Explored partitions	Total partitions	Optimal partition
3	15	0.01	1	0	1	1	{3}
4	40	0.01	2	1	2	2	{2, 2}
5	55	0.02	2	2	2	2	{5}
6	180	0.06	4	4	3	4	{2, 2, 2}
7	275	0.04	5	6	4	4	{2, 2, 3}
8	544	0.09	9	12	6	7	{2, 2, 2, 2}
9	753	0.09	9	12	7	8	{2, 2, 2, 3}
10	1,300	0.2	15	22	7	12	{2, 2, 2, 2, 2}
11	1,703	0.12	18	27	12	14	{2, 2, 2, 2, 3}
12	2,664	0.15	28	43	14	21	{2, 2, 2, 2, 2, 2}
13	3,365	0.19	31	49	19	24	{2, 2, 2, 2, 2, 3}
14	4,900	0.28	45	71	19	34	{2, 2, 2, 2, 2, 2, 2}
15	6,027	0.24	52	84	29	41	{2, 2, 2, 2, 2, 2, 3}
16	8,320	0.32	72	116	28	55	{2, 2, 2, 2, 2, 2, 2, 2}
17	10,025	0.23	78	125	41	66	{2, 2, 2, 2, 2, 2, 2, 3}
18	13,284	0.25	108	174	39	88	{2, 2, 2, 2, 2, 2, 2, 2, 2}
19	15,743	0.35	125	204	60	105	{2, 2, 2, 2, 2, 2, 2, 2, 3}
20	20,200	0.46	167	274	57	137	{2, 2, 2, ..., 2, 2, 2, 2}
21	23,613	0.42	175	287	76	165	{2, 2, 2, ..., 2, 2, 2, 3}
22	29,524	0.52	229	376	70	210	{2, 2, 2, ..., 2, 2, 2, 2}
23	34,115	0.53	244	400	100	253	{2, 2, 2, ..., 2, 2, 2, 3}
24	41,760	0.87	343	572	100	320	{2, 2, 2, ..., 2, 2, 2, 2}
25	47,777	1.08	349	583	133	383	{2, 2, 2, ..., 2, 2, 2, 3}
30	101,700	3.38	881	1,483	204	1,039	{2, 2, 2, ..., 2, 2, 2, 2}
35	185,327	8.84	1,591	2,691	446	2,573	{2, 2, 2, ..., 2, 2, 2, 3}
40	320,800	30.44	3,600	6,170	642	6,153	{2, 2, 2, ..., 2, 2, 2, 2}
45	508,677	66.24	5,688	9,750	1,230	13,959	{2, 2, 2, ..., 2, 2, 2, 3}
50	782,500	208.89	11,865	20,540	1,672	30,701	{2, 2, 2, ..., 2, 2, 2, 2}
55	1,137,827	405.3	17,614	30,577	3,059	65,121	{2, 2, 2, ..., 2, 2, 2, 3}

the problem requires a careful analysis of problem symmetries, propagation and search strategies. The paper demonstrates how important it is to incorporate problem-specific knowledge into a constraint model, and also how this can be done. Such an analysis is very helpful in teaching constraint programming and highlights a number of key concepts for a student.

Acknowledgments

This work was supported by Science Foundation Ireland under Grant No. 05/IN/I886. This work was completed while the third author was employed at the Cork Constraint Computation Centre.

References

[1] K. Austin. Enigma 1225: Rows are columns. *New Scientist*, 55, 8 February 2003.

[2] D. Cohen, P. Jeavons, C. Jefferson, K. E. Petrie, and B. M. Smith. Symmetry Definitions for Constraint Programming. *Constraints*, 11:115–137, 2006.

[3] A. Csenki. Enigma 1225: Prolog-assisted solution of a puzzle using discrete mathematics. *Computers and Mathematics with Applications*, 52:383–400, 2006.

[4] I. P. Gent, K. E. Petrie, and J.-F. Puget. Symmetry in constraint programming. In F. Rossi, P. van Beek, and T. Walsh, editors, *Handbook of Constraint Programming*, chapter 10. Elsevier, 2006.

[5] Willem-Jan Hoeve and Irit Katriel. Global constraints. In F. Rossi, P. van Beek, and T. Walsh, editors, *Handbook of Constraint Programming*, chapter 6. Elsevier, 2006.

[6] F. Laburthe and the OCRE group. CHOCO: implementing a CP kernel. In *CP00 Post Conference Workshop on Techniques for Implementing Constraint programming Systems (TRICS)*, Singapore, September 2000.

[7] J.C. Régin. Generalized arc consistency for global cardinality constraint. In *National Conference on Artificial Intelligence (AAAI'96)*, pages 209–215, 1996.

[8] F. Rossi, P. van Beek, and T. Walsh, editors. *Handbook of Constraint Programming*. Elsevier, 2006.