Hybridation de la programmation par contraintes et d'un voisinage à très grande taille pour Eternity II.

Pierre Schaus **Yves Deville**

Département d'ingénierie informatique,
Université de Louvain,
Place Sainte Barbe 2,
B-1348 Louvain-la-Neuve, Belgique

{pierre.schaus,yves.deville}@uclouvain.be

Résumé

Eternity II est un puzzle avec compatibilité entre côtés des pièces créé par Christopher Monkton pour l'éditeur de jeu Tomy(TM). Étant donné 256 pièces carrées dont chaque côté est colorié et un tableau de 16×16, le but est de placer toutes les pièces sur le plateau de sorte que deux pièces adjacentes ont leur côté commun de même couleur. Ce problème est NP-complet, il possède très peu de structure et est fortement combinatoire ($256! \cdot 4^{256}$ combinaisons possibles).

Christopher Monkton est tellement confient du fait de la difficulté de trouver une solution à son problème qu'il promet un prix de $2 millions à la première personne qui trouvera une solution. Nous n'avons (désormais) plus grand espoir de trouver une solution à ce problème, mais nous avons néanmoins décidé d'expliquer notre stratégie pouvant être utile à quelqu'un croyant toujours en ses chances de succès. Notre procédure consiste en une initialisation par programmation par contrainte en relâchant le problème. Ensuite, nous améliorons la solution avec une recherche locale stochastique utilisant un voisinage de très grande taille. Notre voisinage est de très grande taille (exponentielle) mais peut néanmoins être exploré en temps polynomial en solutionnant un problème d'appariement avec poids dans un graphe biparti. Notre procédure nous permet d'obtenir rapidement de bonnes solutions avec des scores jusqu'à 458/480 jonctions satisfaites. Notre grand voisinage peut également être utilisé dans une approche bruteforce pour tester $256!/128! \cdot 4^{128}$ combinaisons au lieu de $256! \cdot 4^{256}$. Cette article est également un exemple de VLNS (Very Large Neighborhood Search) pouvant être appliqué à d'autres types de problèmes de placement avec compatibilité entre les objets.

1 Introduction

Le puzzle Eternity II (E2) consiste en n^2 pièces carrées dont chaque côté est colorié, et d'un plateau de jeu de taille $n \times n$. Les pièces doivent être placées sur le plateau de sorte que deux pièces adjacentes ont les côtés correspondants de même couleur. La couleur extérieure du bord du plateau est imposée. La Figure 1 donne un exemple de solution pour un problème de taille 7×7.

E2 est une instance de taille 16×16. Étant donné que la couleur extérieure est imposée, ces contraintes ne sont pas difficiles à satisfaire. C'est pourquoi le score d'une solution est toujours exprimé en nombre de jonctions intérieures satisfaites sur les $2n \cdot (n-1)$ jonctions internes pour puzzle $n \times n$ (sur 480 pour l'instance E2 16×16).

De nombreuses personnes travaillent activement sur la résolution d'E2. Le groupe de discussion le plus significatif dénombre plus de 2000 membres [1]. Deux logiciels de calcul distribués on été développés utilisant une approche brute-force de type backtracking :

– Le premier a été développé par Dave Clark. Environ 1000 ordinateurs calculaient de manière permanente pour ce projet. Ce projet était le plus populaire et à notre connaissance, il est celui qui a soumis la

[1] http://games.groups.yahoo.com/group/eternity_two/

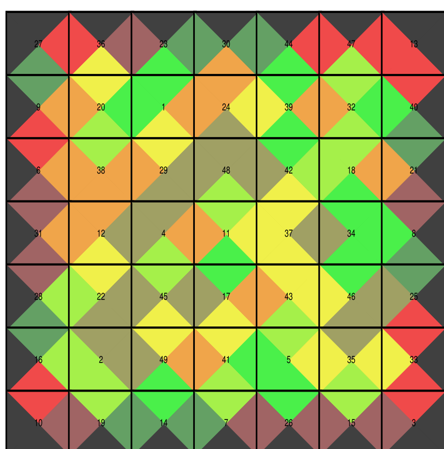

FIG. 1 – Solution exacte d'une instance 7x7

meilleure solution avec un score de 463/480. Ce projet a duré 6 mois. Dave Clark a maintenant abandonné car il ne croyait plus dans l'approche brute-force pour résoudre le puzzle.
– Le deuxième est un projet français [2] qui semble avoir moins de membres. Ils n'ont pas encore publié leur meilleur score.

Tetravex [3] ainsi que E2 [4] sont tous deux des puzzles avec compatibilités entre côtés des pièces. Cette catégorie de puzzle a été démontrée NP-complète [4]. En particulier Tetravex a été prouvé NP-complet par réduction de 1in3-SAT [10].

E2 n'est pas un problème qui a beaucoup de structure. En effet, les contraintes sont très locales. La seule contrainte impliquant toutes les pièces étant que deux mêmes pièces ne peuvent être placées à la même position du plateau. Les couleurs des pièces d'E2 ont été choisies par Christopher Monckton de sorte que les retours en arrière (backtracks) surviennent assez profondément dans l'arbre de recherche (environ après 160 pièces placées). La répartition des couleurs est donnée à la Figure 2. Il y a 22 couleurs, la couleur 0 étant pour le contour du plateau. Les couleurs 1-5 ne sont présentes que sur les pièces de contour (avec un ou deux 0) et ne peuvent être utilisée que pour le coté adjacent des pièces de contour. On peut voir sur la figure que la répartition des couleurs ne peut être plus équitable. Enfin, le nombre de couleurs a probablement été choisi de sorte qu'il y ait très peu de solution sans pour autant contraindre de trop le problème. Il est très facile de construire un contour valide même à la main et placer ensuite une centaine de pièces. Afin de réduire le nombre de solutions et les possibilités d'avoir des solutions symétriques, Monckton a créé des pièces toutes différentes. Il impose également la position d'une pièce indice au centre du plateau. Cette pièce indice rend le problème encore plus difficile car même les symétries de rotation sont supprimées.

[2] http://www.eternity2.fr/
[3] http://live.gnome.org/Tetravex/
[4] http://uk.eternityii.com/

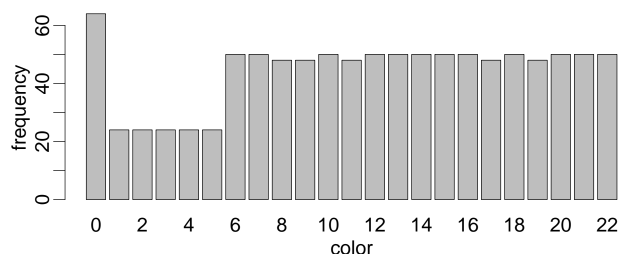

FIG. 2 – Répartition des couleurs sur les 256 pièces d'E2

Beaucoup de techniques standards peuvent être utilisées pour tenter de résoudre E2. Nous pensons que ce jeu pourrait devenir un benchmark standard pour l'optimisation combinatoire.

Voici un résumé des approches que nous avons tentés pour atteindre notre meilleur score de 458/480 :

1. En utilisant un modèle de programmation par contraintes (PC), nous ne sommes pas capables de résoudre de manière exacte des instances de taille plus grande que 8×8 ayant les mêmes caractéristiques qu'E2. Pour l'instance officielle d'E2, nous devons relâcher environ 80 contraintes de jonction pour être capable de le résoudre. Par conséquent, une approche utilisant la programmation par contraintes pure ne permet pas d'atteindre des scores plus grands que 400/480.
2. Nous avons également essayé une recherche locale stochastique tabu utilisant des mouvements échangeant deux pièces. En démarrant d'un placement aléatoire des pièces, cette approche n'était pas capable d'atteindre des scores au-delà de 410/480. Par contre, une instanciation utilisant la solution obtenue par la PC sur le problème relâché nous permettait d'obtenir des scores autour de 425/480.
3. Finalement, nous avons amélioré la recherche locale en utilisant un très grand voisinage. Notre voisinage est capable de déplacer de manière optimale plus de deux pièces à la fois. La seule contrainte étant que les pièces déplacées ne soient pas adjacentes par un côté. Le nombre de pièces déplacées en une fois peut donc atteindre $n^2/2$. Avec ce voisinage très grand, nous avons pu atteindre un score de 458/480 en moins d'un jour de calcul sur un ordinateur standard. Ce score peut être considéré comme rivalisant avec l'état de l'art. Il montre que le voisinage utilisé est prometteur pour résoudre E2 ou, néanmoins, atteindre des scores élevés.

Contributions : Notre contribution majeure est la présentation d'un voisinage de très grande taille pouvant être

exploré de manière optimale en solutionnant un problème d'appariement avec poids dans un graphe biparti. Notre voisinage peut potentiellement être appliqué à n'importe quel puzzle devant faire correspondre des pièces avec compatibilité ou autres problèmes de placement. Nous montrons également qu'une recherche locale basée sur ce voisinage permet d'obtenir des scores plus élevés lorsque la solution initiale est générée par une solution (partielle) réalisée à l'aide de la PC sur un problème (relâché) d'E2.

Plan : La section 2 décrit le modèle de PC. La section 3 explique comment construire et résoudre le très grand voisinage. La section 4 donne notre procédure tabu faisant usage du très grand voisinage. La section 5 présente une hybridation possible de la PC et de la recherche locale pour résoudre E2. Finalement, la section 6 conclut en donnant des résultats expérimentaux.

2 Un modèle de programmation par contraintes

La programmation par contraintes est un paradigme où un problème est modélisé en déclarant des variables avec leur domaine de valeurs possibles ainsi que des contraintes entre ces variables. Le moteur de recherche essaie de trouver une affectation des variables à une valeur de leur domaine satisfaisant l'ensemble des contraintes. Les contraintes ont également la responsabilité de retirer des valeurs inconsistantes du domaine des variables (propagation). Lorsque le point fixe de propagation est atteint et que tous les domaines ne sont pas des singletons et non vides, deux branches sont créées. La première branche réduit le domaine d'une variable à une seule valeur et la branche alternative retire cette valeur du domaine de la variable. L'arbre de recherche est généralement exploré en profondeur d'abord. En résumé, la recherche d'une solution n'est rien d'autre que l'exploration d'un arbre de recherche alternant affectation et propagation. La recherche effectue un retour en arrière dès qu'un domaine devient vide. On peut espérer trouver des solutions plus rapidement en décidant de manière heuristique la variable à instancier et la valeur à lui affecter dans chaque noeud de l'arbre. Pour plus de détails sur la PC en général, nous nous référons à [9].

Les variables : Les différentes couleurs sont $\{0, ..., c\}$. Pour chacune des 16x16 positions (i, j) du plateau, nous définissons les variables suivantes :

- U_{ij}, R_{ij}, D_{ij}, L_{ij} avec domaine $\{0, ..., c\}$ représentent respectivement les couleurs des côtés supérieur, droit, inférieur et gauche de la pièce venant en position (i, j) sur le plateau,
- $I_{ij} \in \{0, ..., n^2-1\}$ est l'identifiant de la pièce venant à cette position.

Nous ajoutons également les variables suivantes permettant une heuristique de branchement plus efficace :

- $O_{ij} \in \{0, ..., 3\}$ représente l'orientation de la pièce venant à cette position (nombre de quarts de rotation dans le sens des aiguilles d'une montre),
- $IO_{ij} \in \{0, ..., 4n^2 - 1\}$ représente à la fois l'identifiant de la pièce et son orientation à cette position.

Les variables redondantes sont liées aux premières avec

$$\forall (i, j) \in [1..n] \times [1..n] : \; IO_{ij} = 4 \cdot I_{ij} + O_{ij}. \quad (1)$$

De manière alternative, on peut utiliser une contrainte `element` [6] spécifiant que $IO_{ij} \in [4k..4k + 3]$ si et seulement si $I_{ij} = k$ et, $IO_{ij} \bmod 4 = l$ si et seulement si $O_{ij} = l$. Ce deuxième ensemble de contraintes de liaison est préférable à la contrainte arithmétique (1). En effet, si la variable IO_{ij} est assignée, les valeurs prisent par I_{ij} et O_{ij} ne peuvent être inférées si la consistance de bornes est appliquée sur (1) [5]. Au contraire, pour la contrainte `element`, dès que IO_{ij} est assignée, l'identifiant de la pièce $I_{ij} = IO_{ij}/4$ et son orientation $O_{ij} = IO_{ij} \bmod 4$ sont inférés.

Les contraintes : Premièrement, tous les I_{ij} doivent être différents. Cela peut être réalisé avec la contrainte globale `AllDifferent` [8]. Nous devons aussi faire en sorte que les quatre couleurs U_{ij}, R_{ij}, D_{ij}, L_{ij} correspondent à une pièce physique.

La première solution est d'utiliser des contraintes en extension. Puisque n^2 pièces sont données, pour chacune, quatre quadruplets de couleur peuvent être créés correspondant aux quatre orientations possibles de la pièce. Donc un total de $4n^2$ quadruplets peuvent être créés. La contrainte est que $[U_{ij}, R_{ij}, D_{ij}, L_{ij}]$ doit appartenir à cet ensemble de quadruplets. Cela peut être réalisé avec les contraintes données en extension [2].

Une autre possibilité est d'utiliser une contrainte `element`. En effet, lorsqu'une pièce et son orientation sont connues pour une position (i, j), les valeurs pour U_{ij}, R_{ij}, D_{ij}, L_{ij} peuvent être récupérées dans quatre tableaux de valeurs encodant tous les tuples valides.

Les contraintes que les couleurs de côtés adjacents doivent être les mêmes s'expriment simplement comme $D_{i,j} = U_{i+1,j}$ et $R_{i,j} = L_{i,j+1}$. Il y a également une contrainte spécifiant que la couleur de contour est 0.

Heuristiques de branchement : Nos expériences ont montré qu'il était plus intéressant de brancher sur les variables IO_{ij} plutôt que fixer les pièces I_{ij} et ensuite l'orientation O_{ij}. L'heuristique utilisée est un *first-fail* classique : le choix de la variable à instancier est le IO_{ij} ayant le plus petit domaine. En cas d'égalité, les variables sont

[5] La consistance de domaines pour les contraintes arithmétiques est généralement trop coûteuse.

départagées aléatoirement. Le choix de la valeur est choisi aléatoirement dans le domaine de la variable.

Améliorations possibles : Le problème de notre modèle est la recherche en profondeur d'abord utilisée. En effet, la recherche peut toujours placer un grand nombre de pièces (typiquement 160) avant de devoir faire retour arrière. Le retour arrière ne s'effectue jamais au niveau des premières pièces placées. Ces choix survenant très tôt dans la recherche ne sont donc plus jamais remis en question. Une recherche plus intelligente pourrait utiliser les impacts et les redémarrages afin de guider mieux la recherche et reconsidérer rapidement les premiers choix [7]. Une autre amélioration possible serait d'augmenter le filtrage effectué. On pourrait par exemple imaginer maintenir une consistance sur des contraintes globales de type *domino* par ligne et par colonne.

3 Un voisinage très grand exploré en temps polynomial

La plupart des problèmes combinatoires ne sont pas traitables de manière exacte et E2 en fait partie. Néanmoins, nous pouvons généralement obtenir d'excellentes solutions à l'aide d'algorithmes fonctionnant par améliorations successives sur une solution courante. L'idée est de partir d'une solution initiale et d'appliquer successivement des modifications améliorantes à cette solution. Le problème est qu'en fonction des modifications envisagées, la solution peut rapidement être bloquée dans un optimum local ou dans un état de cyclage entre deux modifications. Ces situations peuvent être évitées à l'aide de meta-heuristiques. La recherche tabu [5] simple mais très efficace en pratique en est une. Cette recherche consiste à maintenir des mouvements interdits pendant un certain temps afin d'éviter de cycler et dans le but de diversifier l'espace de recherche.

Peu importe la méta-heuristique utilisée, il y a peu d'espoir de trouver de bonnes solutions sans un bon voisinage et une méthode efficace pour l'explorer. En effet, le voisinage est très important pour éviter les optima locaux. Plus la taille du voisinage est grand, mieux c'est car cela laisse plus de possibilités pour échapper aux optima locaux.

Un voisinage est dit très grand par rapport aux données d'entrées s'il est exponentiel. Malheureusement, il peut être très coûteux d'explorer un voisinage de grande taille à chaque itération. Il faut généralement trouver un compromis entre la vitesse et la taille du voisinage. Pour certains problèmes, il est possible d'imaginer de très grands voisinages pouvant être explorés rapidement (en temps polynomial). Pour le célèbre voyageur de commerce, des voisinages de très grandes tailles ont été imaginés [1]. La sélection d'un voisin dans un voisinage de très grande taille se fait généralement en solutionnant un problème d'optimisation tel que trouver un chemin de longueur minimum ou

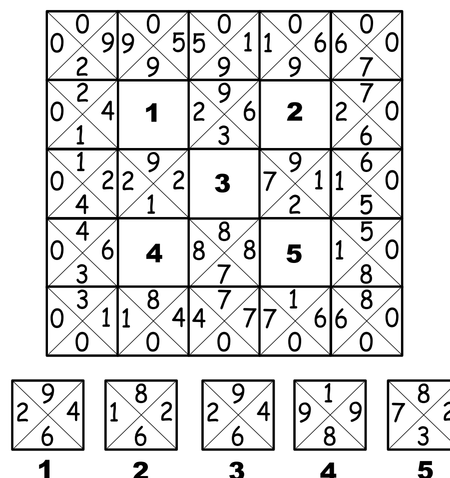

FIG. 3 – 5 pièces non adjacentes par un côté sont retirées de la solution courante laissant donc 5 trous dans le plateau.

résoudre un problème d'appariement (éventuellement avec poids).

Un petit voisinage : Pour le problème E2, le premier voisinage venant à l'esprit est probablement d'échanger deux pièces en leur appliquant éventuellement une rotation. Appelons ce mouvement l'*échange et tourne* . Pour une solution, il y a $n^2 \cdot (n^2 - 1) \cdot 16$ mouvements *échange et tourne* possibles. Pour réduire cette complexité, il est possible de procéder en deux étapes en choisissant d'abord la première pièce (typiquement la plus violée) et ensuite la seconde sur base de la première. La complexité pour explorer le voisinage est alors en $O(n^2)$ plutôt que $O(n^4)$.

Un voisinage de très grande taille : Nous proposons de généraliser le mouvement *échange et tourne* à plus de deux pièces. Nous considérons un mouvement d'échange et de rotation sur un ensemble de pièces. Malheureusement, pour un ensemble de pièces retirées du plateau, les replacer de manière optimale est aussi difficile que résoudre E2 lui-même. Néanmoins, si l'on choisit correctement l'ensemble des pièces, il est possible de les replacer de manière optimale dans les trous en temps polynomial. En effet, si aucune des pièces de l'ensemble ne se touchent par un côté, elles peuvent être replacées de manière optimale dans les trous en résolvant un problème d'appariement de coût maximum. La Figure 3 montre un ensemble de pièces qui ne sont pas adjacentes par un côté qui sont retirées de la solution courante.

Les pièces retirées peuvent être replacées dans les trous de manière optimale en solutionnant un problème d'appariement avec coût. Le problème d'appariement est défini sur le graphe biparti complet entre les pièces retirées et les trous. Un arc entre une pièce et un trou est annoté par un

TAB. 1 – Annotation (orientation, poids) des arcs du graphe biparti des pièces vers les trous de l'exemple de la Figure 3.

Pièces	Trous 1	2	3	4	5
1	2,3	2,2	0,1	1,1	1,1
2	0,1	0,1	2,2	1,3	3,3
3	0,1	0,1	0,1	1,1	1,1
4	1,2	1,2	0,1	0,2	1,2
5	0,1	0,1	2,4	1,1	3,2

couple (r, w) :

– r est la rotation optimale de la pièce si elle est placée à cette position et
– w est le nombre de correspondances correctes pour la pièce si elle est placée à cette position avec la rotation r.

Nous avons $r \in [0..3]$ (nombre de rotations de quart de tour dans le sens des aiguilles d'une montre) et $w \in [0..4]$. La Table 1 donne les annotations pour les arcs de l'exemple de la Figure 3.

Une fois que les poids optimaux et les rotations correspondantes des pièces ont été calculées, nous pouvons calculer l'appariement de poids maximum sur le graphe biparti. Les arcs sélectionnés nous indiquent où et comment replacer les pièces de manière optimale dans les trous. Trouver l'appariement de poids maximum dans un graphe biparti se résout en temps polynomial par exemple avec l'algorithme Hongrois ou un algorithme primal-dual en $O(m^4)$ [6] pour un graphe biparti pondéré de $m \times m$ (nous nous référons à [3] pour plus d'informations sur les problèmes d'appariement).

Sur notre exemple, les arcs sélectionnés (pièce $\mapsto$ trou) sont $(1 \mapsto 1)$, $(2 \mapsto 4)$, $(3 \mapsto 3)$, $(4 \mapsto 5)$ et $(5 \mapsto 3)$.

En résumé, les étapes pour construire un voisinage sont :

1. Sélectionner un ensemble S de positions non adjacentes par les côtés.
2. Calculer les annotations (r, w) pour chacun des $|S|^2$ arcs des pièces vers les trous.
3. Calculer l'appariement de poids maximum sur le graphe biparti pondéré.
4. Les arcs de l'appariement optimal donnent la permutation optimale des positions de S, ainsi que les rotations des pièces indiquées sur les annotations des arcs sélectionnés.

La taille du voisinage est $|S|! \cdot 4^{|S|}$ où $|S|$ peut aller jusqu'à $n^2/2$. Le voisinage est donc bien de taille exponentielle.

[6] Des algorithmes plus rapides existent (en $O(m^3)$) mais ils sont également plus compliqués [3].

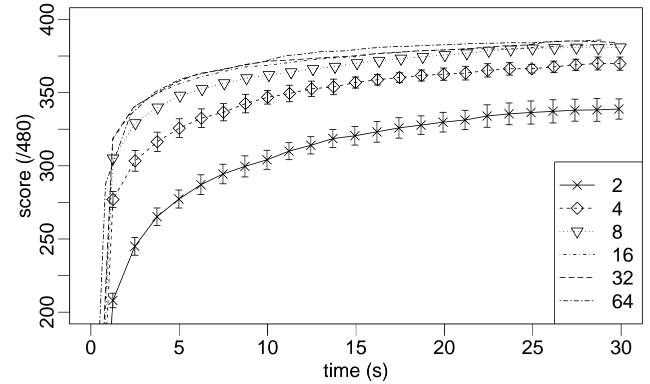

FIG. 4 – Influence du nombre de pièces sélectionnées dans le voisinage de grande taille sur la qualité de l'optimum local.

La Figure 4 nous donne plus intuition sur l'influence de la taille de l'ensemble S. Nous avons appliqué successivement le voisinage à grande taille durant 30 secondes par sélection d'ensembles aléatoires de taille k de pièces non adjacentes par les côtés . Pour chaque k, 10 expériences ont été effectuées en démarrant à chaque fois d'un placement aléatoire des pièces. Nous avons essayé pour k les valeurs 2, 4, 8, 16, 32 et 64. La Figure 4 donne l'évolution moyenne du score (480 étant une solution parfaite) sur 30 secondes. Les écarts types ne sont représentés que pour $k = 2$ et $k = 4$ afin d'alléger le graphique. La qualité de l'optimum locale n'est plus réellement améliorée pour des valeurs de k au delà de 16.

4 Une recherche Tabu

Une recherche tabu permet de diversifier la recherche en évitant les minima locaux et les cyclages en rendant certains mouvements interdits durant un certain nombre d'itérations (nous nous référons à [5] pour plus d'informations sur la recherche tabu). Généralement, les mouvements interdits sont maintenus dans une liste tabu. Le nombre d'itérations durant lesquelles un mouvement se trouve dans la liste tabu est la *tenure* de la liste tabu. Deux listes tabus peuvent être imaginées pour le mouvement de grande taille décrit ci-dessus :

– Une première liste tabu peut stoker durant quelques itérations les positions récemment choisies pour constituer l'ensemble S. Les positions constituant S seront alors diversifiées au long des itérations.
– Une seconde liste tabu peut interdire certaines permutations entre positions. Considérons qu'à l'itération courante, une pièce en position i est déplacée vers la position j (avec $i \neq j$). Afin d'éviter de cycler entre ces deux positions, il est souhaitable d'éviter le mouvement opposé durant quelques itérations. La paire

(j, i) est donc ajoutée à la liste tabu spécifiant qu'une pièce en position j ne peut être déplacée vers la position i tant que (j, i) est dans la liste tabu. Cela est réalisé facilement en attribuant un très petit poids à l'arc (j, i) dans la construction du graphe biparti pondéré.

Algorithm 1: Algorithme tabu pour E2 utilisant un voisinage de grande taille

$tabu_1 \leftarrow list()$.
$tabu_2 \leftarrow list()$.
while *condition de fin non satisfaite* **do**
  if *condition de diversification* **then**
    diversification
  1. Sélection d'un sous ensemble S de pièces non adjacentes par les côtés ne se trouvant pas dans $tabu_1$.
  2. Ajouter les éléments de S à $tabu_1$ pour un certain nombre d'itérations.
  3. Calculer et appliquer le mouvement optimal sur S en solutionnant le problème d'appariement de poids maximal et en pénalisant les arcs de $tabu_2$.
  4. Ajouter les arcs opposés du mouvement à $tabu_2$.

  if *condition d'intensification* **then**
    rétablir la meilleure solution rencontrée.

L'algorithme 1 est une description assez haut niveau. Nous détaillons un peu plus certaines parties de l'implémentation :
- Pour la sélection de S, les positions les plus violées sont choisies préférentiellement.
- La condition de diversification est satisfaite lorsqu'un plateau d'une taille donnée est rencontré.
- La diversification consiste en quelques mouvements aléatoires d'*échange et tourne*.
- La condition d'intensification est satisfaite si le meilleur score n'est mas amélioré après un nombre donné de plateaux détectés consécutifs.
- La condition de terminaison est satisfaite après un certain nombre d'intensifications.
- La tenure des listes tabu est choisie pour chaque ajout aléatoirement entre deux nombres.

5 Hybridation

Le voisinage de grande taille décrit à la section précédente permet de replacer en temps polynomial de manière optimale des pièces non adjacentes par les côtés. Ce voisinage peut également être utiliser pour réduire le coût d'une approche *brute-force*. Le nombre de placements possibles des pièces sur le plateau est de $n^2! \cdot 4^{n^2}$. Il est possible de réduire fortement ce nombre en plaçant seulement une pièce sur deux à la manière des cases noires d'un jeu

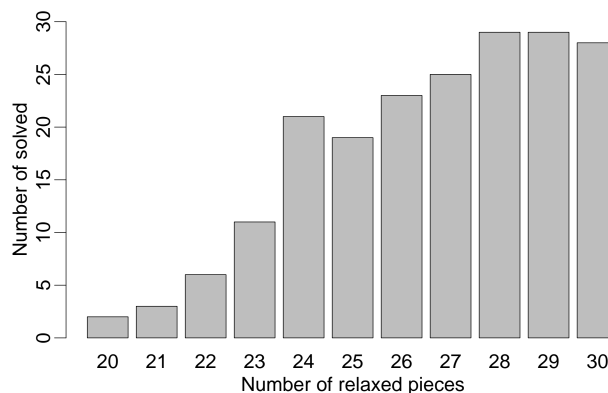

FIG. 5 – Nombre de problèmes relâchés résolus sur 30 instances.

d'échec. Le nombre de placements possibles est alors réduit à $\frac{n^2!}{(n^2/2)!} \cdot 4^{n^2/2}$. Il reste ensuite $n^2/2$ positions vides qui ne sont pas adjacentes par les côtés. Ces positions sont remplies de manière optimale par les pièces restantes à l'aide du mouvement basé sur l'appariement de plus grand poids.

Une autre approche exacte peut combiner la programmation par contrainte et la recherche locale utilisant le voisinage de grande taille :
- E2 n'est pas traitable par la programmation par contraintes. Néanmoins, une solution peut être trouvée à l'aide de la programmation par contraintes sur un problème relâché en permettant à certaines jonctions d'être non satisfaites. En particulier, si les jonctions relâchées sont sur des positions non adjacentes par les côtés (par exemple, un sous ensemble de côtés des positions noires du jeu d'échec), on peut espérer trouver une solution avec la programmation par contraintes.
- Ces positions relâchées peuvent être ensuite remplies de manière optimale avec le voisinage de grande taille.
- S'il y a des violations, répéter la procédure pour la solution suivante donnée par la programmation par contraintes.

Nous avons relâché un certain nombre de positions non adjacentes par les côtés choisies aléatoirement. Pour chaque nombre, nous avons généré 30 instances relâchées d'E2. La Figure 5 donne le nombre d'instances pouvant être résolues en moins de 30 secondes. Il apparaît que le nombre de positions à relâcher doit être supérieur à 24 pour avoir une chance raisonnable de trouver une solution au problème relâché.

La procédure venant d'être décrite peut également être utilisée pour générer de bonnes solutions initiales pour démarrer une recherche locale. L'hybridation que nous avons expérimentée commence une recherche locale avec l'algorithme 1 sur de telles solutions obtenues par la PC sur un

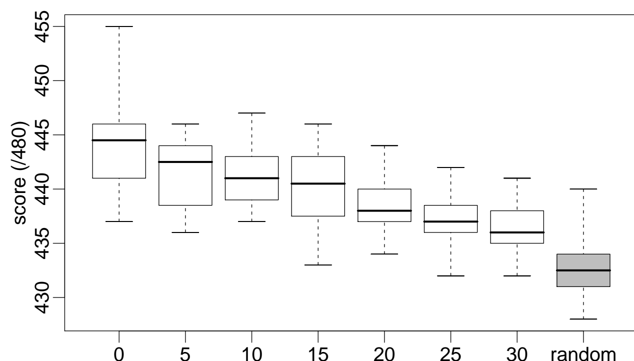

FIG. 6 – Diagrammes boîtes à moustaches des scores obtenus sur 20 exécutions avec une initialisation par PC en utilisant un nombre de pièces relâchées variant (0,5,10,15,20,25,30) ainsi qu'avec une initialisation aléatoire.

problème relâché. Si la PC n'est pas capable de trouver une solution dans le temps imparti, la plus grande affectation partielle observée durant la recherche est complétée aléatoirement (noeud le plus profond de l'arbre de recherche).

6 Section expérimentale

Nous expérimentons dans cette section l'initialisation avec la PC pour différents nombres de positions relâchées. Pour chaque nombre de positions relâchées, nous faisons 20 exécutions avec l'hybridation décrite dans la section précédente. Nous avons également essayé de commencer au départ d'une initialisation aléatoire. Les diagrammes boîtes à moustaches des scores obtenus sont montrés sur la Figure 6. Les boîtes représentent la moyenne et les quartiles et les moustaches s'étendent sur les valeurs extrêmes.

Il semble que l'algorithme de recherche locale 1 soit meilleur avec une initialisation par PC qu'avec une initialisation aléatoire (boîtes blanches contre boîte grise). Ce qui est plus surprenant, c'est que les scores les plus élevés sont obtenus avec moins de pièces relâchées. En effet, à la section précédente, nous avons vu que pour moins de 20 pièces relâchées, la PC n'était pas capable de trouver une solution. Dans ce cas, l'état initial est la plus grande affectation partielle complétée aléatoirement avec les pièces restantes. Nous pensions intuitivement que plus le score de l'état initial était élevé, meilleur serait le score final obtenu avec la recherche locale. Ce n'est pas le cas. Commencer avec une bonne affectation partielle sur un problème non relâché donne de meilleurs résultats que commencer avec une affectation partielle plus grande mais sur un problème relâché. Notre explication est que si nous commençons avec 30 positions relâchées et que nous trouvons une solution à ce problème relâché puis que les pièces relâchées sont replacées à l'aide du voisinage de grande taille, nous commençons précisément dans un optimum local et il est très difficile pour notre recherche locale de s'en échapper.

En laissant tourner le programme 24 heures, nous avons pu obtenir plusieurs solutions avec un score de 458 qui peut être considéré comme un résultat de l'état de l'art pour E2.

Toutes les expériences ont été conduites sur un processeur Intel Xeon(TM) 2.80GHz. Nous avons utilisé la librairie de PC Gecode 2.0.1 [11]. La recherche locale à voisinage de grande taille a été implémentée en C++. Notre implémentation maintient incrémentalement les violations à chaque mouvement.

7 Conclusions et perspectives

Nous avons imaginé un voisinage à très grande taille pour E2 pouvant être exploré de manière optimale en temps polynomial en trouvant un appariement de poids maximum dans un graphe biparti pondéré. Nous avons également expliqué comment ce voisinage peut être utilisé pour réduire le nombre de combinaisons à essayer dans une approche brute-force exacte. Nous avons donné une procédure tabu basique utilisant le voisinage et nous avons montré que l'initialisation au départ d'une solution (partielle) générée par la PC pouvait améliorer les résultats obtenus par la recherche locale.

Nous ne pensons pas qu'E2 pourra être résolu par des méthodes exactes ni par des méthodes heuristiques dans un futur proche. Néanmoins, nous pensons que les deux approches sont intéressantes à combiner pour obtenir de meilleures solutions. La PC n'est probablement pas un bon choix pour l'initialisation car elle fait un retour en arrière dès qu'un domaine devient vide. Les approches utilisées par la plupart des personnes sont plutôt de placer un maximum de pièces avec leurs jonctions satisfaites. Faire un retour arrière dès qu'un domaine se vide n'est peut-être pas une bonne idée pour cela. En effet, nous pourrions continuer avec les domaines non vides jusqu'à ce que tous les domaines soient vides. Il semble que des personnes soient capables de placer plus ou moins 210 pièces avec cette approche alors que la PC n'est pas capable de produire des solutions partielles avec plus de 170 pièces placées. Il peut être intéressant d'utiliser les initialisations faites par ces personnes pour démarrer notre recherche locale. Enfin, le mouvement à grand voisinage que nous proposons peut également s'intégrer dans bien d'autres méta-heuristiques et être combiné avec d'autres voisinages.

Remerciements

Cette recherche est supportée par la région wallonne, le projet Transmaze (516207) et partiellement par le programme des pôles d'attraction inter-universitaires (État belge, Politique de science belge).

Références

[1] Ravindra K. Ahuja, Özlem Ergun, James B. Orlin, and Abraham P. Punnen. A survey of very large-scale neighborhood search techniques. *Discrete Applied Mathematics*, 123(1-3) :75–102, 2002.

[2] Christian Bessière, Jean-Charles Régin, Roland H. C. Yap, and Yuanlin Zhang. An optimal coarse-grained arc consistency algorithm. *Artif. Intell.*, 165(2) :165–185, 2005.

[3] R. Burkard, M. DellAmico, and S. Martello. *Assignment Problems*. SIAM Monographs on Discrete Mathematics and Applications, 2008.

[4] Erik D. Demaine and Martin L. Demaine. Jigsaw puzzles, edge matching, and polyomino packing : Connections and complexity. *Graph. Comb.*, 23(1) :195–208, 2007.

[5] Fred Glover. *Tabu Search*. Kluwer Boston, Inc., 1998.

[6] Van Hentenryck P. and Carillon J.-P. Generality versus specificity : an experience with ai and or techniques. *In Proceedings of AAAI-88*, 1988.

[7] Philippe Refalo. Impact-based search strategies for constraint programming. In *CP*, pages 557–571, 2004.

[8] Jean-Charles Régin. A filtering algorithm for constraints of difference in csps. In *AAAI '94 : Proceedings of the twelfth national conference on Artificial intelligence (vol. 1)*, pages 362–367, Menlo Park, CA, USA, 1994. American Association for Artificial Intelligence.

[9] Francesca Rossi, Peter van Beek, and Toby Walsh. *Handbook of Constraint Programming (Foundations of Artificial Intelligence)*. Elsevier Science Inc., New York, NY, USA, 2006.

[10] Yasuhiko Takenaga and Toby Walsh. Tetravex is np-complete. *Inf. Process. Lett.*, 99(5) :171–174, 2006.

[11] Gecode Team. Gecode : Generic constraint development environment. available from http ://www.gecode.org. 2006.