

# Solving edge-matching problems with satisfiability solvers

Marijn J.H. Heule\*

Department of Software Technology  
Faculty of Electrical Engineering, Mathematics and Computer Sciences  
Delft University of Technology  
marijn@heule.nl

**Abstract.** Programs that solve Boolean satisfiability (SAT) problems have become powerful tools to tackle a wide range of applications. The usefulness of these solvers does not only depend on their strength and the properties of a certain problem, but also on how the problem is translated into SAT. This paper offers additional evidence for this claim.

To show the impact of the translation on the performance, we studied encodings of edge-matching problems. The popularity of these problems was boosted by the release of Eternity II in July 2007: Whoever solves this 256 piece puzzle first wins \$ 2,000,000. There exists no straightforward translation into SAT for edge-matching problems. Therefore, a variety of possible encodings arise.

The basic translation used in our experiments and described in this paper, is the smallest one that comes to mind. This translation can be extended using redundant clauses representing additional knowledge about the problem. The results show that these redundant clauses can guide the search – both for complete and incomplete SAT solvers – yielding significant performance gains.

## 1 Introduction

The Boolean satisfiability (SAT) problem deals with the question whether there exists an assignment –a mapping of the Boolean values to the Boolean variables– that satisfies a given formula. A formula, in Conjunctive Normal Form (CNF), is a conjunction of clauses, each clause being a disjunction of literals. Literals refer either to a Boolean variable  $x$  or to its negation  $\bar{x}$ .

SAT solvers have become very powerful tools to solve a wide range of problems, such as Bounded Model Checking and Equivalence Checking of electronic circuits. These problems are first translated into CNF, solved by a SAT solver, and a possible solution is translated back to the original problem domain.

Translating a problem into CNF in order to solve it does not seem optimal: Problem specific information, which could be used to develop specialized solving methods, may be lost in the translation. However, due to the strength of modern

---

\* Supported by the Dutch Organization for Scientific Research (NWO) under grant 617.023.611.

SAT solvers, it could be very fruitful in practice: Problem specific methods to beat the SAT approach may take years to develop.

SAT solvers have been successfully applied to various combinatorial problems ranging from lower bounds to Van der Waerden numbers [3] to Latin Squares. However, on many other combinatorial problems, such as Traveling Salesman and Facility Allocation [8], SAT solvers cannot compete with alternative techniques such as Linear Programming. A possible explanation is that the former (successful) group can be naturally translated into CNF, while the latter, due to arithmetic constraints cannot.

For most problems, there is no straight-forward translation into CNF. Whether SAT solvers can efficiently solve such problems does not only depend on the strength of the solvers, but also on the translation of the problem into CNF. This paper offers an evaluation of the influence of a translation on the performance of SAT solvers. The translation of edge-matching problems into CNF serves as this paper's experimental environment. The problem at hand appears both challenging and promising; because 1) there is no "natural" translation into CNF, yielding many alternative translations, and 2) there are no arithmetic constraints that seem hard for SAT solvers.

The focus of this paper will be on the influence of *redundant* clauses – those clauses which removal / addition will not increase / decrease the number of solutions. Notice that redundancy as stated above should be interpreted in the *neutral* mathematical sense of the word and not in the *negative* connotation of day-to-day talk. In fact, as we will see, redundant clauses can improve the performance of SAT solvers. Furthermore, all presented encodings will use the same set of Boolean variables.

After introducing edge-matching problems (Section 2), this paper presents the smallest translation into CNF that comes to mind. First the choice of the variables (Section 3), followed by the required clauses (Section 4). This translation can be extended with clauses representing additional knowledge about the problem (Section 5). Then it reflects on the influence of the translation (with and without extensions) on the performance (Section 6) and concludes that encoding is crucial to solve the hardest instances (Section 7).

## 2 Edge-Matching Problems

Edge-matching problems [5] are popular puzzles, that appeared first in the 1890's. Given a set of pieces and a grid, the goal is to place the pieces on the grid such that the edges of the connected pieces match. Edge-matching problems are proved to be NP-complete [2]. Most edge-matching problems have square pieces and square grids. Yet, there exists a large variety of puzzles<sup>1</sup> with triangle or 3D pieces and irregular grids.

---

<sup>1</sup> See for instance <http://www.gamepuzzles.com/edgemtch.htm>

There are two main classes of edge-matching problems. First, the edges are colored and connected edges much have the same color. These problems are called *unsigned*. Second, instead of colors, edges can have a partial image. These edges match if they have complementary parts of the same image. These problems are called *signed*. A famous signed edge-matching problem is Rubik’s Tangle.

Edges on the border of the grid are not connected to pieces, so they cannot match as the other edges. In case there are no constraints placed on these edges, we call the problem *unbounded*. On the other hand, problems are *bounded* if these edges are constraint. A common constraint is that these edges must have the same color. Throughout this paper, when we refer to bounded edge-matching problems, we assume this constraint and that their is a special color only for these border edges.

The popularity of edge-matching problems was boosted by the release of the Eternity II puzzle in July 2007: Whoever solves it first wins \$2,000,000. Eternity II is a  $16 \times 16$  bounded unsigned edge-matching problem invented by Christopher Monckton and published by Tomy. Apart from the large 256 piece puzzle, also four smaller clue puzzles have been released.

### 3 Choosing the Variables

The selection of Boolean variables for the translation is an important first step to construct an efficient encoding. This section introduces the variables used in the proposed translation of edge-matching problems into CNF. These consist of two types: Variables representing a mapping from pieces to squares (Section 3.1) and variables describing the colors of the diamonds (Section 3.2). Apart from these variables, this section describes the clauses relating to variables of the same type. Clauses that consist of both types will be discussed in Section 4.

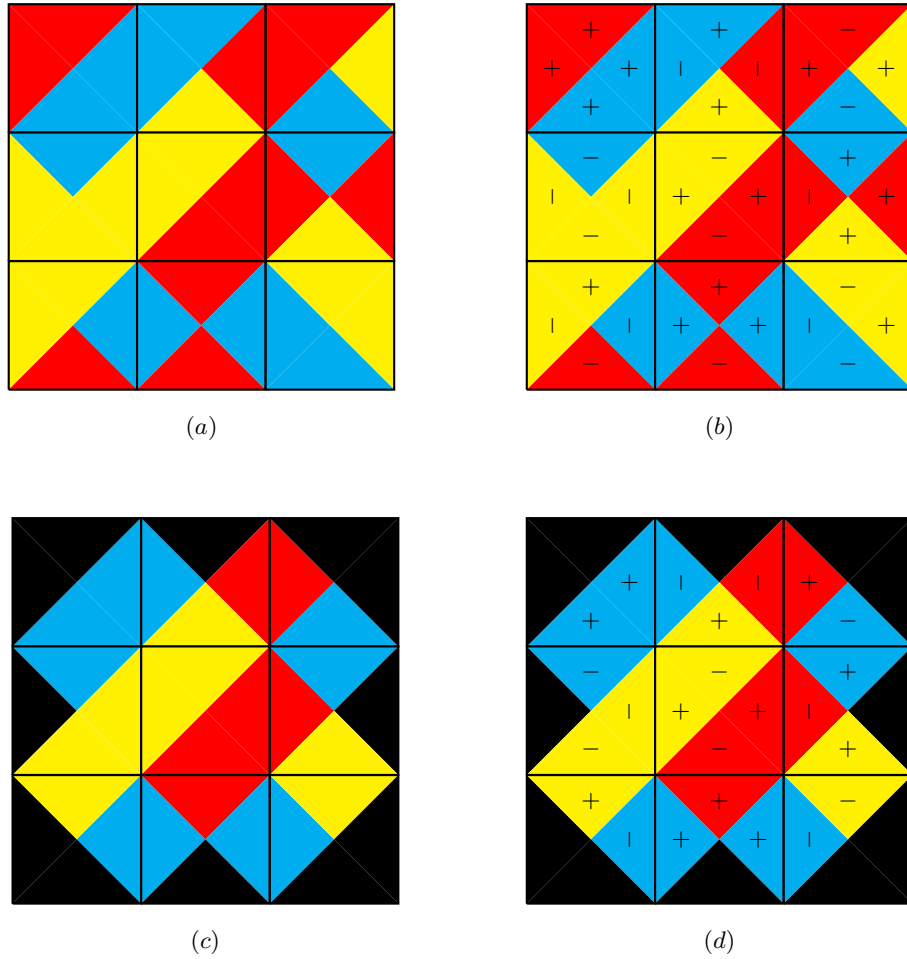
Throughout this paper, no auxiliary variables are introduced. Using only the variables in this section, one can already construct dozens of alternative translations. Therefore, evaluating these translations seems a natural starting point. That said, related work such as [7] shows that auxiliary variables can be very helpful to reduce the computational costs of solving the problem at hand.

#### 3.1 Mapping Pieces to Squares

Arguably the most intuitive way to translate edge-matching problems into CNF would be a mapping of the pieces to the squares of the grid. A similar approach has been proposed to translate edge-matching problems into a Constraint Satisfaction Problem [9]. This requires the following Boolean variables:

$$x_{i,j} \begin{cases} 1 & \text{if piece } p_i \text{ is placed on square } q_j \\ 0 & \text{otherwise} \end{cases}$$

Notice that there is no rotation embedded in the variable encoding. As we will see in Section 4, rotation does not require additional variables and can be achieved by clauses.



**Fig. 1.** Examples of an (a) unbounded unsigned edge-matching problem, (b) unbounded signed edge-matching problem, (c) bounded unsigned edge-matching problem, and (d) bounded signed edge-matching problem.

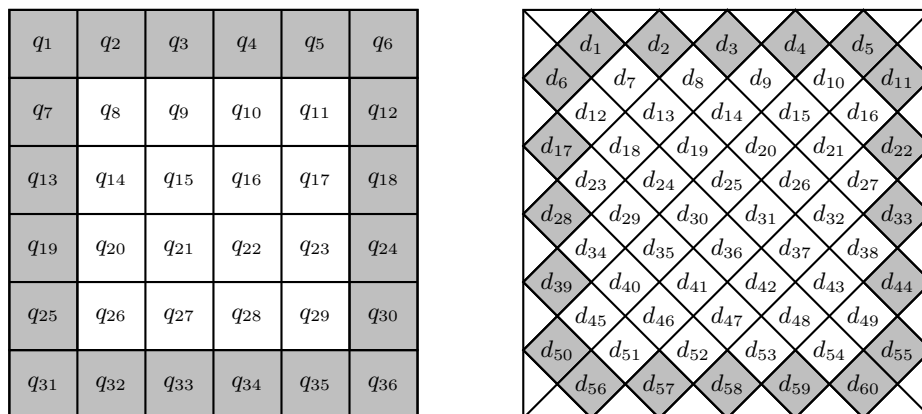
For bounded edge-matching problems, *zero edges* refer to those edges that should be placed along the boundary of the grid. Given a  $n \times n$  grid, these problems contain four corner pieces with two zero edges (denoted by set  $P_{\text{corner}}$ ) and  $4n - 8$  border pieces with one zero edge (denoted by set  $P_{\text{border}}$ ). The pieces with no zero edges are denoted by set  $P_{\text{center}}$ . For unbounded edge-matching problems, all pieces are in set  $P_{\text{center}}$ .

Likewise, corner pieces can only be placed in the corners of the grid (denoted by set  $Q_{\text{corner}}$ ), border pieces only along the border (denoted by set  $Q_{\text{border}}$ ), and the other pieces can only be placed in the center (denoted by set  $Q_{\text{center}}$ ). So, the mapping variables are related as follows:

$$\left( \bigvee_{p_i \in P_{\text{corner}}} x_{i,a} \right) \wedge \left( \bigvee_{p_i \in P_{\text{border}}} x_{i,b} \right) \wedge \left( \bigvee_{p_i \in P_{\text{center}}} x_{i,c} \right) \text{ for } q_a \in Q_{\text{corner}}, q_b \in Q_{\text{border}}, q_c \in Q_{\text{center}} \quad (1)$$

$$\left( \bigvee_{q_j \in Q_{\text{corner}}} x_{a,j} \right) \wedge \left( \bigvee_{q_j \in Q_{\text{border}}} x_{b,j} \right) \wedge \left( \bigvee_{q_j \in Q_{\text{center}}} x_{c,j} \right) \text{ for } p_a \in P_{\text{corner}}, p_b \in P_{\text{border}}, p_c \in P_{\text{center}} \quad (2)$$

The above encoding requires  $|P_{\text{corner}}|^2 + |P_{\text{border}}|^2 + |P_{\text{center}}|^2$  variables and  $2|P_{\text{corner}}| + 2|P_{\text{border}}| + 2|P_{\text{center}}|$  clauses. Notice that the encoding only forces each piece on *at-least-one* square and each square to hold *at-least-one* piece. In fact, in any valid placement, this should be *exactly-one*. Forcing them exactly-one *explicitly* – each mapping of one piece on two squares would violate a specific (binary) clause – is very expensive (in terms of additional (binary) clauses), as we will discuss in Section 5.2. Instead, the clauses presented in Section 4 force the one-on-one mapping *implicitly* which makes the explicit encoding redundant.



**Fig. 2.** The numbering of squares  $q_j$  (left) and diamonds  $d_k$  (right) for a 6x6 edge-matching problem. Gray squares are corner and border squares, gray diamonds are border diamonds.

Implicit encoding assumes that all pieces are unique. In case two pieces are equivalent (modulo rotation), then a few additional clauses have to be added to force equivalent pieces to be placed on different squares. To ensure a valid mapping, we therefore need some additional clauses for each square  $q_j$ :

$$(\overline{x}_{i,j} \vee \overline{x}_{l,j}) \quad \text{for } p_i \text{ equivalent to } p_l \text{ and } i < l \quad (3)$$

### 3.2 Colored Diamonds

The only constraint forced on a placement is that colors of connecting edges must match. Edges are represented as triangles and connected edges as diamonds. Given a  $n \times n$  grid, there are  $n^2 - 2n$  diamonds. Diamonds are numbered from left to right, from top to bottom, see Figure 2. This brings us to the second type of variables.

$$y_{k,c} \quad \begin{cases} 1 & \text{if diamond } d_k \text{ has color } c \\ 0 & \text{otherwise} \end{cases}$$

The colored edges can be partitioned into *border edges* (those directly next to zero edges) and *center edges* (those not directly next to zero edges). Set  $C_{\text{border}}$  consists of all colors of border edges and set  $C_{\text{center}}$  consists of all colors of center edges. Likewise, diamonds are partitioned into two sets, one for the border edges, called  $D_{\text{border}}$ , and one for the center edges, called  $D_{\text{center}}$ . Figure 2 shows the partition for a  $6 \times 6$  grid. The disjunction of  $C_{\text{border}}$  and  $C_{\text{center}}$  could be empty, but that is not as a rule. The number of variables  $y_{k,c}$  equals  $|C_{\text{border}}| \cdot |D_{\text{border}}| + |C_{\text{center}}| \cdot |D_{\text{center}}|$ . In case either  $|C_{\text{border}}|$  or  $|C_{\text{center}}|$  is large, the number of required binary clauses will be enormous.

$$\left( \bigvee_{c \in C_{\text{border}}} y_{k,c} \right) \wedge \left( \bigvee_{c \in C_{\text{center}}} y_{l,c} \right) \wedge \bigwedge_{c,c' \in C_{\text{border}}, c < c'} (\overline{y}_{k,c} \vee \overline{y}_{k,c'}) \wedge \bigwedge_{c,c' \in C_{\text{border}}, c < c'} (\overline{y}_{l,c} \vee \overline{y}_{l,c'}) \quad \text{for } \begin{cases} d_k \in D_{\text{border}} \\ d_l \in D_{\text{center}} \end{cases} \quad (4)$$

**Example 1.** Given an edge-matching problem with  $C_{\text{border}} = \{\text{blue, green, red}\}$  and  $C_{\text{center}} = \{\text{cyan, green, pink, yellow}\}$ . The following clauses will encode that each diamond has exactly one color:

$$\left. \begin{aligned} & (y_{k,\text{red}} \vee y_{r,\text{green}} \vee y_{k,\text{blue}}) \wedge \\ & (\overline{y}_{k,\text{red}} \vee \overline{y}_{k,\text{green}}) \wedge (\overline{y}_{k,\text{red}} \vee \overline{y}_{k,\text{blue}}) \wedge (\overline{y}_{k,\text{green}} \vee \overline{y}_{k,\text{blue}}) \end{aligned} \right\} d_k \in D_{\text{border}} \quad (5)$$

$$\left. \begin{aligned} & (y_{k,\text{cyan}} \vee y_{r,\text{green}} \vee y_{k,\text{pink}} \vee y_{k,\text{yellow}}) \wedge \\ & (\overline{y}_{k,\text{cyan}} \vee \overline{y}_{k,\text{green}}) \wedge (\overline{y}_{k,\text{cyan}} \vee \overline{y}_{k,\text{pink}}) \wedge (\overline{y}_{k,\text{cyan}} \vee \overline{y}_{k,\text{yellow}}) \wedge \\ & (\overline{y}_{k,\text{green}} \vee \overline{y}_{k,\text{pink}}) \wedge (\overline{y}_{k,\text{green}} \vee \overline{y}_{k,\text{yellow}}) \wedge (\overline{y}_{k,\text{pink}} \vee \overline{y}_{k,\text{yellow}}) \end{aligned} \right\} d_k \in D_{\text{center}} \quad (6)$$

## 4 Essential Clauses

This section deals with the question of how to connect the mapping variables  $x_{i,j}$  with the colored diamond variables  $y_{k,c}$ . The encoding presented here is one of many alternatives. This one uses only a few clauses per mapping variable  $x_{i,j}$ . All constraints have the format “if  $p_i$  is mapped on  $q_j$  ..., then  $d_k$  has color  $c$ ”. Or as clause  $(\bar{x}_{i,j} \vee \dots \vee y_{k,c})$ . The number of these clauses and their sizes depend on the type of piece  $p_i$ . Besides corner and border pieces (discussed in Section 4.1), the center pieces are grouped in seven types (see Section 4.2).

### 4.1 Corner and Border Pieces

First the easy part. Recall that the zero edges are known. So, corner and border pieces can only be placed on a square with a specific rotation. Therefore, only one binary clause is required for each non-zero edge of the center and border pieces.

**Example 2.** Given a corner piece  $p_A$  with a red east edge and a blue south edge which should be placed on a  $n \times n$  grid (see Figure 2). Then the eight clauses below should be added (per corner piece depending on the colors). Notice that  $q_1, q_n, q_{n^2-n+1}, q_{n^2}$  are the corresponding corner squares.

$$\begin{aligned} & (\bar{x}_{A,1} \vee y_{1,\text{red}}) \quad \wedge \quad (\bar{x}_{A,1} \vee y_{n,\text{blue}}) \quad \wedge \\ & (\bar{x}_{A,n} \vee y_{2n-1,\text{red}}) \quad \wedge \quad (\bar{x}_{A,n} \vee y_{n-1,\text{blue}}) \quad \wedge \\ & (\bar{x}_{A,n^2-n+1} \vee y_{2n^2-4n+2,\text{red}}) \quad \wedge \quad (\bar{x}_{A,n^2-n+1} \vee y_{2n^2-3n+2,\text{blue}}) \quad \wedge \\ & (\bar{x}_{A,n^2} \vee y_{2n^2-2n,\text{red}}) \quad \wedge \quad (\bar{x}_{A,n^2} \vee y_{2n^2-3n+1,\text{blue}}) \end{aligned}$$

Similarly, given a border piece  $p_B$  with a pink east edge, a yellow south edge and a green west edge, that should be placed on the same grid, the following clauses should be added for  $j \in \{1, \dots, n-2\}$ :

$$\begin{aligned} & (\bar{x}_{B,j+1} \vee y_{j,\text{green}}) \quad \wedge \quad (\bar{x}_{B,j+1} \vee y_{j+n+1,\text{yellow}}) \quad \wedge \\ & (\bar{x}_{B,j+1} \vee y_{j+1,\text{pink}}) \quad \wedge \quad (\bar{x}_{B,nj+1} \vee y_{(2n-1)j+n,\text{green}}) \quad \wedge \\ & (\bar{x}_{B,nj+1} \vee y_{(2n-1)j+1,\text{yellow}}) \quad \wedge \quad (\bar{x}_{B,nj+1} \vee y_{(2n-1)j-n+1,\text{pink}}) \quad \wedge \\ & (\bar{x}_{B,n(j+2)-1} \vee y_{(2n-1)j,\text{green}}) \quad \wedge \quad (\bar{x}_{B,n(j+2)-1} \vee y_{(2n-1)j+n-1,\text{yellow}}) \quad \wedge \\ & (\bar{x}_{B,n(j+2)-1} \vee y_{(2n-1)(j+1),\text{pink}}) \quad \wedge \quad (\bar{x}_{B,j+n^2-n+1} \vee y_{j+2n^2-3n+2,\text{green}}) \quad \wedge \\ & (\bar{x}_{B,n^2-n+1} \vee y_{j+2n^2-4n+2,\text{yellow}}) \quad \wedge \quad (\bar{x}_{B,j+n^2-n+1} \vee y_{j+2n^2-3n+2,\text{pink}}) \end{aligned}$$

Concluding, for each variable  $x_{i,j}$  with  $p_i \in P_{\text{corner}}$  we only need two binary clauses, while for each  $x_{i,j}$  with  $p_i \in P_{\text{border}}$ , we need three binary clauses. The next section will discuss which clauses to add for those  $x_{i,j}$  with  $p_i \in P_{\text{center}}$ .

### 4.2 Center Pieces

Given the choice of the variables presented in Section 3, the encoding of corner and border pieces (as above) is quite straight-forward. However, encoding the

center pieces efficiently is much more tricky. The crux is that if a certain mapping variable  $x_{i,j}$  of a center piece is true, we cannot directly color the corresponding diamonds<sup>2</sup>. We need to know how  $p_i$  is rotated ( $0^\circ$ ,  $90^\circ$ ,  $180^\circ$ , or  $270^\circ$ ).

Rotation can be encoded using two kinds of clauses: *positive rotation clauses* and *negative rotation clauses*. First, positive rotation clauses consist of only positive literals  $y_{k,c}$  and the negative mapping literal  $\bar{x}_{i,j}$ . These clauses force a subset of the corresponding diamonds to be colored in correspondence with one of the edges. The number of these clauses and their sizes depend on how many times a color occurs on a piece. If a color occurs only once then this is encoded as a single clause of length five. If all edges have the same color then a binary clause is required per edge. In the other cases, these clauses have length three and the number depends on the relative location of the edges with the same color. All positive rotation clauses are used in the proposed encoding.

Second, for negative rotation clauses, all literals are negated except for one literal  $y_{k,c}$ . The negated literals represent the conditions to force diamond  $d_k$  to color  $c$ . Most negative rotation clauses are ternary clauses. For instance,  $\bar{x}_{1,5} \vee \bar{y}_{7,\text{yellow}} \vee y_{8,\text{red}}$ , which could be read as “if  $p_1$  is mapped on  $q_5$  and  $d_7$  is yellow, then  $d_8$  is red”. In case a piece contains three or four different colors, some negative rotation clauses are required to make the encoding valid.

Center pieces can be partitioned into seven types: 1) All edges have the same color; 2) three edges have the same color; 3) two neighbouring pairs of edges have the same color; 4) both opposite pairs of edges have the same color; 5) one neighbouring pair of edges has the same color; 6) one opposite pair of edges has the same color; 7) all edges have a different color. The number of clauses that should be added for each variable  $x_{i,j}$  depends on the type of piece  $p_i$  – ranging from 4 (type 1) to 20 (type 7). Figure 3 lists the combination of clauses that should be added per type of each piece.

Notice that the positive rotation clauses of length five are not listed in Figure 3 for types 5, 6, and 7. First, piece type 5 does not need the long positive rotation clauses because the shown clauses are enough to force a valid encoding. For piece types 6 and 7 it is required to add at least of these long clauses. We omitted it in Figure 3, because there is a choice – anyone of them will make the encoding valid. To make the encoding independent of the choice, as stated before, all the positive rotation clauses will be used.


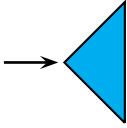

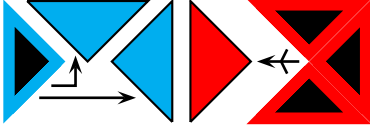

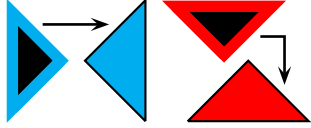
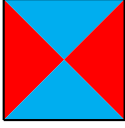
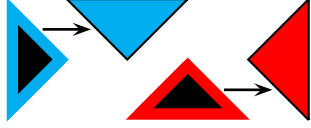

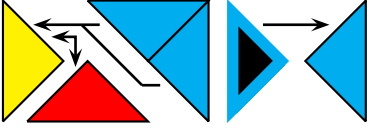

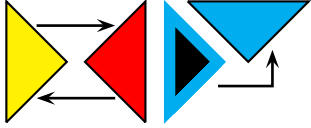

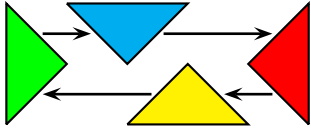
## 5 Redundant Clauses

Translation of edge-matching problems into CNF as presented in Sections 3 and 4 is the smallest one that came to mind. This translation is such that each satisfying assignment corresponds with a unique valid positioning of the pieces. Moreover, the translation is satisfiable if and only if there exists a valid placement of the the original problem.

---

<sup>2</sup> Expect for the special case in which all edges have the some color (and the same sign, for signed problems).



#	Type	Implications	Clauses
1			$(\bar{x}_{i,15} \vee y_{25,\text{blue}})^b$
2			$(\bar{x}_{i,15} \vee y_{24,\text{blue}} \vee y_{19,\text{blue}})^b$ $(\bar{x}_{i,15} \vee y_{24,\text{blue}} \vee y_{25,\text{blue}})^a$ $(\bar{x}_{i,15} \vee y_{19,\text{red}} \vee y_{24,\text{red}} \vee y_{25,\text{red}} \vee y_{30,\text{red}})$
3			$(\bar{x}_{i,15} \vee y_{24,\text{blue}} \vee \bar{y}_{25,\text{blue}})^a$ $(\bar{x}_{i,15} \vee y_{19,\text{red}} \vee y_{30,\text{red}})^a$
4			$(\bar{x}_{i,15} \vee y_{19,\text{blue}} \vee y_{24,\text{blue}})^b$ $(\bar{x}_{i,15} \vee y_{25,\text{red}} \vee y_{30,\text{red}})^b$
5			$(\bar{x}_{i,15} \vee \bar{y}_{24,\text{yellow}} \vee y_{30,\text{red}})^b$ $(\bar{x}_{i,15} \vee y_{24,\text{yellow}} \vee \bar{y}_{30,\text{red}})^b$ $(\bar{x}_{i,15} \vee y_{24,\text{blue}} \vee y_{25,\text{blue}})^a$ $(\bar{x}_{i,15} \vee \bar{y}_{19,\text{blue}} \vee \bar{y}_{25,\text{blue}} \vee y_{24,\text{yellow}})^b$
6			$(\bar{x}_{i,15} \vee \bar{y}_{24,\text{yellow}} \vee y_{25,\text{red}})^b$ $(\bar{x}_{i,15} \vee \bar{y}_{25,\text{red}} \vee y_{24,\text{yellow}})^b$ $(\bar{x}_{i,15} \vee y_{19,\text{blue}} \vee y_{24,\text{blue}})^b$
7			$(\bar{x}_{i,15} \vee \bar{y}_{24,\text{green}} \vee y_{19,\text{blue}})^b$ $(\bar{x}_{i,15} \vee \bar{y}_{19,\text{blue}} \vee y_{25,\text{red}})^b$ $(\bar{x}_{i,15} \vee \bar{y}_{25,\text{red}} \vee y_{30,\text{yellow}})^b$ $(\bar{x}_{i,15} \vee \bar{y}_{30,\text{yellow}} \vee y_{24,\text{green}})^b$

<sup>a</sup> two clauses; apply permutation  $\{(y_{19,c}, y_{25,c}), (y_{25,c}, y_{30,c}), (y_{30,c}, y_{24,c}), (y_{24,c}, y_{19,c})\}$  to obtain the other one.

<sup>b</sup> four clauses; apply permutation  $\{(y_{19,c}, y_{25,c}), (y_{25,c}, y_{30,c}), (y_{30,c}, y_{24,c}), (y_{24,c}, y_{19,c})\}$  iteratively to obtain the other three.

**Fig. 3.** The translation of the seven types of center pieces to CNF. The most frequent occurring color is represented by blue, followed by red, yellow and green. Each arrow (implication) is encoded a (set of) clause(s). Black diamonds refer to the complement of an edge. The last column shows one clause per arrow for a piece  $p_i$  placed on  $q_{15}$  on a  $6 \times 6$  grid. The corresponding diamonds are  $d_{19}$  (north),  $d_{25}$  (east),  $d_{30}$  (south),  $d_{24}$  (west).

Although the translation is sufficient, it may not be optimal in case one wants to solve it with a SAT solver. With the addition of some (or even many) clauses and variables, some SAT solvers may find a solution much faster. This section discusses two extensions of the compact translation. Both represent additional knowledge about the problem and require only some extra clauses.

### 5.1 Forbidden Color Clauses

Once a diamond is given a certain color, then several pieces are not allowed to be placed on the corresponding squares. This knowledge can be added to the formula with several binary clauses. For each diamond, if assigned to a color, then all pieces without that color (on at least one of its edges) cannot be placed on one of the two corresponding squares.

**Example 3.** Given a piece  $p_C$  with one blue edge, two pink edges and a red edge. Say we want to place it on square  $q_{15}$  and the  $d_{30}$  is one of the corresponding diamonds and  $C_{\text{center}} = \{\text{blue, cyan, green, orange, pink, red}\}$ . The forbidden color clauses would be:

$$(\bar{x}_{C,15} \vee \bar{y}_{30,\text{cyan}}) \wedge (\bar{x}_{C,15} \vee \bar{y}_{30,\text{green}}) \wedge (\bar{x}_{C,15} \vee \bar{y}_{30,\text{orange}}) \quad (7)$$

Notice that, provided the encoding of corner and border pieces as described in Section 4.1, these clauses only make sense for center pieces. Let  $C(p_i)$  be the set of colors of piece  $p_i$  and  $q_k^<$  be the smallest index of the corresponding square of diamond  $d_k$ , and  $q_k^>$  the largest index of the corresponding square.

$$\bigwedge_{\text{color} \in C_{\text{center}} \setminus C(p_i)} (\bar{x}_{i,q_k^<} \vee \bar{y}_{k,\text{color}}) \wedge \bigwedge_{\text{color} \in C_{\text{center}} \setminus C(p_i)} (\bar{x}_{i,q_k^>} \vee \bar{y}_{k,\text{color}}) \text{ for } p_i \in P_{\text{center}}, d_k \in D_{\text{center}} \quad (8)$$

Several assignments that are implicitly violated by the compact translation, become explicitly violated by the forbidden color clauses. For instance, two pieces cannot be placed on neighbouring squares if they do not have at least one edge in common, because the diamond between these squares cannot be colored. In the compact translation, not all rotation clauses can be satisfied in that situation, although the SAT solver may not see it, yet. However, with the additional forbidden color clauses this directly results in a conflict.

The disadvantage of adding forbidden color clauses, as with all types of additional clauses, is that the encoding will require more resources. Especially when the number of center colors is large, the number of forbidden color clauses will be enormous.

### 5.2 Explicit One-on-One Mapping

Recall that diamonds are explicitly forced to have exactly one-color which in turn *implicitly* forces each piece on exactly one square. Optionally, we can extend

the translation by adding it *explicitly*. A straight-forward translation of this enforcement is:

$$(\bar{x}_{i,j} \vee \bar{x}_{i,l}) \quad \text{for } p_i \in P_{\text{corner}} \text{ and } q_j, q_l \in Q_{\text{corner}} \text{ and } j < l \quad (9)$$

$$(\bar{x}_{i,j} \vee \bar{x}_{i,l}) \quad \text{for } p_i \in P_{\text{border}} \text{ and } q_j, q_l \in Q_{\text{border}} \text{ and } j < l \quad (10)$$

$$(\bar{x}_{i,j} \vee \bar{x}_{i,l}) \quad \text{for } p_i \in P_{\text{center}} \text{ and } q_j, q_l \in Q_{\text{center}} \text{ and } j < l \quad (11)$$

Notice that the number of additional clauses by this extension is  $\mathcal{O}(|P_{\text{center}}|^3)$ . Recall that for unbounded edge-matching problems, all pieces are in  $P_{\text{center}}$ , so the addition is much cheaper for bounded problems. However, if the problem is large enough, say  $|P_{\text{center}}| > 40$ , the number of additional clauses will exceed the number of original clauses. Yet, one cannot conclude that this addition is counterproductive (in terms of solving speed).

## 6 Results

This section offers some results of the proposed translations of edge-matching problems to CNF on a test set of bounded unsigned edge-matching problems. Four instances arise from the clue puzzles by Tomy called `clueX`. Additionally, eight problems were generated with various sizes (**a**), number of border colors (**b**) and number of center colors (**c**) called `em-a-b-c`. The smaller five generated instances have relatively many colors yielding only few solutions, while the larger three instances have few colors and therefore many solutions. For each instance from the test set we constructed four different encodings:

- $\mathcal{F}_{\text{compact}}$ : The compact translation as described in Section 3 and 4;
- $\mathcal{F}_{\text{fbcolors}}$ : The forbidden color clauses (Section 5.1) added to  $\mathcal{F}_{\text{compact}}$ ;
- $\mathcal{F}_{\text{explicit}}$ : The explicit one-on-one mapping (Section 5.2) added to  $\mathcal{F}_{\text{compact}}$ ;
- $\mathcal{F}_{\text{all}}$ : All presented clauses, the union of  $\mathcal{F}_{\text{fbcolors}}$  and  $\mathcal{F}_{\text{explicit}}$ .

Table 1 offers several properties of the test set instances. Next to the names, the second column lists the size (rows  $\times$  columns) of the grid. Although, we explained the translations using square grids, they can be used for rectangular grids as well. The third column shows the number of colors in the format  $(|C_{\text{border}}|, |C_{\text{center}}|)$ . The fourth column shows the number of variables used for all encodings. The number of clauses of  $\mathcal{F}_{\text{compact}}$ , and the number of the additional knowledge clauses are listed in the last three columns.

Only two state-of-the-art SAT solvers are used for the experiments: `picosat` [1] and `ubcsat` [10]. The former is a complete solver – it can also prove that no solution exists – while the latter is a local search solver. Initially, more solvers were used, but the results of complete solvers were strongly related, as were those of various incomplete ones. Therefore, only the strongest solver (based on earlier experiments) of each category was selected. The `picosat` solver was faster than `minisat` [4], probably due to use of rapid restarts in the former. For the `ubcsat`

**Table 1.** Properties of the selected benchmarks. The number of variables is denoted by  $\#_{\text{variables}}$ . The last columns offer the size of the formulas expressed in the number of (additional) clauses.

name	size	colors	$\#_{\text{variables}}$	$ \mathcal{F}_{\text{compact}} $	$ \mathcal{F}_{\text{fbcolors}} $	$ \mathcal{F}_{\text{explicit}} $
clue1	$6 \times 6$	(4, 3)	728	3688	+704	+3864
clue2	$6 \times 12$	(4, 4)	2904	23570	+9120	+41808
clue3	$6 \times 6$	(5, 4)	788	3836	+1792	+3864
clue4	$6 \times 12$	(5, 4)	2936	23574	+9280	+41808
em-7-3-6	$7 \times 7$	(3, 6)	1473	11563	+7500	+11324
em-7-4-8	$7 \times 7$	(4, 8)	1617	14513	+11400	+11324
em-7-4-9	$7 \times 7$	(4, 9)	1677	15063	+13800	+11324
em-8-4-5	$8 \times 8$	(4, 5)	2420	22340	+11232	+29328
em-9-3-5	$9 \times 9$	(3, 5)	3857	39932	+19796	+68232
em-11-3-4	$11 \times 11$	(3, 4)	8713	99699	+29808	+285144
em-12-2-4	$12 \times 12$	(2, 4)	12584	155712	+47200	+526224
em-14-7-3	$14 \times 14$	(7, 3)	24356	305744	+41475	+1536792

solver, one can select from many different stochastic local search algorithms. From those algorithms `ddfw` [6] appeared to be the fastest one on the smaller instances of the test set. Therefore, this algorithm<sup>6</sup> was selected.

Each solver was run on each formula with ten seeds. The execution times differed significantly for those seeds, for both `picosat` and `ubcsat`. Therefore, the results show besides the average computational costs also the variance. In case at least five seeds took more than an hour,  $> 3600$  is listed.

The most striking result is that the complete solver `picosat` is unable to solve most benchmarks in the test set when the explicit one-on-one mapping clauses are not added, see Table 2. Although these clauses are redundant, they appear crucial to solve the problem. Only the four clue puzzles can be solved without these clauses, although `clue2` and `clue4` are solved considerably faster with them.

The results of the incomplete solver `ubcsat` shown in Table 3 are much more ambivalent. In contrast to `picosat`, the most elaborate translation hardly seems the optimal encoding. Yet, only `em-7-3-6` is solved fast using the compact encoding. The smaller generated instances (with relatively few solutions) are faster solved by adding the redundant forbidden color clauses, while for the larger ones (with many solutions) the explicit one-on-one mapping clauses appear useful. Apparently, redundant clauses can guide the search for incomplete SAT solvers as well.

Yet, despite the weakness shown on the translation without the additional clauses, the (complete) `picosat` appears the best overall choice. When the compact translation is extended with both sets of additional clauses, of `picosat` outperforms `ubcsat` on the harder instances. Moreover, the results suggest that extending the translation with even more additional knowledge could further improve the performance.

<sup>6</sup> Using the default settings with `runs = 1,000`, `cut-off = 1,000,000`

**Table 2.** Computational costs (in seconds) to solve the test set using picosat.

name	$\mathcal{F}_{\text{compact}}$		$\mathcal{F}_{\text{fbcolors}}$		$\mathcal{F}_{\text{explicit}}$		$\mathcal{F}_{\text{all}}$	
clue1	0.11	(0.02)	<b>0.10</b>	(0.00)	<b>0.10</b>	(0.00)	<b>0.10</b>	(0.00)
clue2	506.44	(364.19)	164.84	(49.87)	2.75	(1.68)	<b>0.95</b>	(0.37)
clue3	0.22	(0.12)	0.12	(0.04)	<b>0.10</b>	(0.00)	<b>0.10</b>	(0.00)
clue4	1527.54	(536.34)	269.77	(84.72)	1.90	(2.13)	<b>0.54</b>	(0.07)
em-7-3-6	> 3600	-	> 3600	-	140.00	(135.18)	<b>34.91</b>	(27.18)
em-7-4-8	> 3600	-	> 3600	-	1132.54	(1054.23)	<b>852.32</b>	(890.52)
em-7-4-9	> 3600	-	> 3600	-	45.94	(43.75)	<b>41.89</b>	(55.94)
em-8-4-5	> 3600	-	> 3600	-	209.35	(187.79)	<b>86.58</b>	(67.23)
em-9-3-5	> 3600	-	> 3600	-	501.81	(220.31)	<b>152.81</b>	(121.13)
em-11-3-4	> 3600	-	> 3600	-	163.48	(99.87)	<b>51.68</b>	(35.56)
em-12-2-4	> 3600	-	> 3600	-	249.66	(151.65)	<b>88.36</b>	(81.92)
em-14-7-3	> 3600	-	> 3600	-	80.24	(49.73)	<b>32.58</b>	(17.91)

**Table 3.** Computational costs (in seconds) to solve the test set using ubcsat.

name	$\mathcal{F}_{\text{compact}}$		$\mathcal{F}_{\text{fbcolors}}$		$\mathcal{F}_{\text{explicit}}$		$\mathcal{F}_{\text{all}}$	
clue1	0.04	(0.03)	0.04	(0.02)	<b>0.02</b>	(0.01)	<b>0.02</b>	(0.01)
clue2	1.78	(1.40)	1.37	(1.21)	<b>0.19</b>	(0.04)	0.21	(0.11)
clue3	0.06	(0.06)	0.08	(0.06)	<b>0.03</b>	(0.02)	0.04	(0.02)
clue4	1.38	(0.89)	1.69	(1.38)	<b>0.33</b>	(0.19)	0.38	(0.31)
em-7-3-6	<b>60.73</b>	(17.65)	138.23	(134.30)	670.65	(701.69)	225.70	(78.76)
em-7-4-8	2376.62	(2169.16)	<b>1732.65</b>	(1801.32)	> 3600	-	> 3600	-
em-7-4-9	1690.02	(1815.71)	<b>1284.47</b>	(1502.43)	> 3600	-	> 3600	-
em-8-4-5	155.62	(180.88)	<b>80.54</b>	(74.25)	381.91	(309.00)	128.88	(135.49)
em-9-3-5	1258.10	(1492.35)	<b>177.36</b>	(138.91)	1928.79	(2352.45)	839.40	(870.74)
em-11-3-4	82.73	(35.38)	34.16	(5.10)	<b>2.65</b>	(1.30)	3.78	(1.85)
em-12-2-4	154.99	(27.92)	32.52	(17.88)	<b>2.32</b>	(1.70)	4.41	(1.89)
em-14-7-3	145.72	(21.20)	48.97	(23.19)	<b>11.02</b>	(7.99)	44.72	(39.55)

## 7 Conclusions and Future Work

This paper presented a compact translation of edge-matching problems into CNF, as well as several extensions. The compact translation rarely resulted in the fastest performance, both for complete and incomplete SAT solvers. For complete solvers, the extensions even appeared crucial to solve the harder instances. Yet, these results are not very surprising and mostly show the extend of the importance of redundant clauses.

On the other hand, it is harder to explain why redundant clauses also guide the search for incomplete SAT solvers. Arguably, any performance gain due to adding redundant clauses could be interpreted as a flaw of the local search algorithm – redundant clauses only require additional resources. More research is needed to explain these results.

The focus of this paper, both the presentation and the experiments, is on bounded unsigned edge-matching problems. Translating unbounded and / or signed problems into CNF can be done in a similar manner. Future experiments will have to show whether SAT solvers can be used to solve these problems, such as Rubik’s Tangle, too.

Within the domain of edge-matching problems, there remains the enormous challenge of constructing a translation of the Eternity II puzzle that could be solved with a SAT solver. The proposed encoding is merely a first step in this direction. A possible next step is to determine which other knowledge could be added using redundant clauses.

Regarding the big picture, the challenge arises how to translate a problem into CNF in general. The presented results suggest that adding redundant clauses can significantly reduce the computational costs. Therefore, further research on the use of redundant clauses may provide insight in how to meet this challenge. Also, the results show that the optimal encoding will not only depend on properties of a given problem, but also on the preferred solver, since complete and incomplete solvers will require different translations.

## Acknowledgments

The author would like to thank Sean Weaver and the anonymous reviewers for their valueble comments.

## References

1. Armin Biere. PicoSAT Essentials. *Journal on Satisfiability, Boolean Modeling and Computation* **4** (2008), pp. 75–97.
2. Erik D. Demaine and Martin L. Demaine. Jigsaw Puzzles, Edge Matching, and Polyomino Packing: Connections and Complexity. Special issue on Computational Geometry and Graph Theory: The Akiyama-Chvatal Festschrift. *Graphs and Combinatorics* **23** (Supplement), June 2007, pages 195–208.

3. M. R. Dransfield, L. Liu, V. W. Marek and M. Truszczynski. Satisfiability and Computing van der Waerden Numbers. In *The Electronic Journal of Combinatorics*. Vol. **11** (1), (2004).
4. Niklas Eén and Niklas Sörensson, An extensible SAT-solver, In Giunchiglia and Tacchella (eds). *Theory and applications of satisfiability testing, 6th international conference, SAT 2003*. Santa Margherita Ligure, Italy, may 5-8, 2003 selected revised papers, *Lecture Notes in Computer Science* **2919** (2004), pp. 502–518.
5. Jacques Haubrich. *Compendium of Card Matching Puzzles*. Self-published, May 1995. Three volumes.
6. Abdelraouf Ishtaiwi, John Thornton, Abdul Sattar, Duc Nghia Pham: Neighbourhood Clause Weight Redistribution in Local Search for SAT. *CP 2005* (2005), pp. 772–776.
7. João P. Marques-Silva and Inês Lynce. Towards Robust CNF Encodings of Cardinality Constraints. *CP'07, Lecture Notes in Computer Science* **4741** (2007).
8. Rogier Poldner. *MINISAT: A semi SAT-based pseudo-Boolean solver*. MSc thesis TU Delft (2008).
9. Pierre Schaus and Yves Deville. Hybridization of CP and VLNS for Eternity II. *Journées Francophones de Programmation par Contraintes (JFPC'08)* (2008).
10. Dave A.D. Tompkins and Holger H. Hoos. UBCSAT : An implementation and experimentation environment for SLS algorithms for SAT and MAX-SAT. *Lecture Notes in Computer Science* **3542** (2005), pp. 306–320.