La programmation par contraintes à l'attaque d'Eternity II

Thierry Benoist, Eric Bourreau

La Programmation Par Contraintes à l'attaque d'Eternity II™

Thierry Benoist[1], Eric Bourreau[2]

[1]e-lab - Bouygues SA – 32 avenue Hoche 75008 Paris
[2] LIRMM - 161 rue Ada -34392 Montpellier
tbenoist@bouygues.com, eric.bourreau@lirmm.fr

Résumé

Nous nous intéressons dans cet article à l'énumération de toutes les solutions d'un puzzle de type edge-matching. Nous montrons qu'une modélisation adaptée du problème combinée à l'utilisation de structures algorithmiques efficaces permet d'obtenir un filtrage efficace et global, de complexité *O(1)*. Nous vérifions expérimentalement la pertinence du compromis filtrage/complexité proposé par comparaison avec un des meilleurs algorithmes arborescents disponibles.

Abstract

In this paper we consider the enumeration of all solutions of an 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)*. Our experiments show the relevancy of the proposed power/complexity trade-off, compared to the results of one of the best available tree search algorithms.

1. Introduction

Les puzzles de type *edge-matching* ont connu un regain d'intérêt depuis l'été 2007 et le lancement par la société TOMY du jeu Eternity II, doté d'un prix de deux millions de dollars. La simplicité de l'énoncé (placer 256 pièces à 4 côtés sur une grille 16x16 de façon à ce que toutes les arêtes adjacentes aient la même couleur) et l'importance du prix ont amené un certain nombre de chercheurs et d'amateurs à imaginer divers algorithmes de résolution ou à identifier des propriétés remarquables de ce type de problèmes (voir les plus de 5500 messages sur le forum de plus de 2000 membres dédié à ce sujet à l'adresse http://games.groups.yahoo.com/group/eternity_two/). En particulier (Demaine & Demaine 2007) établissent la NP-complétude du problème et d'un certain nombre de variantes.

Nous nous intéressons ici à l'énumération de toutes les solutions d'un puzzle. De nombreux compétiteurs comparent ainsi leurs algorithmes énumératifs sur différentes instances. Néanmoins le critère le plus souvent mis en avant est le nombre de nœuds explorés par seconde. Pourtant il peut être utile dans une recherche arborescente de consacrer un peu plus de temps à chaque nœud si cela permet d'en explorer beaucoup moins. C'est le principe de la Programmation Par Contraintes (PPC), qui permet à chaque nœud d'éliminer un maximum de branches de l'arbre de recherche (Montanari 1974, Mackworth 1988). Peu d'approches exploitant les techniques de la PPC ont été décrites pour ce problème, à l'exception des algorithmes de Geoff Harris à base de *path-consistency*.

Nous vérifions dans cet article que des algorithmes de filtrage globaux permettent de diminuer grandement le nombre de nœuds à explorer pour énumérer toutes les solutions. Nous montrons surtout qu'une modélisation adaptée du problème combinée à l'utilisation de structures algorithmiques efficaces permet d'obtenir un filtrage efficace et global, de complexité *O(1)*. Comparé à l'un des meilleurs algorithmes arborescent disponibles, nous obtenons une diminution significative du temps d'exploration complète, qui croît avec la complexité du problème.

2. Le problème

2.1. Définition

Le problème peut se résumer de la manière suivante. Etant donnés deux entiers N et K (le côté du plateau et le nombre de couleurs) et une collection T de N^2 quadruplets d'entiers dans l'intervalle $[0,K]$ (les pièces définies par leurs couleurs dans l'ordre *top*=0, *right*=1, *bottom*=2, *left*=3), peut-on placer toutes ces pièces dans une grille $N \times N$ de manière à ce que toutes les pièces adjacentes aient des couleurs identiques sur leur côté commun et que tous les côtés de couleur « 0 » se trouvent contre le bord du plateau ?

En d'autres termes il s'agit d'affecter à chaque couple $(i, j) \in [0,N\text{-}1]^2$, un couple $(t(i,j), r(i,j)) \in T \times [0,3]$ décrivant la pièce positionnée sur la case *i,j* et la rotation appliquée à cette pièce.

Les contraintes à respecter sont les suivantes :

1. Chaque pièce est utilisée une et une seule fois
   $\forall (i_1, j_1, i_2, j_2) \in [0,N\text{-}1]^2$, $t(i_1, j_1) = t(i_2, j_2)$ si et seulement si $i_1 = i_2$ et $j_1 = j_2$
2. Deux pièces consécutives sur une colonne ont des couleurs identiques sur leur arête commune
   $\forall i \in [0,N\text{-}1] \; \forall j \in [0,N\text{-}2]$,
   $t(i, j)[0 + r(i,j) \bmod 4] = t(i, j+1)[2 + r(i,j+1) \bmod 4]$
3. Deux pièces consécutives sur une ligne ont des couleurs identiques sur leur arête commune
   $\forall i \in [0,N\text{-}2] \; \forall j \in [0,N\text{-}1]$,
   $t(i, j)[1 + r(i,j) \bmod 4] = t(i+1, j)[3 + r(i,j+1) \bmod 4]$
4. Tous les bords ont la couleur « 0 »
   $\forall i \in [0,N\text{-}1]$,
   $t(i, 0)[2 + r(i,j) \bmod 4] = t(i, N\text{-}1)[0 + r(i,j) \bmod 4] = 0$
   et
   $t(0, i)[3 + r(i,j) \bmod 4] = t(N\text{-}1, i)[1 + r(i,j) \bmod 4] = 0$

2.2. Algorithmes de filtrage

La contrainte 1 est une contrainte d'affectation classique, dont la consistance peut être assurée de manière faible par une clique de contraintes binaires de type $t(i_1, j_1) \neq t(i_2, j_2)$, qui propagera essentiellement le fait qu'une fois placée sur une case une pièce ne peut plus être utilisée ailleurs sur le plateau. Lors de l'instanciation d'une variable $t(i, j)$, on effectue donc au pire cas T-1 retraits de valeurs. La consistance de cette contrainte peut également être assurée de manière complète par une contrainte *AllDifferent* [Régin 1994] qui sera capable de détecter des ensembles de p pièces qui ne peuvent être placées que sur un ensemble de p cases (composantes fortement connexes du graphe de couplage biparti). La complexité de cette contrainte est $O(n^{2.5})$. Dans le reste de l'article nous parlerons de modèle sans ou avec *AllDifferent* pour désigner ces deux mode de filtrage. En section 6, nous illustrerons cette puissance supplémentaire de filtrage sur une exploration totale de l'espace de recherche.

Les contraintes 2,3 et 4 pourraient être modélisées telles quelles, avec des contraintes d'égalité et des contraintes *Element* ($v=L(i)$ avec i et v des variables et L un tableau d'entiers de taille égale au domaine de i). Néanmoins un tel modèle impose bien qu'une case ayant une arête rouge doit recevoir une pièce ayant une arête rouge mais ne détecte pas qu'une case ayant deux arête rouges doit recevoir une pièce ayant deux arêtes rouge. Nous parlerons de *4-edges consistency* pour désigner l'algorithme de filtrage éliminant du domaine d'une variable *t(i,j)* toutes les pièces dont aucune rotation ne permet de respecter les couleurs des arêtes voisines, et éliminant les rotations impossibles du domaines de *r(i,j)* une fois que la variables *t(i,j)* est instanciée[1]. La complexité de ce filtrage est inférieure à $O(N^2)$ puisqu'il suffit de vérifier (en temps constant) pour chaque valeur du domaine de *t(i,j)* s'il existe une rotation compatible.

3. Modèle proposé

Dans cette section et les suivantes, nous montrerons qu'il est possible d'effectuer en *temps constant* un filtrage très similaire à *4-edges consistency* + *allDiff*.

3.1. Modélisation basée sur les paires de couleurs

Pour une paire de couleurs k_1, k_2 il est intéressant de dénombrer le nombre de pièces sur lesquelles apparaissent les couleurs k_1, k_2, l'une après l'autre dans le sens des aiguilles d'une montre. Par exemple sur le problème Eternity II™, ce nombre est inférieur à 12 si $k_1=0$ ou $k_2=0$ et inférieur à 7 sinon, c'est-à-dire pour les 196 cases centrales. On remarque même que sur les 17^2 combinaisons possibles des 17 couleurs centrales, 20 n'existent sur aucune pièce. Cette propriété est vérifiée dès que K^2 est suffisamment grand par rapport à $4N^2$, ce qui est le cas pour la plupart des instances difficiles. En effet si le nombre de couleurs est petit alors il risque d'y avoir un grand nombre de solutions : dans le cas extrême K=1, tout affectation est solution. Au contraire si le nombre de couleurs est très grand il y a peu de choix : dans le cas extrême où chaque couleur n'apparaît que deux fois, l'assemblage peut se faire de manière gloutonne.

Cette remarque amène à construire un modèle centré sur ces paires de couleurs. Notons ici qu'une particularité de ce modèle est que si les variables sont bien les couples $(t(i,j), r(i,j))$ pour chaque case (i,j), le domaine de ces variables n'est pas maintenu du tout, dans un souci d'efficacité. En contrepartie, nous maintenons d'une part l'état courant du plateau c'est-à-dire les valeurs des couples $(t(i,j), r(i,j))$ déjà instanciés et d'autre part K^2 structures $ColorPair(k_1, k_2)$ sur lesquelles se base notre algorithme de filtrage.

Pour chaque paire de couleurs k_1, k_2, $ColorPair(k_1,k_2)$ contient deux ensembles.

- $Offer(k_1,k_2) \subseteq T$ est l'ensemble des pièces disponibles contenant au moins une fois la séquence de couleur k_1, k_2 dans le sens des aiguilles d'une montre. Par convention nous noterons $Unavailable(k_1,k_2)$ l'ensemble des

[1] Il paraît clair que filtrer le domaine de *r(i,j)* tant que *t(i,j)* n'est pas instancié présenterait peu d'intérêt, puisque cela reviendrait à comparer les rotations possibles de pièces très différentes.

pièces ayant cette propriété mais déjà placées sur le plateau.

- *Demand*(k_1,k_2) $\subseteq [0,N\text{-}1]^2$ est l'ensemble des cases non instanciées dont deux arêtes consécutives dans le sens des aiguilles d'une montre présentent respectivement les couleurs k_1,k_2, c'est-à-dire que les pièces déjà placées sur deux cases voisines[2] imposent à cette case de recevoir une pièce de l'ensemble *Offer*(k_1,k_2).

Pour chaque pièce nous maintenons également les quatre ensembles *Offer*(k_1,k_2) dans les 4 directions. De même pour chaque case nous maintenons quatre ensembles *Demand*(k_1,k_2) (au plus) auxquels elle appartient. Ainsi, pour chaque affectation d'une pièce à une case avec une certaine rotation, c'est à dire pour chaque affectation($t(i,j)$, $r(i,j)$) := (τ, ρ), les structures *ColorPair* sont maintenues de la manière suivante. La pièce τ est supprimée de ses quatre ensembles *Offer*(k_1,k_2) et la case (i,j) est supprimée de ses ensembles *Demand*(k_1,k_2). Puis les quatre cases voisines sont analysées (sauf celles qui ont déjà une pièce) pour déterminer si le positionnement de la pièce τ en (i,j) amène ces cellules voisines à intégrer de nouveaux ensembles *Demand*(k_1,k_2). Par exemple, si la case (i+1,j+1) contient une pièce τ' avec la rotation ρ', alors si la case (i+1,j) est vide elle demande désormais la séquence de couleurs $\tau[1+\rho],\tau'[2+\rho']$ dans le sens des aiguilles d'une montre. La case (i+1,j) doit alors être ajoutée à l'ensemble *Offer*(k_1,k_2), qui ici correspond à *Offer*($\tau[1+\rho \bmod 4],\tau'[2+\rho' \bmod 4]$) comme illustré sur la figure 1.

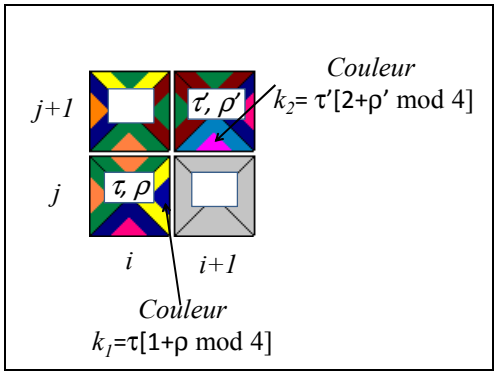

Figure 1

Le maintien de ces structures *ColorPair*(k_1,k_2) permet d'effectuer les inférences suivantes :

- **Contradiction**. Si |*Offer*(k_1,k_2)| < |*Demand*(k_1,k_2)| alors le nombre de cases demandant la séquence de couleurs k_1,k_2 dans le sens des aiguilles d'une montre est plus grand que le nombre de pièces non encore posées offrant cette séquence. Le problème est donc inconsistant.
- **Instanciation**. Si |*Offer*(k_1,k_2)| = |*Demand*(k_1,k_2)| = 1 alors une seule pièce τ peut être placée sur la case (i,j) demandant la séquence de couleur en question. Si cette pièce ne présente qu'une seule fois[3] la séquence k_1,k_2 alors il y a aussi une unique rotation possible ρ de la pièce τ.

3.2. Propriétés

Remarquons tout d'abord que la notion de paires de couleurs « en séquence » (dans l'ordre des aiguilles d'une montre) se généralise au cas des N(N-1)/2 paires de couleurs opposées. Dans ce cas seules les pièces de type [k_1, k_1, k_2, k_2] peuvent présenter deux fois les couleurs k_1 et k_2 en opposition. Tout ce que nous écrirons ici sur les paires en séquence s'applique également sur les paires en opposition. Nous avons implémenté ces deux visions si bien que chaque pièce appartient au plus à 6 ensembles *Offers*. Dans la suite de cet article, nous appellerons D le nombre maximum de pièces possibles pour une paire de couleurs en séquence *ou* en opposition.

En termes de complexité, l'algorithme de maintien des structures *ColorPair* décrit en section 3.1 nécessite à chaque affectation au plus 6 retraits de pièce d'ensembles *Offer* et 6 retraits de cases d'ensembles *Demand*, puis, après analyse des arêtes des cases voisines non instanciée, au plus 4x6 ajouts de case dans des ensemble *Demand*. En représentant ces ensembles dans des structures assurant un ajout et un retrait en temps constant[4], on obtient un filtrage en *O(1)*. L'opération inverse (*unassign*) a la même complexité.
Si l'on considère maintenant le filtrage obtenu sur une case, on constate que le domaine qui serait inféré par la *4-edges consistency* définie plus haut, est contenu dans l'intersection des *Offers*(k_1,k_2) demandés. En particulier pour les cases à 0 et 2 voisins le domaine obtenu par *4-edges consistency* est identique à l'ensemble *Offer*(k_1,k_2) correspondant à la paire de couleurs voisines. Notre modèle n'effectue aucun filtrage sur les cases à un seul voisin mais un tel filtrage est généralement très faible : par exemple sur Eternity II™, connaître un voisin d'une case conduit à un domaine de 43 à 49 pièces. Par contre il serait utile d'utiliser des structures similaires aux *ColorPair* pour comparer le nombre de cases demandant une couleur k_0 au

[2] Naturellement, avoir une arête sur le bord du plateau est équivalent pour une case à avoir une pièce voisine présentant la couleur 0. Nous ne reviendrons plus sur ce point dans le reste de l'article et compterons les bords du plateau comme des « voisins ».

[3] Notons que, si $k_1 \neq k_2$, la seule pièce pouvant présenter deux fois une séquence k_1,k_2 est [k_1,k_2,k_1,k_2]. Quant aux répétitions d'une séquence k_1,k_1 elles ne sont possibles que sur les pièces [k_1,k_1,k_1,k_1] et [k_1,k_1,k_1,k_3] avec k_3 une couleur quelconque.

[4] Nous utilisons la structure classique dans laquelle chaque élément maintient sa propre position dans un tableau. Les |X| premières positions de ce tableau contiennent les éléments de l'ensemble X.

nombre de pièces disponibles offrant cette couleur. Les mêmes règles de *contradiction* et d'*instanciation* s'appliqueraient.
Quant aux cases à 3 ou 4 voisins nous verrons en section 4.2 que notre stratégie de descente branche sur ces types de cases dès qu'elles apparaissent. Réduire leur domaine est alors moins crucial. C'est pour ces raisons que nous ne maintenons pas les domaines des variables en elles même mais plutôt ces structures *ColorPair* qui, sans être plus nombreuses (K^2 au lieu de N^2), présentent l'avantage d'être de taille réduite et de permettre un déclenchement direct des règles de *contradiction* et d'*instantiation* définie plus haut. Or ces règles produisent des inférences que ne détecte pas une consistance locale. En effet en considérant une case isolément on ne pourrait détecter que l'absence de pièce disponible correspondant au voisinage d'une case ou l'unicité d'une telle pièce mais pas des contradictions globales portant sur le nombre total de cases nécessitant un type de pièce.

4. Filtrage renforcé

4.1. Préemptions

Notre modèle à base de *ColorPair*(k_1,k_2) permet également un autre type de déduction.
Si |*Offer*(k_1,k_2)| = |*Demand*(k_1,k_2)| alors il est certain que toutes les pièces de *Offer*(k_1,k_2) seront nécessairement utilisées sur les cases de *Demand*(k_1,k_2). Par exemple dans le premier cas de la figure 2, on peut affirmer que les deux pièces présentant la séquence 1,2 iront sur les deux cases demandant cette séquence. Nous dirons que ces pièces sont *préemptées* par *ColorPair*(k_1,k_2) et elles sont donc supprimées des ensembles *Offer*(k_3,k_4) auxquels elles appartenaient, ce qui peut entrainer de nouvelles inférences : instanciations, contradictions ou même de nouvelles préemptions. Notons ici que ce raisonnement n'est valable qu'en l'absence de cases à 3 ou 4 voisins instanciés, c'est-à-dire si les ensembles *Demand*(k_1,k_2) sont deux à deux disjoints. Ainsi sur le même cas de la figure X, l'existence d'une pièce ayant des arêtes 5,1,2 interdit de supprimer les deux pièces « 1,2 » de *Offer(5,1)* car une telle suppression rendrait |*Offer*(5,1)| strictement inférieur à |*Demand(5,1)*|, causant ainsi une contradiction infondée. En pratique les pièces préemptées ne sont donc pas retirées des ensembles *Offer* mais simplement marquées comme *preempted* et un compteur *nbPreempted* est maintenu sur chaque *Offer*. On considère alors que le nombre de pièces disponible est |*Offer*| ou |*Offer*|-*nbPreempted* selon qu'il y a ou non des cases à trois voisins. Comme annoncé plus haut, nous verrons en section 4.2 que notre stratégie de descente amène à se trouver la plupart du temps dans le second cas (pas de case à 3 ou 4 voisins).
En termes de couplages, cette notion de préemption correspond à la détection d'une forme particulière de Composante Fortement Connexe (CFC) sur le graphe orienté défini à partir d'un couplage de référence (voir [Regin 1994]). Ce filtrage n'est cependant pas équivalent au *AllDifferent* complet, comme le montre le second cas de la figure 2.

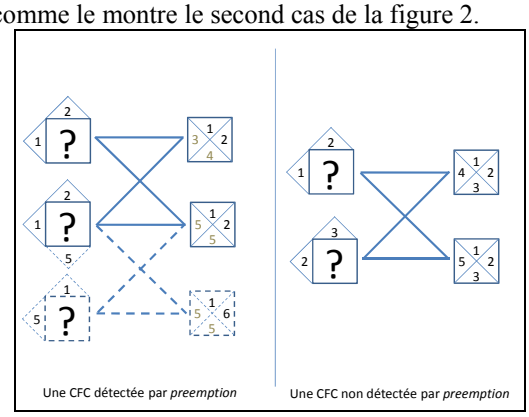

Figure 2

Par contre la complexité est bien plus faible qu'un « vrai » *AllDifferent*. Ces CFC particulières sont détectées en O(1), et le marquage des pièces comme *preempted* se fait en 6x|*Offer*| opérations. Comme chaque pièce peut au plus être préemptée par une *ColorPair* la complexité amortie de ce marquage est $O(N^2)$ pour toute branche de l'arbre de recherche soit une complexité moyenne de *O(1)* par nœud.
Nous verrons en section 6 qu'un ratio de 10 à 100 est observé sur le temps total, entre l'approche renforcé efficace et la modélisation standard avec *AllDifferent* en $O((N^2)^{2.5})$.

4.2. Raisonnement sur les pièces non placées

Borne sur les arêtes partagées

Avant même d'avoir positionné des pièces il est possible de détecter qu'elles ne pourront pas être placées de manière correcte. Par exemple s'il n'y a plus d'arêtes rouges demandées et qu'il n'en reste que deux non placées, sur la même pièce, alors placer cette pièce sera impossible. Plus généralement s'il reste *m* arêtes d'une couleur *k* et seulement m' demandées, alors *m-m'* arêtes de couleur *k* devront se toucher les unes les autres. Or [Benoist 2008] montre que le nombre maximal d'arêtes que peuvent partager N pièces est égal à *h(N)* :

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

Ainsi si ces *m* arêtes sont réparties sur *m''* pièces, la propriété *m-m' ≤ 2h(m'')* doit être respectée, chaque arête partagée « consommant » deux facettes de couleur *k*. Cette propriété est vrai pour tout ensemble de couleur, mais n'est vérifiée en temps constant que si l'on ne considère que les singletons, chaque affectation nécessitant la vérification de quatre couleurs au plus.

Détection de cycle

Si k est une couleur de la bordure et qu'il n'y a plus que deux arêtes de couleur k à placer et aucune case demandant la couleur k, alors, sous réserve que ces deux arêtes soient sur deux pièces distinctes, ces deux pièces seront nécessairement voisines. Dans une telle situation (détectable en temps constant), on peut facilement chercher si d'autre liens similaire ne prolongent pas cette chaîne : une chaine de longueur c sera identifiée en un temps $O(c)$ et c ne peut pas excéder K puisque chaque arc de la chaîne correspond à une couleur. Si la chaîne est en fait un cycle alors le bord ne peut plus être complété.

Ces deux règles de filtrage ont une complexité très faible et permettent de détecter des inconsistances sur les pièces non encore placées. Malheureusement nous verrons en section 6 que le gain obtenu sur le nombre de nœuds explorés est de l'ordre de 2%.

5. Stratégie d'exploration

Nous décrivons dans cette section les deux caractéristiques principale de notre recherche arborescente à savoir notre stratégie de descente (choix de la case à affecter) et notre stratégie de remontée c'est-à-dire la détection des branches dont l'exploration est inutile.

5.1. Stratégie de descente (Branching Strategy)

Motifs particuliers

Comme évoqué en section 3.1, le grand nombre de pièces possibles pour les cases ayant zéro ou une voisine suggère de ne brancher que sur les cases ayant au moins deux voisines. Une telle stratégie est possible car tant que le plateau n'est pas entièrement couvert de pièces, il existe nécessairement au moins une case vide ayant deux voisines instanciées. Par exemple la case vide de coordonnées (i,j) minimales au sens lexicographique a nécessairement à sa gauche et en dessous soit un bord soit une case instanciée, donc deux arêtes de couleurs déterminées. Nous qualifierons ces cases à deux voisines instanciées ou plus de « *cases entourées* ».

Deux critères guident notre choix de la prochaine case à considérer. Nous privilégions tout d'abord les cases les plus contraintes c'est-à-dire, par priorités décroissantes :

1. Les cases ayant quatre voisines instanciées
2. Les cases ayant trois voisines instanciées
3. Les cases ayant deux voisines instanciées et une voisine entourée
4. Les cases ayant une voisine instanciée et deux voisines entourées. Dans ce dernier cas, nous choisissons alors d'instancier l'une des ces deux voisines entourées.

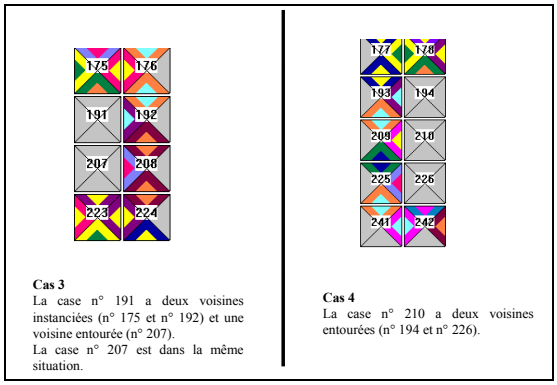

Figure 3

Les cas 1 et 2 correspondent à des situations particulièrement contraintes dans lesquelles le nombre de pièces possibles pour une case risque d'être faible voire nul[5]. Privilégier ces variables obéit à un principe de type *MinDomain*. Le cas 3 correspond au cas d'une case dont l'instanciation contraindra fortement une case voisine. Sur l'exemple considéré dans la figure ci-dessus, placer une pièce sur la case n° 191 fera de la case n°207 une case à trois voisines : quelque soit la taille du domaine de la case n° 191, le nombre de pièces ne rendant pas impossible l'instanciation de la case n° 207 est probablement très faible voir nul. Enfin dans le cas 4 pris en exemple, l'instanciation d'une des deux cases n° 194 et n° 226 conduira à la situation du cas 3. La détection de ces motifs se fait en temps constant, simplement en maintenant les nombres de cases voisines *instanciées* et *entourées* pour chaque case.

Heuristiques de choix de la paire la plus « urgente »

En l'absence de tels motifs nous appellerons *MinOffer* la stratégie consistant à sélectionner une *ColorPair*(k_1,k_2) vérifiant *Demand*$(k_1,k_2)\neq\varnothing$ et telle que *Offer*(k_1,k_2) soit de cardinalité minimale (*MinDomain* classique) puis nous choisissons aléatoirement une case dans *Demand*(k_1,k_2). Si l'on maintient, en temps constant, D ensembles de *ColorPair* correspondant aux D cardinalités possibles des ensembles *Offer*, alors la sélection de cette *ColorPair* se fait en $O(D)$[6].

Il est possible d'améliorer cette stratégie de choix de la *ColorPair* en appliquant un raisonnement probabiliste simple. Pour chaque paire

[5] Rappelons que nous ne maintenons pas de consistance locale autour de chaque case : l'intersection des ensembles *Offer*(k_1,k_2) attachés à la cellule peut tout à fait être vide (d'où l'importance de privilégier ces cases dans la stratégie de choix de variable).

[6] Rappelons que D est généralement petit : inférieur à 12 sur Eternity II™

ColorPair(k_1,k_2), la probabilité pour chacune des pièces de *Offer*(k_1,k_2) d'être utilisée sur une des case de *Demand*(k_1,k_2) vaut a priori *Demand*(k_1,k_2)/ *Offer*(k_1,k_2). On vérifie que cette probabilité vaut 0 si la paire k_1,k_2 n'est demandée par aucune case et 1 s'il y a autant de demandes que d'offre pour cette paire, c'est-à-dire dans le cas d'une *préemption* (décrit en section 4). Raffinons cette estimation en considérant pour une pièce τ l'ensembles *pairs*(τ) de toutes les paires p telles que $\tau \in offer_p$, en notant respectivement $offer_p$ et $demand_p$ les deux ensembles attachés à un paire p. Nous pouvons alors approximer la probabilité que la pièce τ soit utilisée sur une des cases de $demand_p$ par la formule

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

En effet pour être utilisée par p_0 il faut que cette pièce τ ne soit utilisée par aucune autre paire. Nous appelons alors *expected offer,* et notons $E(|offer_p|)$ la somme de ces probabilités pour toutes les pièces de $offer_p$.

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

A partir de cette valeur nous pouvons définir un critère à minimiser lors du choix, parmi les paires demandées, de la paire sur laquelle effectuer le prochain branchement.

- *MinExpectedOffer* consiste simplement à choisir la paire p minimisant $E(|offer_p|)$,
- *MinExpectedGap* consiste à minimiser $E(|offer_p|) - |demand_p|$, afin de focaliser la recherche sur les paires les plus susceptibles de causer une contradiction,
- *MinExpectedRatio* consiste à minimiser le ratio $E(|offer_p|) / |demand_p|$, sur le même principe que la critère précédent.
- *MaxExpectedFiltering* consiste à minimiser $E(|offer_p|) - |offer_p|$. Plus cette valeur est négative, plus le choix d'une pièce pour une des cases correspondante aura de conséquences sur d'autres paires.

Nous analysons les gains apportés par ces différentes heuristiques en section 6.

Dans tous les cas, motifs particulier ou non, et quelle que soit le critère de choix de la paire, les valeurs possibles pour la case choisie sont calculées à partir de la paire *ColorPair*(k_1,k_2) ayant le plus petit ensemble *Offer*(k_1,k_2) parmi celles demandées par la case. L'heuristique de choix parmi ces valeurs n'a pas d'importance dans notre étude car nous étudions des explorations complètes. Néanmoins trier ces pièces par probabilités $P(\tau \to p_0)$ décroissantes semble assez naturel.
Dans le cas des cases à 3 ou 4 voisins, certaines pièces de l'ensemble *Offer*(k_1,k_2) peuvent ne pas être des valeurs possible si aucune rotation ne permet de respecter les couleurs des 3 ou 4 arêtes déjà fixées. Par ailleurs rappelons que certaines pièces peuvent au contraire être placées sur une case avec deux rotations différentes (voir note 3). Finalement la liste des branches issues d'un nœud est une liste d'élément T × [0,3] c'est-à-dire des couples pièces/rotation compatible avec les couleurs actuelles des arêtes de la case.

Shaving

Au cours de nos expérimentations, nous avons également constaté que l'application d'un shaving systématique offrait un gain significatif en temps. C'est-à-dire qu'à chaque nœud nous considérons toutes les cases ayant deux voisins ou plus et nous essayons chacune des pièces possibles. Si nos algorithmes de filtrage rejettent toutes ces valeurs alors le nœud courant est inconsistant. Si une seule valeur est consistante alors nous pouvons affecter cette pièce à cette case. Sinon, nous ne déduisons rien de ce shaving.

5.2. Stratégie de remontée (Backtracking Strategy)

Exemple

Lors d'un échec dans l'arbre de recherche, plutôt que de revenir au nœud immédiatement supérieur, il peut être préférable de détecter la cause de l'échec afin de remonter plus haut dans l'arbre, jusqu'à une variable responsable du conflit évitant ainsi d'explorer des sous-arbres voués à l'échec.
Dans l'exemple ci-dessous, en testant chacune des deux pièces restantes (dans *Offers(1,2)*) pour la case marquée « ? » on constate que chacune mène à un échec. Or supposons que les nœuds précédents de l'arbre concernent des cases non représentées sur ce dessin. Remettre en cause ces cases ne résoudra pas le problème détecté ici et conduira donc à explorer en vain un nombre de nœuds potentiellement grand. Au contraire, si l'on parvient à détecter que tant qu'aucune des cases dessinées ici avec leur couleurs ne sera modifiée le conflit demeurera, alors on pourra effectuer un retour-arrière « intelligent » ou *conflict-based backjumping* (Gaschnig 1979, Ginsberg 1993) en remontant dans l'arbre jusqu'à trouver un nœud impliquant une de ces cases.

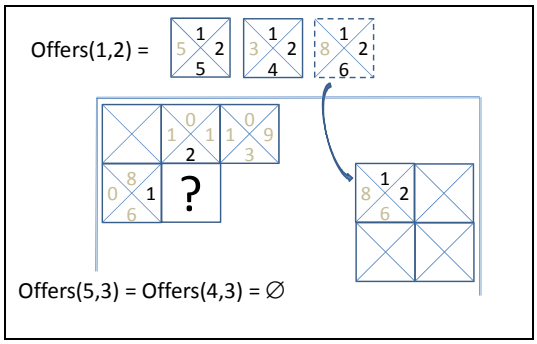

Figure 4

Calcul d'explications

Lorsque nous détectons une contradiction il importe donc de pouvoir déterminer sa cause c'est-à-dire un ensemble de cases (appelé *explication*) dont la valeur actuelle suffit à expliquer la contradiction. Dans l'arbre de recherche, lorsque toutes les branches filles d'un nœud ont été explorées sans qu'une solution ne soit trouvée, l'explication de ce nœud est simplement l'union des explications des nœuds fils.
Dans notre modèle, les échecs sont détectés de deux manières seulement:

- Soit en écartant d'emblée certains couples pièce/rotation. Il s'agit du cas particulier des cases à 3 ou 4 voisines décrit en section 4.2, et pour lequel l'explication est simplement l'ensemble des 3 ou 4 cases voisines instanciées.
- Soit par la règle $|Offer(k_1,k_2)| < |Demand(k_1,k_2)|$ définie au paragraphe 3.1. Dans ce cas l'explication est composée de :
    - Pour chaque case de $Demand(k_1,k_2)$, les deux voisines, dans l'ordre des aiguilles d'une montre présentant respectivement les couleurs k_1 et k_2 ;
    - Pour chaque pièce de $Unavailable(k_1,k_2)$, la case sur laquelle elle a été placée.

Ces explications sont valides puisque retirer toutes les pièces placées sauf celles des cases de l'explication ne résout pas le conflit. Elles sont également minimales au sens de l'inclusion c'est-à-dire qu'il existe des cas où dé-instancier une seule case de l'explication résout le conflit.
Remarquons au sujet des pièces τ de $Unavailable(k_1,k_2)$ que, si elles ont sur leurs côtés k_1 et k_2 deux voisines instanciées, alors l'explication contenant ces deux voisines au lieu de la case contenant τ est également valide. De plus, si ces pièces ont été placées avant τ alors cette nouvelle explication est meilleure car elle implique des cases plus « anciennes » et permettra donc un retour à un nœud plus élevé dans l'arbre.

Dans le cas de pièces préemptées par une *ColorPair* (voir section 4), l'explication de leur non disponibilité et composée comme dans le cas des contradiction de type $|Offer(k_1,k_2)| < |Demand(k_1,k_2)|$. Cette explication est enregistrée au moment ou la préemption est détectée et réutilisée ensuite à chaque fois que la préemption d'une des pièce doit être justifiée.

La complexité du calcul d'une explication est terminée par l'opération la plus couteuse du calcul c'est-à-dire les parcours de l'ensemble $Demand(k_1,k_2)$ et de l'ensemble $Unavailable(k_1,k_2)$, soit une complexité de $O(D)$ (rappelons que D désigne la taille du plus grand ensemble *Offer*). Quant à l'union de deux explications, sa complexité théorique est $O(N^2)$ mais une implémentation sous la forme de vecteurs de bit permet d'avoir en pratique une efficacité toute à fait satisfaisante.

Tableau 1. Résultats

instance	*doc_smith backtracker*		*ColorPairs*		*ColorPairs without backjumping*		*ColorPairs without preemptions*		*CP_choco*	
	time (seconds)	nodes (millions)	time (seconds)	nodes (millions)	time (seconds)	nodes (millions)	time (seconds)	nodes (millions)	time (seconds)	nodes (millions)
E_10_5	0	2,60	3	0,69	4	1,02	2	0,83	152	0,001
E_10_7	0	0,37	3	0,71	3	0,88	3	0,99	50	0,0004
E_9_3	0	14,82	7	1,57	8	2,36	7	2,10	206	0,004
E_10_8	0	15,44	17	4,24	24	7,10	15	5,22	530	0,005
E_10_16	17	467,97	2	0,42	3	0,70	1	0,53	222	0,002
E_9_4	1	54,04	3	0,55	3	0,66	2	0,61	21	0,0004
E_10_11	1	31,89	8	1,92	14	3,70	8	2,88	462	0,005
E_10_9	1	39,73	18	4,35	35	9,97	13	4,75	528	0,005
E_10_17	22	596,36	3	0,77	4	1,16	2	0,75	123	0,001
E_9_13	177	5 441,01	3	0,57	3	0,75	2	0,63	262	0,006
E_10_12	2	53,89	11	2,57	14	3,97	8	2,75	480	0,0045
E_10_22	98	2 924,86	4	0,82	4	1,10	3	0,94	275	0,003
E_9_11	48	1 444,37	4	1,02	6	1,47	4	1,23	408	0,008
E_10_18	31	556,06	5	1,22	8	2,44	5	1,64	190	0,002
E_9_12	148	4 464,16	6	1,50	8	1,93	5	1,68	585	0,013
E_7_1	34	908,35	7	1,73	7	1,89	6	1,77	382	0,037
E_7_2	565	14 367,24	11	2,50	10	2,86	8	2,57	283	0,028
E_9_5	8	262,37	19	4,65	24	6,53	19	6,20	399	0,008
E_10_14	12	371,16	11	2,56	16	4,55	9	3,18	1357	0 ,016
E_9_7	16	489,14	11	2,59	14	3,82	10	3,31	472	0,008
E_10_25	1141	32 599,16	12	2,98	18	4,81	12	4,17	1667	0,019
E_9_10	N/A	N/A	12	2,86	20	5,31	13	4,06	1029	0,009
E_10_21	43	1 271,18	18	4,36	38	10,53	19	6,71	643	0,007
E_8_1	1997	56 161,34	23	5,76	31	8,67	19	5,93	1446	0,070
E_10_19	39	1 166,51	79	19,37	118	33,26	67	23,94	1135	0,013
E_10_20	42	979,04	86	21,82	164	46,54	87	31,11	1381	0,014
E_10_15	N/A	N/A	77	18,62	132	37,04	81	28,09	7687	0,097
E_10_13	N/A	N/A	124	29,39	324	90,59	125	43,20	5756	0,062
E_9_14	432	13 067,46	149	36,21	208	57,08	133	44,20	22013	0,223
E_10_23	185	5 390,09	169	40,84	379	105,61	144	50,44	2211	0,025
E_10_10	N/A	N/A	170	42,60	454	129,49	160	55,44	8420	0,100
E_9_6	N/A	N/A	252	65,04	290	84,64	206	66,79	24114	0,299
E_9_2	N/A	N/A	240	58,26	298	82,73	216	67,36	24927	0,635
E_9_16	1517	45 278,80	249	59,93	432	114,77	218	70,53	16700	0,163
E_7_3	647	16 886,24	244	56,04	263	69,43	230	69,03	11450	1,048
E_9_15	992	30 083,49	359	86,35	430	116,43	381	125,16	26242	0,306
E_10_6	N/A	N/A	471	115,54	838	232,60	365	126,71	N/A	N/A
E_10_4	N/A	N/A	468	120,51	726	216,90	416	143,35	N/A	> 0,729
E_9_9	N/A	N/A	1565	394,45	2135	598,46	1226	400,86	N/A	N/A
E_9_1	N/A	N/A	2006	495,26	2466	691,14	1704	547,70	N/A	N/A
E_9_8	N/A	N/A	2378	583,97	2973	803,49	2045	656,54	N/A	N/A
E_10_26	3030	85 381,84	2634	638,35	3601	1 010,00	2171	763,53	N/A	N/A
E_10_1	N/A	N/A	2239	566,69	3601	1 065,00	2575	802,87	N/A	N/A
E_10_3	N/A	N/A	3217	797,72	3601	1 066,00	3323	1 128,31	N/A	N/A
E_10_2	N/A	N/A	3275	836,33	3603	1 015,00	3271	1 055,83	N/A	N/A

6. Résultats expérimentaux et conclusions

Nous comparons ici différentes versions de notre algorithme avec un des meilleurs algorithmes arborescents disponibles, posté par « doc_smith »[7], sur le forum consacré à Eternity II™. Cette recherche arborescente implémentée en C parvient à explorer des millions de nœuds par seconde. Des structures de données adaptées permettent de trouver rapidement l'ensemble des pièces possibles pour chaque case, c'est-à-dire l'ensemble des fils d'un nœud, mais aucun raisonnement global n'est implémenté pour éliminer des branches.

Nous avons généré pour ces expériences 140 puzzles de tailles 7x7 à 10x10 de la façon suivante. Pour une plateau de côté N et un nombre maximum de couleurs K, nous parcourons les 2N(N-1) arêtes de la grille et affectons à chacune une couleur choisie aléatoirement (et uniformément) dans l'intervalle [1,K]. Les pièces ainsi crées sont pivotées et mélangées aléatoirement. Appliquant nos algorithmes sur toutes ces instances[8], nous avons conservé les 45 pour lesquelles au moins une approche parvenait à énumérer l'intégralité des solutions en moins d'une heure. Pour ces 45 instances nous comparons le temps total et le nombre total de nœuds explorés. Dans le tableau 1 la mention N/A désigne les recherches arborescentes non terminées au bout d'une heure de temps de calcul. Les instances ont été triées par meilleur temps croissant. On constate que pour les instances qu'au moins un des algorithmes résout en moins d'une seconde, explorer 4.6 fois moins de nœuds en moyenne grâce à la PPC ne suffit pas à obtenir les meilleurs temps. Par contre on voit sur le tableau 2 ci dessous que pour des instances plus difficiles, le gain en nombre de nœuds permet rapidement des gains significatif en temps.

Tableau 2. Gains en nœuds et en temps

When the best algorithm takes…	*< 1 s*	*1s to 10s*	*11s to 3mn*
ColorPairs divides the number of nodes by…	4.3	1760	2720
ColorPairs divides the total time by…	-	13.4	23.8

Enfin, parmi les 14 instances dont nous énumérons les solutions en 3 à 60 minutes, le *backtracker* pris comme référence ne termine son exploration en moins d'une heure que dans 4 cas.

Si nous comparons maintenant les apports respectifs de la détection des préemptions (section 4) et du backjumping (section 5.2) nous constatons que ces deux approches participent à la minimisation du nombre de nœuds à explorer. Sans le backjumping on mesure 51% de nœuds en plus en moyenne, et sans la détection des préemptions 24% de nœuds en plus en moyenne. Si le gain en temps est clair pour le backjumping (30% de temps en plus si l'on s'en passe), on ne met pas en évidence d'apport véritable des préemptions sur le temps total. En d'autre termes une fois misent en place les deux règles d'instanciation et de contradiction qui détectent globalement en O(1) qu'il n'y a plus de couplage possible, la détection des préemptions n'offre pas un compromis filtrage/complexité suffisant.

Dans le même ordre d'idée, nous avons lancé sur un certain nombre d'instances le modèle avec *AllDifferent* et *Element* évoqué en section 2.2 implémenté en CHOCO [Laburthe&al 2000]. Les temps obtenus peuvent aller jusqu'à plus de 50 fois notre approche *ColorPair* de référence (sans shaving et sans heuristique de choix de variable), malgré les gains constaté sur le nombre total de nœud (jusqu'à un facteur de 1000).

Enfin le tableau 3 présente les résultats respectifs des heuristiques de choix de variables présentées en section 4.2, en omettant la stratégie *MaxExpectedFiltering* qui s'est révélée moins bonne que le simple *MinOffer*. Les gains sont ici les baisses du nombre de nœud et du temps total obtenues par comparison à la stratégie *MinOffer*.

Tableau 3. Apports des heuristiques

When the best algorithm takes…		*< 1 s*	*1s to 10s*	*11s to 3mn*	*3mn to 60mn*
MinExpectedOffer	*time gain*	25%	18%	15%	7%
	node gain	25%	24%	24%	18%
MinExpectedGap	*time gain*	*29%*	*19%*	*45%*	*29%*
	node gain	*28%*	*33%*	*52%*	*39%*
MinExpectedRatio	*time gain*	14%	18%	39%	23%
	node gain	24%	27%	47%	34%

On constate que les stratégies comparant le nombre de pièces demandées pour une paire au nombre de pièces susceptible d'être disponibles pour cette paire (*expected offer*) sont systématiquement les plus performantes en nœuds et en temps.

Quant à l'apport du shaving il est en moyenne de 78% sur le nombre de nœuds et de 66% sur le temps total.

En conclusion, ces résultats montrent que, pour énumérer les solutions d'un puzzle de type *edge-matching*, les algorithmes de filtrages proposés dans cet article (*ColorPairs* + *preemptions*) combinés à une heuristique efficace (*expected gap*) et à un backtracking intelligent bien connu (*backjumping*) surpassent déjà nettement les

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

[8] Pour ces expériences nous avons fixé les 4 coins des instances. Des résultats sans fixer ces coins sont disponibles sur http://pagesperso-orange.fr/tbenoist/, ainsi que l'intégralité des instances utilisées dans cet article.

approches focalisées sur la vitesse d'exploration de chaque nœud pour des tailles 10x10. Pour aller plus loin, la performance de cette recherche arborescente pourrait bénéficier à une recherche locale à grand voisinage visant à résoudre le problème Eternity II™ à 256 pièces. En effet, on constate qu'améliorer une solution partielle nécessite généralement de déplacer un grand nombre de pièces, ce qui semble condamner des approche par recherche locale à « petits voisinages ». Nous avons implémenté une telle recherche à grand voisinage qui cherche actuellement une solution à cette instance 16x16.

Au-delà de ces puzzles, ce travail vient également contredire l'idée assez répandue selon laquelle la Programmation Par Contraintes paie la puissance de ses inférences par une complexité élevée à chaque nœud, l'un ne compensant pas toujours l'autre. Certains travaux ont montré qu'un filtrage moins fort pouvait se révéler gagnant lorsqu'il offre une forte diminution de la complexité (Lopez-Ortiz et al.). Ici aussi nous montrons qu'en exploitant au mieux les propriétés du problème et en utilisant des structures de données dédiées et efficaces, il est parfois possible d'obtenir un filtrage suffisamment puissant et global en un temps raisonnable, voire en temps constant.

References

T. Benoist (2008). *How many edges can be shared by N square tile on a board ?* In e-lab research report, april 2008.

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

J. Gaschnig (1979). *Performance measurement and analysis of certain search algorithms.* Tech. rep. CMU-CS-79-124, Carnegie-Mellon University.

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

François Laburthe et le projet OCRE (2000). *Choco : Implémentation du noyau d'un système de Contraintes*, Sixièmes journées Nationales de la Résolution Pratiques de Problèmes NP-Complet, JNPC'00.

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

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

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

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