Fast Global Filtering for Eternity II

Eric Bourreau, Thierry Benoist

Fast Global Filtering for Eternity II™

Thierry Benoist TBENOIST@BOUYGUES.COM
e-lab - Bouygues SA
32 avenue Hoche
75008 Paris

Eric Bourreau ERIC.BOURREAU@LIRMM.FR
LIRMM
161 rue Ada
34392 Montpellier

Editor: Pascal Van Hentenryck

Abstract

In this paper we consider the enumeration of all solutions of a 2D edge-matching puzzle. We show that a judicious modeling of the problem, combined with the use of appropriate data structures allows obtaining an effective filtering algorithm with complexity *O(1)* at each node of tree search. Our experiments show the relevancy of the proposed power/complexity trade-off, compared to the results of some of the best available brute force tree search algorithms and to classical Constraint Programming models.
Keywords: Edge-matching puzzles, Alldifferent, Backjumping

1. Introduction

Recent months have seen an upsurge of interest in *Edge matching puzzles* since the release in July 2007 of the Eternity II game by TOMY, with a $2 million prize for the first submitted solution. Competition rules are summarized as follows on *http://us.eternityii.com*:

Make a square from all the pieces,
With the grey ones round the border;(…)
Match all touching pairs of edges
Do it first and win US$2 Million!

The simplicity of the rules (placing 256 square tiles on a 16x16 board such that all adjacent tiles have matching colors along their common edge) and the value of the prize, lead a certain number of researchers and puzzle fans to imagine numerous solving algorithms or identifying properties of this kind of problem. A yahoo forum has been created on this topic and gathers more than 2000 members (http://games.groups.yahoo.com/group/eternity_two/). In particular Demaine & Demaine (2007) established the NP-completeness of the problem and some of its variants. We also refer the reader to Owen (2007) for remarks on the design of hard instances.
In this paper, we focus on the enumeration of all solutions of a puzzle. Many competitors perform such enumerations on different instances in order to compare their backtracking algorithms. However the criteria usually put forward is the number of explored nodes per second. However it can be profitable in a tree search to spend more time on each node, if it allows decreasing the number of nodes to be explored. This is one of the founding principles of Constraint Programming (CP), whose application at each node of a search tree allows eliminating many branches (Montanari 1974, Mackworth 1988). Few CP based approaches have been described for edge matching puzzles, apart from Geoff Harris path-consistency based algorithms (Harris 2008) and the CP model introduced by Schaus and Deville (2008).
We confirm in this article that global filtering algorithms can yield a dramatic decrease of the number of nodes to be explored in the enumeration of all solutions. Besides, we show that an appropriate model for this problem, combined with the use of efficient algorithmic structures allows obtaining global and effective filtering with complexity *O(1)*. Compared to some of the best available backtracking algorithms, our approach explores a much smaller number of nodes in a competitive time.

2. The problem

In this section we define the problem and describe a first CP model.

2.1. Definition

The problem can be described as follows. With *N* and *K* two positive integers (the side of the board and the number[1] of colors) and *T* a collection of N^2 quadruplets of integers in interval $[0,K]$ (tiles as defined by their colors in the order *north*=0,*east*=1, *south*=2, *west*=3), is it possible to place all these tiles on a $N \times N$ board such that all adjacent tiles have matching colors along their common edge, with all edges of color « 0 » on the border of the board ? By convention the cell (0,0) will be the bottom left corner and the rotation is defined as the side of the tile oriented northwards.
More formally, a solution is an assignment to each pair $(i, j) \in [0,N\text{-}1]^2$, of a pair $(t(i,j), r(i,j)) \in T \times [0,3]$ describing the tile placed on cell *i,j* and the rotation applied to this tile, such that:
1. Each tile is used exactly once
$\forall (i_1, j_1, i_2, j_2) \in [0,N\text{-}1]^2, t(i_1, j_1) = t(i_2, j_2)$ if and only if $i_1 = i_2$ and $j_1 = j_2$
2. Two consecutive tiles on a column have identical color along their common edge
$\forall\, i \in [0,N\text{-}1]\ \forall\, j \in [0,N\text{-}2]$,
$t(i, j)[(0 + r(i,j)) \bmod 4] = t(i, j{+}1)[(2 + r(i,j{+}1)) \bmod 4]$
3. Two consecutive tiles on a row have identical color along their common edge
$\forall\, i \in [0,N\text{-}2]\ \forall\, j \in [0,N\text{-}1]$,
$t(i, j)[(1 + r(i,j)) \bmod 4] = t(i{+}1, j)[(3 + r(i,j{+}1)) \bmod 4]$
4. All border have color « 0 »
$\forall\, i \in [0,N\text{-}1]$,
$t(i, 0)[(2 + r(i,j)) \bmod 4] = 0,\ t(i, N\text{-}1)[(0 + r(i,j)) \bmod 4] = 0$,
$t(0, i)[(3 + r(i,j)) \bmod 4] = 0,\ t(N\text{-}1,i)[(1 + r(i,j)) \bmod 4] = 0$

2.2. Filtering algorithms

Constraint 1 is a classical matching constraint, whose consistency can be filtered (weakly) by a clique of binary constraints of the kind $t(i_1, j_1) \neq t(i_2, j_2)$. These constraints will basically infer that once placed on a cell, a tile cannot be used on any other cell. When instantiating a variable $t(i, j)$, at most $|T|$-1 value removal are performed. The *complete* consistency of this constraint can also be ensured with an *AllDifferent* constraint (Régin 1994). This global constraint is able to detect that a subset of *p* tiles can only be assigned to a subset of *p* cells (strongly connected components in the bipartite graph associated to the matching problem). The complexity of this constraint is $O(|T|^{2.5})$. In the remaining of this paper, we will refer to models with or without *AllDifferent* to denote these two filtering modes. In Section 6, we illustrate this additional filtering power.
Constraints 2,3 et 4 could be modeled “as is”, with equality and *Element* constraints ($v=L(i)$ with *i* and *v* two variables and L an array of integers). However, if such a model would correctly infer that a cell with a red edge must receive a tile with a red edge, it would not be able to detect that a cell with two red edges must receive a tile with two red edges. We will refer to *4-edge consistency* to denote the filtering algorithm removing from the domain of a variable $t(i,j)$ all tiles unable (in any rotation) to match the colors of the four edges of the cell, and eliminating all impossible rotations of the domain of $r(i,j)$ once variable $t(i,j)$ is instantiated[2]. The complexity of this filtering is smaller than $O(|T|)$ since it merely requires checking (in constant time) for each value of the domain of $t(i,j)$ that a consistent rotation exists. Note that this level of consistency could also be modeled with an extensional (table) constraint or with a specific variable representing the pair $t(i,j),r(i,j)$. Creating variables for

[1] If we take into account the « 0 » color on the border, we have K+1 colors.
[2] It seems clear that filtering the domain of $r(i,j)$ while $t(i,j)$ is not instantiated would be hardly useful , since it would amount to comparing possible rotations of very different tiles.

the color of each edge would be useful as well, and we will see in section 6.3 that it can significantly improve domain reductions.

3. Proposed Model

In the following sections we will prove that it is possible to implement *in constant time* a filtering algorithm very similar to *4-edges consistency + AllDifferent*. To achieve this goal, we focus on pairs of colors and related filtering rules in Section 3.1 and discuss the resulting model in Section 3.2

3.1. Color Pairs

For any color pair k_1, k_2 it can be interesting to count the number of tiles with colors k_1, k_2 in clockwise consecutive order. For instance in the case of Eternity II^{TM}, this number is smaller than 12 if $k_1=0$ or $k_2=0$ and smaller than 7 otherwise, namely for the 196 inner tiles. It shall also be noted that among the 17^2 possible combinations of the 17 inner colors, 20 appear on no tile. In fact for an instance with |T| tiles and K colors, the average number of matching tiles for each pair of colors is always smaller than $4|T| / K^2$, which is a relatively small number for difficult instances with homogenous color distribution. Indeed, with a small number of colors the number of solutions is likely to be huge: in the extreme “K=1”case, any assignment is a solution. On the contrary, with a large number of colors, we have very little choice: in the extreme case where each color appears only twice, the puzzle can be built with a greedy algorithm.

This remark suggests a model centered on color pairs, with K^2 *ColorPair*(k_1, k_2) structures. For each pair of colors k_1, k_2, *ColorPair*(k_1,k_2) contains two sets:

- *Offer*(k_1,k_2) $\subseteq T$ is the set of available tiles containing at least once the color sequence k_1, k_2 in clockwise order. The set of already placed tiles having this property will be denoted by *Unavailable*(k_1,k_2).
- *Demand*(k_1,k_2) $\subseteq [0,N\text{-}1]^2$ is the set of uninstantiated cells with two clockwise-consecutive edges of colors k_1 and k_2, respectively. In other words it means that tiles on two neighbor cells constrain this cell to receive a tile from set *Offer*(k_1,k_2). Naturally, having an edge on the border of the board is equivalent for a cell to having a neighbor cell presenting color 0. In the remaining of this paper we will consider the border of the board as « neighbors ».

For each assignment of a tile to a cell with a certain rotation, that is to say for each assignment $(t(i,j), r(i,j)) := (\tau, \rho)$, the *ColorPair* structures are maintained as follows. Tile τ is removed from the four *Offer*(k_1,k_2) sets it belongs to, and cell (i,j) is removed from the (at most) four *Demand*(k_1,k_2) sets it belongs to. Then all four neighbor cells are analyzed (but for the instantiated ones) in order to determine whether the placement of tile τ in (i,j) causes neighbor cells to integrate new *Demand*(k_1,k_2) sets. For instance if cell $(i+1,j+1)$ contains a tile τ' with rotation ρ', then if cell $(i+1,j)$ is empty it now requires the color sequence $\tau[(1+\rho) \bmod 4], \tau'[(2+\rho') \bmod 4]$ in clockwise order. Cell $(i+1,j)$ must then be added to the corresponding *Demand*(k_1,k_2) set, namely *Demand*($\tau[(1+\rho) \bmod 4], \tau'[(2+\rho') \bmod 4]$) as illustrated on figure 1.

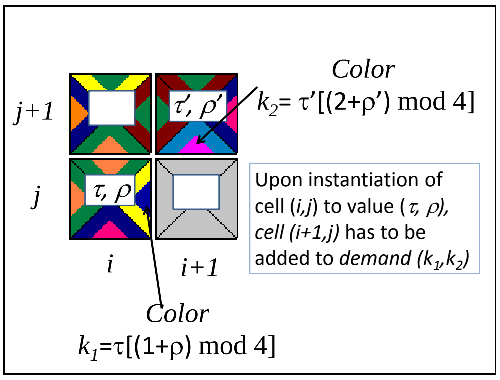

Color
$k_2 = \tau'[(2+\rho') \bmod 4]$
j+1
j
i
i+1
τ', ρ'
τ, ρ
Upon instantiation of cell (i,j) to value (τ, ρ), cell (i+1,j) has to be added to demand (k_1,k_2)
Color
$k_1=\tau[(1+\rho) \bmod 4]$

Figure 1

Maintaining these *ColorPair*(k_1,k_2) structures allows performing the following inferences:

- **Contradiction**. If $|Offer(k_1,k_2)| < |Demand(k_1,k_2)|$ then the number of cells requiring color sequence k_1,k_2 in clockwise order is larger than the number of remaining tiles offering this sequence. The problem is inconsistent.
- **Instantiation**. If $|Offer(k_1,k_2)| = |Demand(k_1,k_2)| = 1$ then only one tile τ can be placed on cell (i,j) requiring this color sequence. If this tile contains sequence k_1,k_2 only once, then there is a unique possible rotation ρ of tile τ.

3.2. Properties

First of all, the notion of color pair "in sequence" (in clockwise order) can be generalized to the case of the K(K-1)/2 pairs of opposite colors. All properties described in this paper on sequence pairs apply to opposite pairs as well. We have implemented both views, which means that each tile belongs to at most 6 *Offers* sets. This notion can also be "generalized" to single colors, counting the number of offered and demanded edges or each color, but knowing a single edge of a color yields little filtering (in Eternity II each color appear on around 40 tiles).

In terms of complexity, the algorithm (described in Section 3.1) maintaining these *ColorPair* structures requires for each assignment at most 6 tile removal from *Offer* sets and 6 cell removal from *Demand* sets, then, after analyzing the edges of uninstantiated neighbor cells, at most 4x3 cell additions in *Demand* sets. Storing these sets in data structures providing constant time addition and removal[3], yields to an *O(1)* filtering algorithm. The reverse operation (*unassign*) has the same complexity.

A specificity of this constraint-Programming model is that the domain of variables ((t(i,j), r(i,j)) for each cell (i,j)) is not maintained at all, for the sake of efficiency. What we maintain is the current state of the board, that is to say the values of already assigned couples (t(i,j), r(i,j)), together with K^2 structures *ColorPair*(k_1, k_2) required by our filtering algorithm. Now if we consider the resulting filtering on an empty cell, we note that the domain that would be inferred by the above-mentioned *4-edges consistency* is contained in the intersection of the required *Offers*(k_1,k_2). In particular, for cells with 2 neighbors, the domain computed by *4-edges consistency* is identical to the *Offer*(k_1,k_2) set corresponding to the pair of neighbor colors.

As for cells with 3 or 4 neighbors we will see in Section 4.2 that our search strategy branches on this kind of cells as soon as they appear. Then restricting their domain appears less crucial. This is the reason why we do not maintain the domain of variables themselves but rather these *ColorPair* structures (note that *Offer* sets are not domains since they end with value $\varnothing$ for solutions). Their number is similar (K^2 instead of |T|), but they present the advantage of having a smaller size and allow a direct triggering of the *contradiction* and *instantiation* rules.

Finally, these rules yield inferences that would not be detected by local consistency. Indeed, considering a single cell would only allow detecting the absence of available tiles

[3] We use the classical data structure in which each element maintains its own position in an array. The |X| first positions of this array contain the elements of set X. See *http://www.carva.org/thierry.benoist/eternity2.html* for an implementation of this structure.

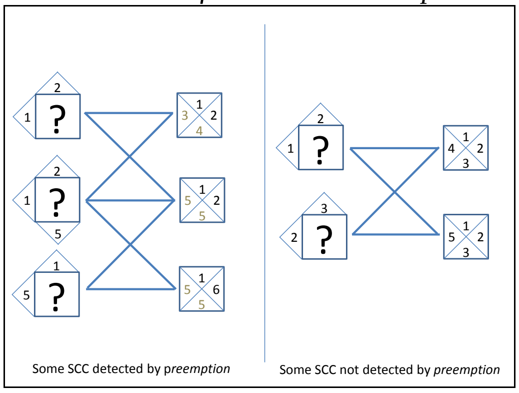
corresponding to the neighborhood of this cell or the uniqueness of such a tile but not global contradictions on the total number of cells demanding a certain type of tile. We will see in Section 6 that this simple model can decrease the number of nodes by 100 compared to brute force search.

4. Reinforced filtering

Our *ColorPair* based model can be enriched with *AllDifferent*-like inferences and with specific reasoning on remaining tiles.

4.1. Preemptions

If $|Offer(k_1,k_2)| = |Demand(k_1,k_2)|$ then all tiles of $Offer(k_1,k_2)$ will necessarily be used on cells of set $Demand(k_1,k_2)$. For instance in the first case of figure 2, we can assert that both tiles presenting sequence 1,2 will be placed on both cells demanding this sequence. We will say that these tiles are *preempted* by $ColorPair(k_1,k_2)$ and are thus removed from all other *Offer* sets they belong to, what may trigger new inferences: instantiations, contradictions or even new preemptions. It shall be noted that this reasoning only applies in the absence of cells with 3 or 4 instantiated neighbors, that is to say if $Demand(k_1,k_2)$ sets are pairwise disjoint. In the same case as figure 2, the existence of a cell with edges 5,1,2 prevents the removal of both tiles « 1,2 » from *Offer(5,1)* because such a removal would make |*Offer*(5,1)| strictly smaller than |*Demand(5,1)*|, thus triggering an undue contradiction. In practice, preempted tiles are not removed from *Offer* sets but merely tagged as *preempted* and an *nbPreempted* counter is maintained on each *Offer*. Hence we consider that the number of available tiles is either |*Offer*| or |*Offer*|-*nbPreempted* depending on the presence of cells with three neighbors or more. As announced above, we will see in Section 4.2 that our search strategy makes the latter case much more frequent (no cell with 3 or 4 neighbors).
This concept of preemption can be compared to the filtering principles of the *AllDifferent* constraint. A key feature in this constraint is the notion of Strongly Connected Component (SCC) which corresponds to a set of variables which cannot takes values outside of a set of values (see Regin, 1994 for details). Similarly, the preemption mechanism described above detects a set of tiles which must be assigned to a set of cells. Yet preemption filtering is not equivalent to the complete *AllDifferent*, as illustrated in the second case of figure 2. In this second example the *AllDifferent* constraint would detect that both tiles on the right side cannot be assigned to any other cells than those on the left side. On the contrary our preemption rules would not detect anything since |*Offer(1,2)*|=2 and |*Demand(1,2)*|=1, |*Offer(2,3)*|=2 and |*Demand(2,3)*|=1. In other words we only detect strongly connected components involving offers and demand *for the same color pair*.


Some SCC detected by *preemption*
Some SCC not detected by *preemption*
Figure 2

However, the complexity of preemption detection is much smaller than that of a “real” *AllDifferent*. These specific SCC are detected in $O(1)$, and tagging tiles as *preempted* is done in at most 6x|*Offer*| operations. Since each tile cannot be preempted by more than one *ColorPair*, the amortized complexity of this tagging is $O(|T|)$ for each branch of the search tree what amounts to an average complexity per node of $O(1)$.

We will see in Section 6 that on relatively small instances (7x7 to 10x10) a factor of 100 to 1000 is observed on the total time when comparing our efficient reinforced filtering (preemptions) to the standard modeling with *AllDifferent* in $O(|T|^{2.5})$.

4.2. Reasoning on remaining tiles

Upper bound on shared edges

It is possible to detect that a set of tiles cannot be correctly placed. For instance if there are no demanded red edge anymore whereas there are exactly two such edges left, on the *same* tile, then placing this tile will be impossible. More generally, if there are *m* edges left for a color *k* and only *m'* such edges are demanded, then *m-m' k*-colored edges will have to touch each others. Now [Benoist 2008] shows that the maximum number of edges that can be shared by *M* square tiles is equal to *h(M)* :

$$with\ s = \lfloor \sqrt{M} \rfloor, h(M) = 2s(s-1) + \begin{cases} 0 & if \quad M - s^2 = 0 \\ 2(M - s^2) - 1 & if \quad 0 < M - s^2 \leq s \\ 2(M - s^2) - 2 & if \quad s < M - s^2 \leq 2s \end{cases}$$

Hence if these *m* edges are dispatched on *m''* tiles, the inequality $m\text{-}m' \leq 2h(m'')$ must be satisfied, each shared edges “consuming” two *k*-colored edges. This property holds for any set of colors but can be checked in constant time if we consider singletons only, each assignment requiring the check of at most four colors.

Cycle detection

If *k* is a color of the border, and there are only two *k*-colored edges (on two tiles) to be placed and no cell demanding color *k*, then these two tiles will necessarily be placed next to each other. In such a situation (detected in constant time), similar links extending this chain can be searched easily: a chain of length *c* will be identified in *O*(*c*) and *c* cannot be larger than *K* since each link of the chain correspond to a color. If this chain is a cycle then the border cannot be completed anymore which means that the current assignment is inconsistent.

These two filtering rules have a small complexity and allow detecting inconsistencies on remaining tiles. Unfortunately, we will see in Section 6 that the obtained gain on the total number of explored nodes is around 2%.

5. Search strategy

Now that both our model and our filtering algorithms have been defined (sections 3 and 4), we describe in this section our tree search which is defined by our descent strategy (choice of the next cell to be instantiated) and our backtracking strategy (detection of branches whose exploration is useless). Both strategies decrease the total time by more than 25% (see Section 6.2).

5.1. Branching Strategy

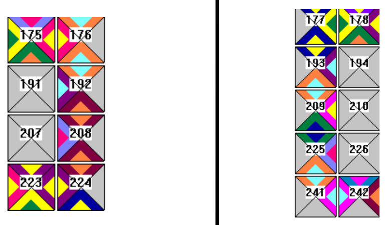
The choice of the next cell to be instantiated is based on two criteria. First of all we focus on highly constrained patterns. If not such pattern is found we select the most urgent *ColorPair* and pick one of its demanding cells.

Patterns

As evoked in Section 3.1, the large number of possible tiles for cells having zero or one neighbor suggests branching on cells with at least two neighbors. Such a strategy is possible since while the board is not covered with tiles there is necessarily at least one empty cell with two instantiated neighbors. For instance the empty cell with minimal coordinates (*i*,*j*) (in a lexicographic sense) has necessarily to its left and below, either a border or an instantiated cell, hence two edges with determined colors. These cells with at least two neighbors will be referred to as "*surrounded cells*".

We favor the most constrained cells which means, by decreasing priority:

1. Cells with four instantiated neighbors
2. Cells with three instantiated neighbors
3. Cells with two instantiated neighbors and one surrounded neighbor
4. Cells with one instantiated neighbor and two surrounded neighbors. In this latter case, we select one of the two surrounded neighbors.



Case 3
Cell n° 191 has two instantiated neighbors (n° 175 and n° 192) and one surrounded neighbor (n° 207).
Cell n° 207 is in the same situation.

Case 4
Cell n° 210 has two surrounded neighbors (n° 194 and n° 226).

Figure 3

Cases 1 and 2 correspond to highly constrained situations where the number of possible tiles is likely to be very small. It can even be null since we do not maintain local consistency around each cell: the intersection of *Offer*(k_1,k_2) sets attached to the cell can be empty (hence the importance of favoring these cells in our variable selection strategy). Favoring these variables follows a classical *MinDomain* principle. Case 3 correspond to the case of a cell whose instantiation will strongly constrain a neighbor cell. In the example considered in the above figure, placing a tile on cell n° 191will make cell n°207 a 3-neighbor cell : regardless of the size of the domain of cell n° 191, the number of tiles preserving the feasibility of cell n° 207 is probably very small if not null. Finally is the case 4 of this figure, instantiating cell n° 194 or n° 226 will lead to the situation of case 3. Detecting these patterns is done in constant time since it just requires maintaining the number of *instantiated* and *surrounded* neighbors for each cell. If such a pattern is detected, the possible values for the selected cell are computed from the intersection of the relevant *Offer* sets.

Heuristics for the choice of the most urgent pair

In the absence of such patterns, we will denote *MinOffer* the strategy consisting in selecting a *ColorPair*(k_1,k_2) with *Demand*(k_1,k_2)$\neq\emptyset$ and such that *Offer*(k_1,k_2) has minimal size (classical *MinDomain*). Then we randomly choose a cell in *Demand*(k_1,k_2). If we maintain, in constant time, *D* sets of *ColorPair* corresponding to the *D* possible cardinalities for *Offer* sets, then the selection of this *ColorPair* is done in *O(D)*. As pointed out in Section 3.1, *D* is usually small : smaller than 12 in Eternity II^{TM} (16x16 instance).

This selection strategy can be improved by simple statistic reasoning. For each pair *ColorPair*(k_1,k_2), the probability[4] for each tile of *Offer*(k_1,k_2) to be used on one of the cells of *Demand*(k_1,k_2) is *a priori* equal to *Demand*(k_1,k_2)/ *Offer*(k_1,k_2). Logically this probability is 0 if pair k_1,k_2 is not demanded by any cell and 1 if the number of demands equals the number of offers for this pair, namely in a *preemption* case (described in Section 4.1). This estimation can be refined taking into account the concurrent demands for each tile. For a tile τ, let *pairs*(τ) be the set of all pairs *p* such that $\tau\in offer_p$. Denoting by $offer_p$ and $demand_p$ the two

[4] The term « probability » is improper for these heuristic estimators, but was kept for the sake of readibility.

sets attached to a pair *p*, we can estimate the probability for tile τ to be used on one of the cells of *demand*$_p$ by the following formula.

$$P(\tau \to p_0) = \prod_{\substack{p \in pairs(\tau) \\ p \neq p_0}} \left(1 - \frac{|demand_p|}{|offer_p|}\right)$$

Indeed, being available for p_0 requires not to be used by any other pair. We call *expected offer,* and denote by E(|*offer*$_p$|) the sum of these probabilities over all tiles of *offer$_p$*.

$$E(|offer_p|) = \sum_{\tau \in offer_p} P(\tau \to p)$$

Using this expected value we can define a criterion for the choice of the most urgent color pair.

- *MinExpectedOffer* consists in selecting the pair *p* minimizing E(|*offer*$_p$|),
- *MinExpectedGap* consists in minimizing E(|*offer*$_p$|) - |*demand*$_p$|, in order to focus on pairs likely to trigger a contradiction (*first fail principle*),
- *MinExpectedRatio* consists in minimizing the ratio E(|*offer*$_p$|) / |*demand*$_p$|, pursuing the same goal as the above criterion,
- *MaxExpectedFiltering* consists in minimizing E(|*offer*$_p$|) - |*offer*$_p$| (always negative). The smaller this value, the greater the consequences on other pairs of the instantiation of one of the corresponding cells.

We will see in Section 6.2 that our best results have been obtained with *MinExpectedGap.* In any case, once the *ColorPair* is selected, the cell to be instantiated is randomly chosen among demands for this pair. The list of branches issued from a node in the search tree is a list of tile/rotation pairs (in T × [0,3]) consistent with the current colors of the four edges of the cell.
The value selection heuristic is of no importance in this paper since we consider complete search (enumeration of all solutions). Yet sorting candidate tiles by decreasing probability $P(\tau \to p_0)$ would be an natural criterion, and improving this selection in order to get feasible solutions as soon as possible is still an interesting open question for these puzzle matching problems.

5.2. Backtracking Strategy

Example

Upon failure during tree search, rather than backtracking to the above node, it can be preferable to detect the cause of the failure so as to backtrack higher in the tree, to a variable involved in the conflict, thus avoiding the exploration of worthless subtrees.
In the example below, testing each of the two remaining tiles (in *Offers(1,2)*) for the cell tagged "?" we observe that both lead to failure. Now what if previous nodes involve cells that are not represented in this figure ? Revising these cells will not fix the problem detected there and time will be wasted exploring vainly a potentially large number of nodes. On the contrary, if we manage to detect that the conflict will remain while all cells represented in this picture keep their current colors, then we can perform an "intelligent" backtrack or *conflict-based backjumping* (Gaschnig 1979, Ginsberg 1993), backtracking in the tree until a node involving one of these cells is found.

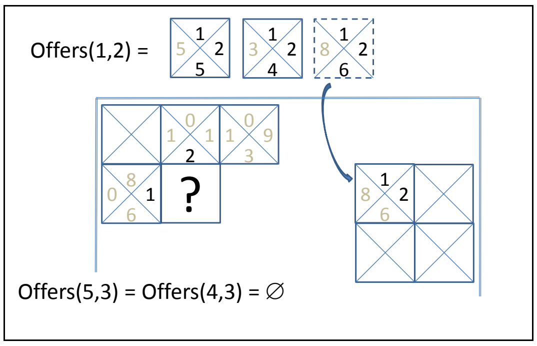


Figure 4

Computing explanations

When detecting an inconsistency, we need to determine its cause, namely a set of cells (called *explanation*) whose current values are sufficient to explain the current contradiction. In the search tree, when all branches issued from a node have been explored, the explanation for the failure of this node is merely the union of the explanation of child nodes.
In our model, failures are detected in only two ways:

- Either discarding some tile/rotation pairs for a cell because of mismatching edges. It is the case of cells with 3 or 4 neighbors described in Section 4.2, for which the explanation is the set of the 3 or 4 instantiated neighbors.
- Either with the rule $|Offer(k_1,k_2)| < |Demand(k_1,k_2)|$ defined at paragraph 3.1. In this case, the explanation is made of:
    - For each cell of $Demand(k_1,k_2)$, both neighbors presenting colors k_1 et k_2;
    - For each tile of $Unavailable(k_1,k_2)$, the cell this tile was assigned to.

These explanations are valid since uninstantiating all cells except those of the explanation cannot resolve the conflict. They are also minimal for the inclusion relation that is to say that there are cases when uninstantiating a single cell of the explanation would resolve the conflict.
Concerning tiles τ in $Unavailable(k_1,k_2)$, it shall be noted that if we have instantiated neighbors on both sides k_1 and k_2, then replacing in the explanation the cell containing τ by these two neighbor cells is valid. Besides, if these tiles were placed before τ, then this new explanation is better because it involves “older” cells and thus will allow backjumping higher in the tree.

As for tiles preempted by a *ColorPair* (see Section 4), the explanation of their unavailability is built as in the case of contradictions of kind $|Offer(k_1,k_2)| < |Demand(k_1,k_2)|$. This explanation is recorded upon preemption detection and reused later, each time the preemption of one of these tiles need to be explained.

The complexity of the computation of an explanation is determined by the costlier operation of the computation which is the scan of both sets $Demand(k_1,k_2)$ and $Unavailable(k_1,k_2)$, namely $O(D)$ (with D the size of the largest *Offer* set). As for the union of two explanations its theoretical complexity is $O(|T|)$ but an bitvector implementation yields a satisfying practical efficiency.

6. Experiments

We compare the different versions of our algorithm (implemented in Java) with one of the best available backtrackers, posted by *doc_smith*[5] on the yahoo forum dedicated to Eternity II^{TM}. This tree search, implemented in C, is able to explore millions of nodes per second. Adapted data structures allow a quick computation of the set of possible tiles for a cell that is

[5] http://games.groups.yahoo.com/group/eternity_two/files/doc_smith

to say the set of possible children of a node. However no global reasoning is implemented for branches elimination. We also report the results kindly sent by *mgaillard* for this benchmark.
We generated 140 puzzles for these experiments, from size 7x7 to 10x10. For a board of side N and a maximum number of colors K, we randomly assign a color to each of the 2N(N-1) edges of the board (uniform choice in [1,K]). Resulting tiles are randomly scrambled and rotated. Applying the backtrackers and our algorithms to each instances, we kept the 48 ones for which at least one approach succeeded in enumerating all solutions in less than 6 hours. These 48 instances can be downloaded, together with detailed results of our experiments, from *http://www.carva.org/thierry.benoist/eternity2.html*. All results have been obtained on the same 3GHz PC, except for the *mgaillard* backtracker (implemented in C and tested on a 2.2 GHz laptop). All instances but one have less than 8 solutions and 60% have only one solution.
Our goal is to compare the time and number of nodes needed by all these approaches to enumerate all solutions, independently of the heuristics or value ordering strategies embedded in these algorithms. Therefore only full tree search are considered in what follows.

6.1. Main findings

The following curves compare our four main approaches, with backtrackers identified by the name of their author, "*CP O(1)*" referring to the dedicated algorithm described in this paper (whose variants are evaluated in next Section) and *choco* denoting our experiments with a CP solver (Laburthe et al. 2000). Figure 5 represents on a logarithmic scale the number of nodes explored by each approach, with instances sorted by increasing best time on X-axis (excluding the 15 instances that either *doc_smith* or *choco* could not solve in 6 hours).

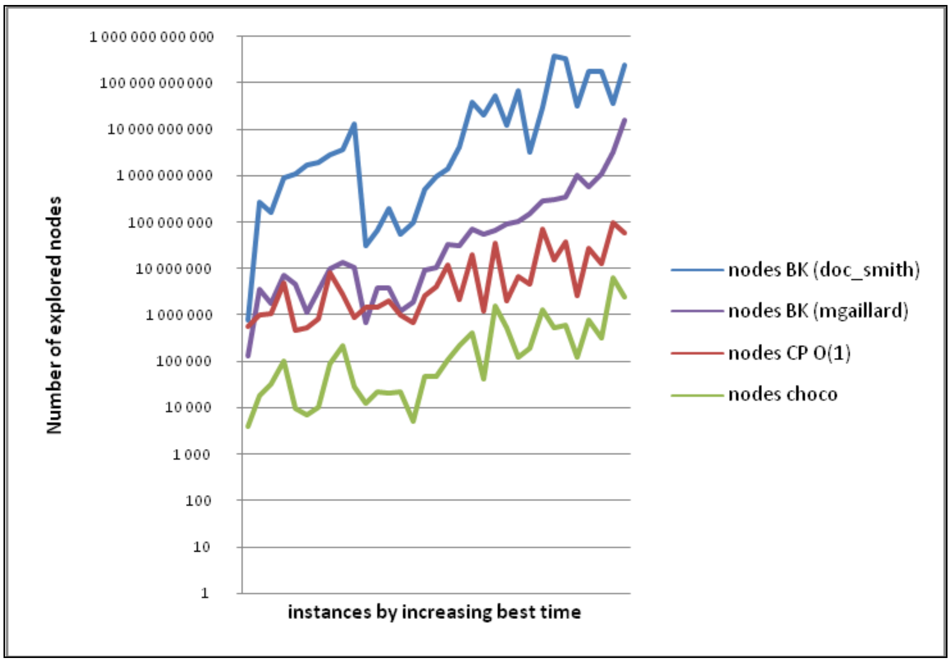


Figure 5

The effectiveness of constraint propagation appears clearly on these curves. Both CP based approaches explore much less nodes. As for the differences between both backtrackers they suggest that some kind of inference is implemented in *mgaillard*.

Since the number of nodes per second is roughly constant for each approach (100 for *choco*, 100 000 for *CP O(1)*, and more than 20 millions for both backtrackers), the result of time comparison is straightforward: the filtering efforts eventually pays off for large instances. Figure 6 compares on a logarithmic scale the time taken by *CP O(1)* and *mgaillard* (the only two approaches able to solve each instance within 6 hours) to complete their tree search. Not surprisingly, CP is competitive for the largest instances of this benchmark, and the tendency seems to indicate that CP would perform better than pure backtrackers on more difficult instances.

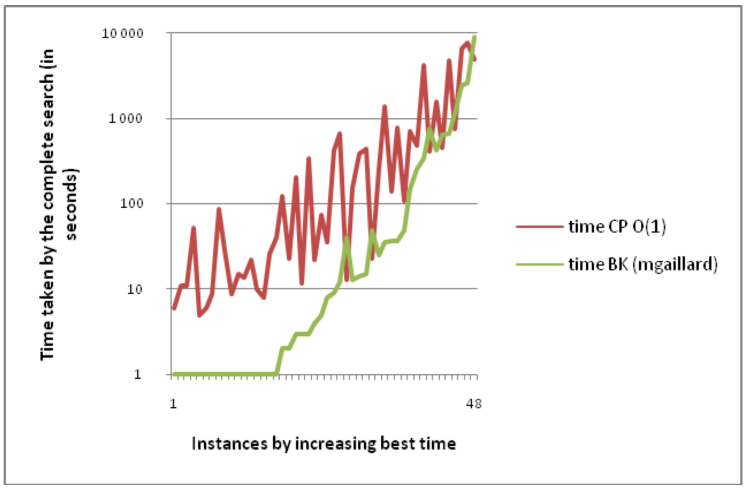


Figure 6

6.2. Key components

It can be interesting to evaluate the relative impact of the filtering algorithms and strategies exposed in this paper. Detecting instantiation and contradictions with *ColorPairs* is the main difference between our approach and pure backtrackers. Yet preemptions detection, shaving, backjumping and variable selection strategies also play a significant role.

Reinforced filtering introduced in Section 4 is the least effective component. Indeed even if ignoring preemptions increases the number of explored nodes by 24% it does not significantly modifies the total time. As for rules on remaining tiles (shared edges count and cycle detection) they only yield a decrease of 2% in terms of time and nodes.

On the contrary disabling backjumping increases the number of nodes by 50% and the total time by 30%. Shaving (Torres & Lopez 2000) also has a key impact as illustrated in the following curve (Figure 7) where total time and number of nodes (over all instances) are represented for different values of our shaving limit. Shaving with limit *L* means that for each cell of domain smaller or equal to *L*, we try all possible values in the domain thus inferring contradictions (if no value can be instantiated without triggering a contradiction) or instantiations (if only one value can be instantiated without triggering a contradiction). In figure 3, “-“ codes for no shaving and “all” codes for L=|T|. We see that shaving can decrease the total time by a factor of 5 or 6.

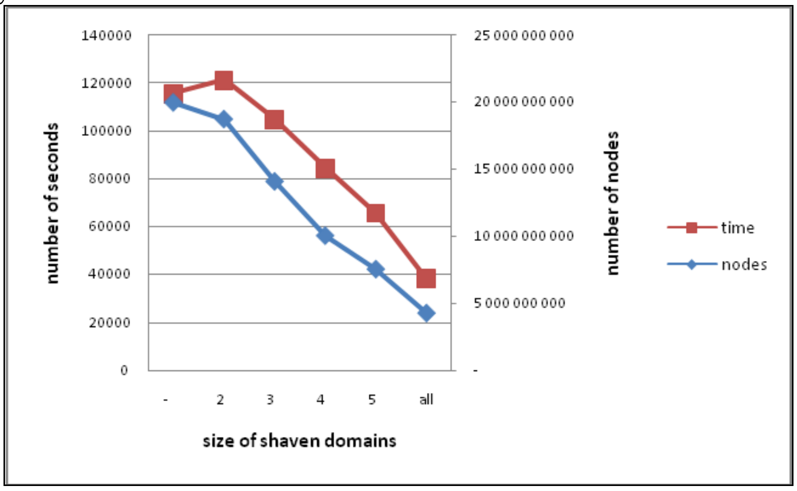


Figure 7. Impact of shaving (sum over all instances)

Finally, variable selection heuristics play an important role. In the table below, we present the gain in time and nodes obtained by the different heuristics described in Section 4.2 against the simple *MinOffer* (or "MinDomain") strategy, ommitting *MaxExpectedFiltering* whose results turned out to be worse than *MinOffer*.

MinExpectedOffer	*time gain*	16,25%
	node gain	22,75%
MinExpectedGap	***time gain***	**30,50%**
	node gain	**38,00%**
MinExpectedRatio	*time gain*	23,50%
	node gain	33,00%

Table1. Impact of variable selection heuristics (sum over all instances)

6.3. Comparison with Choco

In our introduction to Section 3, we claimed that our *constant time* filtering algorithms would be very similar to *4-edges consistency + allDiff*. This can be checked by implementing 4-edge consistency in Choco, combined with the standard complete *AllDifferent* constraint. The results below, on the 10 instances that this choco model could solve in 6 hours, show that the completeness of the *AllDifferent* constraint decreases the number of nodes by 27% in average compared to our preemption rules which only detected some of the strongly connected components (see Section 4.1). Note that even if this basic version of our CP algorithm involves no additional filtering it can sometimes explore a smaller number of nodes because differences in domain reductions will result in differences in variable selection (*minDomain* and simultaneous assignment of tile and rotation for both approaches), hence the search space will be explored differently.

instance	*time BasicCP O(1) (seconds)*	*nodes BasicCP O(1)*	*time choco 4edges (seconds)*	*nodes choco 4edges*	*time choco edge filtering (seconds)*	*nodes choco edge filtering*
E_10_16.b13i13.nohint	12	2 149 082	15 595	4 974 961	303	9 814
E_10_17.b13i13.nohint	24	4 408 684	6 348	1 932 224	239	7 186
E_10_5.b14i14.nohint	68	13 388 596	6 044	1 646 478	440	12 429
E_10_7.b12i15.nohint	34	5 521 750	4 170	1 151 440	159	3 999
E_10_8.b13i14.nohint	96	19 494 856	17 965	5 104 015	785	22 527
E_7_1.b6i6.nohint	68	13 285 760	12 865	20 366 984	468	193 257
E_7_2.b6i6.nohint	38	7 808 848	4 517	8 837 142	319	128 327
E_9_13.b10i10.nohint	32	6 242 786	18 172	11 054 431	521	41 849
E_9_3.b11i12.nohint	38	6 865 888	8 627	3 908 993	428	22 942
E_9_4.b9i12.nohint	26	4 858 100	5 534	2 662 968	79	5 017

Table 2. Constraint Programming results

This comparison also shows that what causes the very small number of nodes of Choco in Section 6.1 is not the complete *AllDifferent* but the rich filtering on the color of edges (*edge filtering* column). Indeed, from the set of possible tiles for a cell we can infer that some colors are not possible for some edges. The obtained results show that these inferences on the domain of "edge variables" trigger useful propagations and reduce the size of the search tree.

7. Conclusion and future work

These results show that, for the enumeration of solutions of 2D edge-matching puzzles, the filtering algorithms proposed in this paper (*ColorPairs* + *preemptions*) combined to an

effective heuristic (*expected gap*) and to a well known intelligent backtracking strategy (*backjumping*), yield a dramatic decrease of the number of nodes to be explored compared to brute force tree search. The constant time complexity of this filtering ensures that this approach becomes faster than brute force enumeration when instances are difficult enough.
A possible way for attacking the Eternity II puzzle consist in applying local search methods to a optimization variant of the puzzle. For instance Shaus & Deville (2008) consider fully instantiated boards and minimize the number of mismatching edges. Alternatively, we can consider partially instantiated boards and maximize the number of tiles positioned without mismatching edge. In the latter model, we have implemented a large neighborhood search using our CP-based tree search for each move. A possible weakness of this kind of approach is that the distance from a partial solution (with a very small number of mismatching edges or a small number of unassigned tiles) to a full solution may be very large in the considered neighborhood. That is to say that placing the last tile is likely to require a very large rearrangement of the board.

Our future works include precomputing endgames, namely all the feasible corner triangles with some fixed side *s*, and storing no-goods. A nogood would be a set of tiles and a list of colors. Once the extensions of a top left square of tiles have been explored, any further arrangement of the same tiles, leading to the same “frontier” of colors can be discarded as no-good.

Beyond these puzzles, this work contradicts the common belief that Constraint Programming pays the power of its inferences by a high complexity at each node, the former not always compensating for the latter. Recent works have shown that a slightly weaker filtering algorithm with a dramatic decrease of complexity can be a good compromise for some problems (Lopez-Ortiz et al. 2003). In this paper we have shown that exploiting properties of the problem and using efficient and dedicated data structures sometimes allows performing global and powerful filtering in a reasonable time if not in constant time.

References

T. Benoist. How many edges can be shared by N square tiles on a board? *E-lab research report*, 2008.

E.D. Demaine, M. L. Demaine. Jigsaw Puzzles, Edge Matching, and Polyomino Packing: Connections and Complexity. *Graphs and Combinatorics* vol 23, pp 195-208, Springer Japan 2007.

J. Gaschnig. Performance measurement and analysis of certain search algorithms. *Technical report* CMU-CS-79-124, Carnegie-Mellon University,1979 .

M.L. Ginsberg. Dynamic backtracking. *Journal of AI Research*, 1:25—46, 1993.

G. Harris. A CSP View of Eternity 2. *On* http://games.groups.yahoo.com/group/eternity_two/files/Beginner%20Benchmark%20Puzzles/ , 2008.

F. Laburthe. CHOCO: implementing a CP kernel. *In CP00 Post Conference Workshop on Techniques for Implementing Constraint programming Systems (TRICS)*, Singapore (2000)

A.K. Mackworth. Constraint Satisfaction, in *Encyclopedia of AI*, pages 205-211. Springer Verlag, 1988.

U. Montanari. Networks of constraints:Fundamental properties and application to picture processing. *Information Science, 7,* 1974

A. Lopez-Ortiz , C-G. Quimper, J.Tromp and P. Van Beek. A Fast and Simple Algorithm for Bounds Consistency of the Alldifferent Constraint. *In Proceedings of IJCAI'2003*.

B. Owen. *http://eternityii.mrowen.net/design.html*, 2007.

J-C. Régin. A filtering algorithm for constraints of difference in CSPs. *In AAAI 94, Twelth National Conference on Artificial Intelligence*, pages 362-367, Seattle, Washington, 1994.

P. Schaus , Y. Deville. Hybridation de la programmation par contraintes et d'un voisinage à très grande taille pour Eternity II. *In Dans JFPC 2008- Quatrièmes Journées Francophones de Programmation par Contraintes (2008) 115-122*

P. Torres , P. Lopez. Overview and possible extensions of shaving techniques for job-shop problems. *In 2nd International Workshop on Integration of AI and OR techniques in Constraint Programming for Combinatorial Optimization Problems (CP-AI-OR'2000).* pp181–186, 2000.

Y. Takenaga,T. Walsh . TETRAVEX is NP-complete. *Information Processing Letters, 99 (5),* Elsevier, 2006.