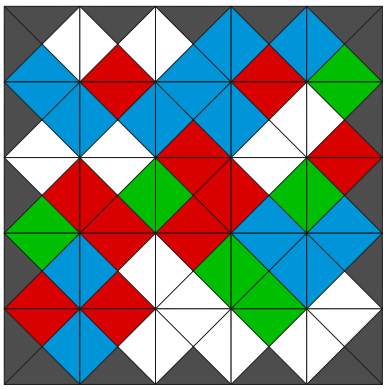
Licence d'informatique fondamentale
École normale supérieure de Lyon
2007-2008

Eternity II : puzzles, algorithmique et complexité

Timo Jolivet



Stage effectué au Laboratoire d'informatique fondamentale de Marseille
Encadrants : Grégory Lafitte et Nicolas Ollinger

Table des matières

1 Introduction

1.1 Règles du jeu

Eternity II est un puzzle inventé par Christopher Monckton, commercialisé par Tomy depuis le 28 juillet 2007. La première personne qui aura résolu ce puzzle avant le 31 décembre 2008 gagnera 2 millions de dollars US.

Le puzzle est constitué de 256 tuiles carrées de même taille dont les arêtes sont colorées. Le but est de disposer toutes les tuiles sur une grille 16×16 de manière à ce que les arêtes communes des tuiles adjacentes aient la même couleur. Une couleur spéciale force le placement de certaines tuiles au bord de la grille. De plus, l'emplacement précis d'une certaine tuile est imposé dès le départ, sur un des carrés au centre de la grille. On n'a pas le droit de retourner les tuiles mais les rotations sont autorisées.

On dénombre au total 22 couleurs d'arêtes différentes (sans compter la couleur de bord) pour : 4 tuiles de coin, 56 tuiles de bord et 196 tuiles centrales. Le nombre de configurations (valides ou non) qui respectent les contraintes de bord et la contrainte de la pièce initiale est de $4!\cdot 56!\cdot 195!\cdot 4^{195}$, soit environ $1,11 \cdot 10^{557}$. Le nombre de solutions annoncé par Christopher Monckton est d'environ 20000 ; une recherche exhaustive de solutions à la main ou à l'ordinateur paraît peu envisageable.

Des « puzzles indices » sont également vendus. Leur résolution permet d'obtenir l'emplacement exact d'une pièce supplémentaire. Cela n'est sûrement qu'une technique commerciale, destinée à vendre plus de puzzles et non à aider les possesseurs du puzzle : le placement d'une pièce supplémentaire réduit le nombre de solutions.

1.2 Généralisation du problème.

Nous nous intéresserons à des variantes d'Eternity II en modifiant la taille de la grille et le nombre de couleurs. Ceci nous pousse à considérer un nouveau problème de décision, que nous appellerons ETER2 dans la suite.

Une instance d'ETER2 est un jeu de mn carrés unité dont chaque arête est colorée par un élément de $\{0, 1, \ldots, k\}$. Il doit y avoir une couleur spéciale (0, gris) qui force l'arête à être sur le bord de la grille ainsi que : 4 tuiles tuiles ayant exactement deux arêtes perpendiculaires grises et $2(m + n - 4)$ tuiles n'ayant qu'une seule arêté de grise. Aucune autre tuile ne doit avoir d'arête grise, de manière à ce qu'un certain sous-ensemble de tuiles forme un bord.

Le problème est de décider s'il est possible de placer toutes les tuiles sur une grille $m \times n$ de sorte que les arêtes communes des tuiles adjacentes soient de la même couleur et qu'une arête ne soit sur le bord de la grille si et seulement si sa couleur est 0. Voir les figures 1 et 2 pour des exemples.

Le nombre de solutions d'une instance ne tient pas compte d'éventuelles symétries des tuiles (ou d'éventuelles tuiles en double) : deux orientations

équivalentes d'une même tuile (ou la permutation de deux tuiles égales) conduisent à deux solutions différentes. Dans la suite, les solutions seront toujours dénombrées à rotation près.

1.3 Cadre et but du stage

Le stage s'est déroulé au sein de l'équipe Escape dans le laboratoire d'informatique fondamentale de Marseille, du 9 juin au 18 juillet 2008.

Les deux principales problématiques du stage sont : « Comment résoudre une instance ? » et « Qu'est-ce qui rend une instance difficile à résoudre ? », ce qui demande de réfléchir à une bonne définition de la difficulté d'une instance. La première question a résulté en l'étude et l'implémentation de différents algorithmes de résolution, en OCaml. Ils ont ensuite été comparés et utilisés pour étudier le comportement du puzzle sur de grands échantillons d'instances de petite taille.

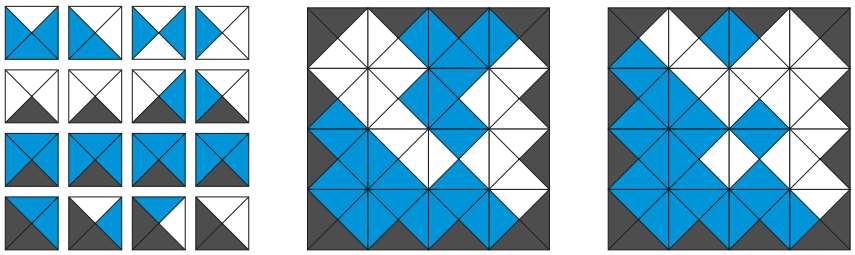


FIG. 1: Une instance avec deux de ses solutions.

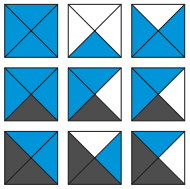


FIG. 2: Pas une instance d'ETER2, à cause des bords.

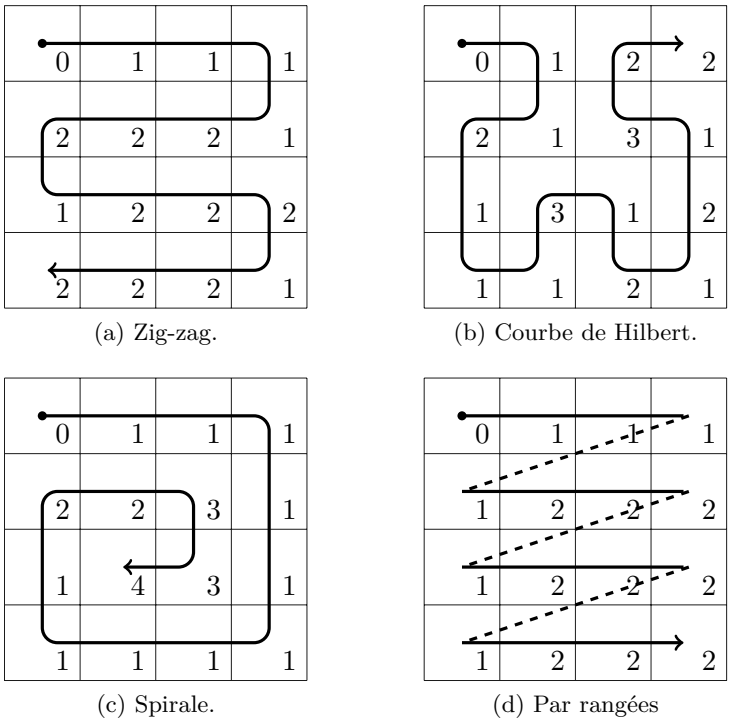
2 Algorithmes de résolution

2.1 Backtracking naïf

Une première approche consiste à poser des tuiles correctement en suivant un chemin qui remplit la grille jusqu'à ce qu'on ne puisse plus en poser. Si la grille n'est pas remplie, on enlève alors la dernière tuile posée pour en mettre une autre qui convient, puis on réitère récursivement le procédé.

Le parcours « par rangées » est le plus naturel, et c'est celui qui a été utilisé dans l'implémentation de l'algorithme. Les autres chemins ne sont *a priori* pas plus performants. En effet, un chemin maximisant les contraintes locales serait plus avantageux car il permettrait de mieux élaguer l'arbre de parcours, mais la somme des contraintes locales est la même quel que soit le chemin : c'est le nombre d'arêtes intérieures de la grille.

Notons qu'il vaut mieux privilégier un chemin ayant des contraintes fortes le plus tôt possible, afin d'éliminer des branches inutiles de l'arbre le plus vite possible.



(a) Zig-zag. (b) Courbe de Hilbert.

(c) Spirale. (d) Par rangées

FIG. 3: Chemins et contraintes locales. L'entier dans chaque case correspond au nombre de contraintes locales imposées au placement d'une pièce à cet endroit du chemin. La somme des contraintes locales vaut 24 pour chaque chemin.

2.2 Réduction vers Exact-Cover

Dans [5], Don Knuth présente un algorithme ingénieux pour trouver toutes les solutions d'une instance de Exact-Cover par backtracking. Divers problèmes de pavages sont ensuites réduits à Exact-Cover puis résolus grâce à l'algorithme initial, appelé DLX. Il est aussi possible de résoudre des instances d'ETER2 de la même manière, et c'est ce que nous allons faire.

Le problème de décision Exact-Cover. Une instance de Exact-Cover est une matrice dont les coefficients sont des 0 ou des 1. La question est de déterminer l'existence d'un sous-ensemble de lignes de la matrice contenant exactement un 1 dans chaque colonne.

Réduction d'ETER2 à Exact-Cover. Une instance d'ETER2 est la donnée de n tuiles sur les couleurs $0, 1, \ldots, k$ (0 est la couleur de bord) et d'une grille rectangulaire de n cases dont les arêtes intérieures sont numérotées de 1 à ℓ.

On part d'une matrice à $2n+k\ell$ colonnes : une colonne pour chaque tuile, une colonne pour chaque case et k colonnes pour chaque arête intérieure. À chaque pièce correspondent alors $4n$ lignes : 4 lignes pour chaque case ; une ligne par rotation.

Il reste à remplir les $k\ell$ colonnes qui sont utilisées pour coder les couleurs. À chaque arête intérieure correspond un k-uplet de colonnes. On associe à ces colonnes un k-bit dans chaque ligne concernant la tuile i :

- $0 \cdots 0 \underbrace{1}_{\text{bit } j} 0 \cdots 0$ si l'arête est de couleur j et que la tuile i est en haut ou à gauche de l'arête ;
- $1 \cdots 1 \underbrace{0}_{\text{bit } j} 1 \cdots 1$ si l'arête est de couleur j et que la tuile i est en bas ou à droite de l'arête ;
- $0 \cdots 0$ si la tuile i ne touche pas l'arête.

Notons qu'on place les pièces en respectant les contraintes au bord, ce qui réduit un peu la réduction. Il est également possible de coder les couleurs en binaire, ce qui donne un nombre de colonnes de $2n + \ell \log(k)$ couleurs.

Fonctionnement de l'algorithme. L'algorithme tourne sur une matrice A, un ensemble C représentant chaque colonne de A et S, une liste des lignes qui constituent les solutions partielles. Le choix de la colonne c peut être quelconque, mais on préférera choisir une colonne comportant le moins de 1 possible, pour parcourir l'arbre plus efficacement. L'appel initial se fait par $search(0)$.

De la même manière que pour l'algorithme naïf, les appels récursifs de la fonction $search(k)$ forment un arbre dont chaque niveau k correspond à un choix de k lignes.

Algorithme 1 *search*(k)

si C est vide **alors**
  afficher la solution $(S[0], S[1], \ldots, S[k])$
sinon
  choisir une colonne c
  pour chaque ligne ℓ telle que $A[\ell, c] = 1$ **faire**
    $S[k] \leftarrow \ell$
    pour chaque j tel que $A[\ell, j] = 1$ **faire**
      $C := C \backslash \{j\}$
      cacher la colonne j de A
      pour chaque i tel que $A[i, j] = 1$ **faire**
        cacher la ligne i de A
      fin pour
    fin pour
    exécuter *search*(k) sur la matrice réduite
    pour chaque j tel que $A[\ell, j] = 1$ **faire**
      $C := C \cup \{j\}$
      découvrir la colonne j de A
      pour chaque i tel que $A[i, j] = 1$ **faire**
        découvrir la ligne i de A
      fin pour
    fin pour
  fin pour
  découvrir la colonne c
fin si

Implémentation. Nous avons utilisé un logiciel écrit en OCaml par Jean-Christophe Filliâtre ([3]) implémentant l'algorithme DLX et fournissant une API utile pour mettre en oeuvre la réduction.

2.3 Groupage de tuiles

Une autre approche possible est de procéder par groupages de tuiles. L'algorithme implémenté pendant le stage établit d'abord une liste de toutes les tuiles 2×2 existantes puis fait du backtracking zig-zag sur des configurations quatre fois plus petites. Le problème ne reste cependant pas le même : il faut désormais tenir compte des dépendances entre les tuiles (une tuile normale peut apparaître dans plusieurs tuiles 2×2), et il ne faut plus utiliser toutes les tuiles mais seulement un sous-ensemble correspondant au bon nombre de cases. Cet algorithme ne marche que sur les configurations de hauteur et de largeur paires.

Le gain de temps énorme que laisse entrevoir la réduction du problème à une grille quatre fois plus petite est compensé par l'explostion du nombre

de pièces 2×2, si bien que les performances obtenues sont similaires aux deux algorithmes précédents (ne sont pas celles qu'on obtiendrait sur une configuration quatre fois plus petite).

2.4 Comparaisons

On observe des différences de performances entre ces algorithmes, malgré le fait qu'ils soient essentiellement les mêmes. Un aperçu de la taille de l'arbre pour chaque algorithme sur diverses configurations est donné ci-dessous (tableau 1).

Par rapport au nombre de nœuds, c'est l'algorithme par groupage qui est le meilleur dans tous les cas, surtout lorsque la taille et le nombre de couleurs augmentent. L'algorithme DLX est meilleur que le parcours en zig-zag sur les petites configurations (moins de 5×5), mais la tendance s'inverse pour les plus grandes configurations.

Par rapport au temps de calcul, DLX devient très vite impraticable comparé aux deux autres algorithmes, ce qui est normal car la réduction vers Exact Cover est conséquente. Aussi, malgré le fait que l'algorithme par groupage est meilleur que le zig-zag au niveau de la taille de l'arbre, c'est le zig-zag qui fonctionne le plus rapidement en pratique. Ce qui rend l'approche par groupage plus lente est le fait que la liste des tuiles à parcourir à chaque étape est beaucoup plus grande qu'avec l'approche par zig-zag.

2.5 Sur le puzzle 16×16 d'origine

Le puzzle d'origine correspond à une instance 16×16 du problème, avec 17 couleurs intérieures et dont les arêtes des tuiles de bord orthogonales au bord ne comportent que 5 couleurs différentes. (Les 5 couleurs de bord sont en fait différentes des autres couleurs dans le puzzle, mais il revient au même de les substituer par 5 des autres couleurs car une tuile intérieure ne pourra jamais être posée au bord, d'où le passage de 22 à 17 couleurs.)

La barrière difficile à dépasser se situe entre 210 et 220 pièces, pour chaque algorithme ; cette barrière n'a pas été dépassée pendant ce stage. Il existe une plateforme de calcul distribué destinée à la résolution d'Eternity II ([1]). Je ne peux malheureusement pas comparer mon meilleur score personnel à celui obtenu grâce à ce site, à cause de la règlementation du jeu qui stipule qu'il est interdit de divulguer son score.

Le nombre de tuiles 2×2 générées par le troisième algorithme est de 1089342 (1312 coins, 72999 bords et 1015031 tuiles intérieures). Une attaque du puzzle par une réitération de cet algorithme pour se ramener à une configuration 4×4 semble peu envisageable, le nombre de tuiles 4×4 étant supérieur à $2 \cdot 10^{14}$.

	DLX	Zig-zag	Groupage
3×3, 3 couleurs	24	36	-
4 couleurs	16	23	-
4×4, 2 couleurs	18474	79090	15214
4×4, 3 couleurs	422	1578	209
4 couleurs	111	203	50
5 couleurs	53	83	20
5×5, 4 couleurs	19320	62218	-
5 couleurs	1292	2659	-
6×6, 6 couleurs	75851	112636	43163
7 couleurs	7762	4063	2077
8 couleurs	3686	1107	335
8 couleurs	968	411	117
8×8, 10 couleurs	trop lent	387114	206123
11 couleurs	trop lent	23992	8819
15 couleurs	12420	863	150
20 couleurs	1608	279	30
10×10, 15 couleurs	trop lent	141085	43281
17 couleurs	trop lent	16357	25358
20 couleurs	trop lent	3475	288
25 couleurs	18941	1005	61
12×12, 20 couleurs	trop lent	236994	57477
25 couleurs	trop lent	13748	1088
30 couleurs	trop lent	4484	92
35 couleurs	trop lent	2180	64
60 couleurs	5839	354	42
16×16, 30 couleurs	trop lent	2152269	237045
35 couleurs	trop lent	358719	4376
45 couleurs	trop lent	57130	144
55 couleurs	trop lent	17309	88

TAB. 1: Valeurs expérimentales du nombre moyen de nœuds de l'arbre de parcours pour chaque algorithme, en fonction de la taille et du nombre de couleurs.

3 Complexité du problème généralisé

Nous allons voir que le problème ETER2 est NP-complet. Ce n'est pas étonnant, au vu des difficultés qu'on a à résoudre des petites instances du problème. La preuve qui suit est tirée de l'article [2], c'est une réduction à partir de 3-partition.

Des variantes du problème ont été étudiées et sont aussi NP-complètes. On peut par exemple interdir les rotations et supprimer les conditions au bord ; voir [6] et [7] pour plus de détails.

Théorème. *Le problème de décision* ETER2 *est NP-complet.*

■ **Preuve**. La vérification de la validité d'une configuration se fait en temps polynomial : il faut que les contraintes locales des $m(n-1)+n(m-1)$ arêtes intérieures soient respectées, ce qui se vérifie en $O(mn)$ comparaisons.

Une instance de 3-partition est un multi-ensemble $A = \{a_1, \ldots, a_{3m}\}$ de $3m$ entiers. La question est l'existence ou non d'une partition de A en triplets telle que la somme de chaque triplet soit égale à

$$S := \frac{1}{m} \sum_{i=1}^{3m} a_i.$$

On peut ajouter la contrainte suivante, qui a pour effet que les seuls sous-ensembles de A dont la somme vaut S sont de cardinal exactement 3 : chaque entier a_i doit être strictement compris entre $\frac{S}{4}$ et $\frac{S}{2}$. Ce problème est NP-complet ([4]).

Réduisons maintenant 3-partition à ETER2. L'idée est d'associer une « barre » (bloc de tuiles de taille $a_i \times 1$) à chaque entier a_i et de paver un rectangle $S \times m$ en forçant les barres à être horizontales. Chaque ligne du rectangle correspond alors à un triplet, ce qui nous donne notre partition. Une 3-partition existe alors si et seulement si le jeu de tuiles admet un pavage, à condition de prendre garde à ce qu'il n'y ait pas de configurations valides en trop.

À chaque nombre a_i sont associées a_i tuiles, qui sont collées ensemble via la couleur i, qui n'apparaît pas ailleurs. Les couleurs extérieures (A et B) servent à forcer les barres à être horizontales, afin que chaque pavage valide corresponde bien à une 3-partition.

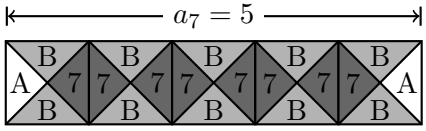


FIG. 4: Exemple d'une représentation d'un nombre a_i par des tuiles.

Le cadre doit entourer un rectangle $S \times m$ et n'a qu'une tuile d'épaisseur, pour un total de $2(S+m)+4$ tuiles. On utilise les couleurs N, W, E, S pour constituer les quatre arêtes, reliées par les coins. Les arêtes intérieureures correspondent à la couleur extérieure des barres (A et B).

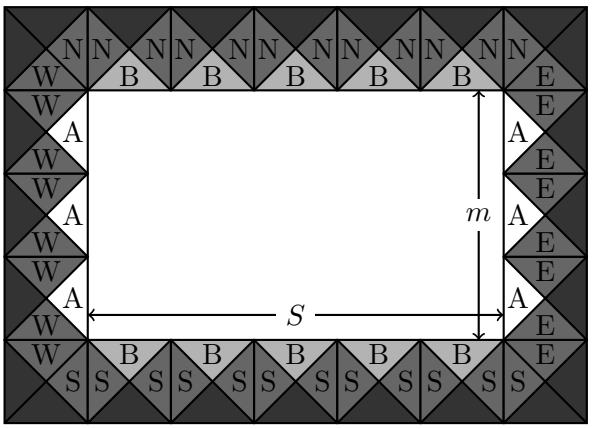


FIG. 5: Les tuiles constituant le cadre.

Cette réduction est bien polynomiale, les $(S+2)(m+2)$ tuiles sur $3m+6$ couleurs étant obtenues directement. □

Ceci n'est qu'un résultat théorique sur une généralisation possible du puzzle Eternity II d'origine. Il ne nous apprend rien de concret sur la difficulté du puzzle ; son intérêt principal étant de nous conforter dans l'idée que ce type de puzzle n'est pas facile dans le cas général.

4 Notion de difficulté

4.1 Vers une définition

Peut-on donner une définition formelle de la difficulté d'une instance ? Les algorithmes de backtrack utilisés pendant ce stage fournissent des arbres de parcours de l'espace des solutions partielles qui contiennent des informations intéressantes sur la difficulté d'une instance.

L'arbre de backtrack, dont la profondeur est le nombre de pièces à placer, est l'arbre des appels de la fonction récursive. Les feuilles de profondeur maximale sont les solutions du puzzle (il suffit de remonter jusqu'à la racine pour avoir la solution) ; les autres feuilles sont les configurations partielles maximales mais pas complètes.

Parmi les instances ayant au moins une solution, trois types d'arbres émergent : les arbres « pyramide » qui ont beaucoup de nœuds et beaucoup de feuilles de profondeur maximale, les arbres « tronc » qui on peu de nœuds et peu de feuilles puis les arbres ayant beaucoup de nœuds pour peu de solutions. Ce sont ces derniers qui correspondent intuitivement à une instance difficile à résoudre : il y a beaucoup de configurations partielles « sans issue », et la recherche d'une solution devra passer par l'exploration de toutes ces configurations.

Définition formelle. Dans l'esprit du paragraphe précédent, on propose la définition suivante, qui dépend de l'algorithme de backtrack A utilisé : la difficulté d'une instance I est donnée par

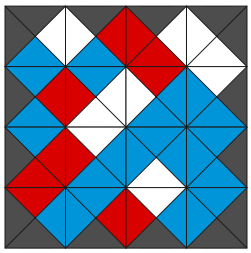
$$\mathrm{diff}_A(I) \;=\; \frac{\text{nombre de nœuds de l'arbre}}{\text{nombre de solutions}}.$$

Il est important de remarquer que cette définition n'est pas intrinsèque, elle dépend de l'algorithme A utilisé. Aussi, diff_A n'est définie que pour les instances admettant au moins une solution. Je n'ai pas trouvé d'instance facile à résoudre par un humain mais ayant une difficulté élevée.

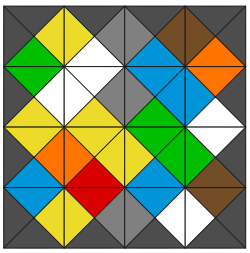
4.2 Recherche d'instances difficiles à résoudre

Comment les concepteurs d'Eternity II ont-ils pu s'assurer de la difficulté de leur puzzle ? Comment ont-ils pu estimer le nombre de solutions à 20000 ?

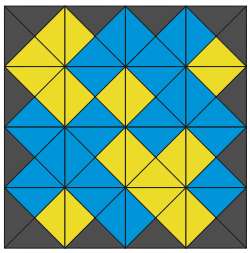
Résoudre des instances 16×16 prend trop de temps, nous allons donc nous faire une idée du comportement de jeux de tuiles plus petits, tirés au hasard. La principale observation que l'on peut faire est l'émergence d'un phénomène de transition de phase. Les instances sur un petit nombre de couleurs (strictement inférieur à n pour une grille $n \times n$) ont beaucoup de solutions et un grand arbre de parcours ; il est aisé d'en trouver une solution mais parcourir tout l'arbre prend beaucoup de temps. En revanche,



(a) instance difficile



(b) instance facile, petit arbre



(c) instance facile, gros arbre

Niveau	(a)	(b)	(c)
0	1	1	1
1	1	1	1
2	2	2	3
3	6	2	15
4	9	3	24
5	33	3	42
6	104	3	175
7	146	2	510
8	80	2	576
9	126	4	1848
10	121	2	3760
11	51	2	3504
12	24	1	4184
13	18	1	4384
14	10	1	2880
15	5	1	1728
16	1	1	1728
Total	738	25363	32

(d) nombre de nœuds par niveau de chaque arbre

FIG. 6: Les trois principaux types d'arbres observés.

le nombre de solutions décroît brutalement lorsque le nombre de couleurs augmente, et les instances deviennent difficiles à résoudre.

Le seuil de transition est, selon les observations, inférieur à n mais il est dur d'en avoir une estimation précise à cause du fait que les instances sur un faible nombre de couleurs sont très longues à résoudre (la transition n'a été observée que sur des configurations 5×5 ou moins). Le tableau 2 ci-dessous donne un aperçu des valeurs de transition de phase.

	2	3	4	5	6	7	8	9
3×4	5	**15**	14	14	14	14	13	13
4×4	6	**36**	30	29	26	24	23	21
4×5	-	59	**137**	76	56	45	43	34
5×5	-	101	**1720**	353	208	132	108	71
5×6	-	-	12084	4505	1426	564	354	177
6×6	-	-	-	369376	16895	3528	1926	623

TAB. 2: Valeurs expérimentales de l'évolution de la difficulté DLX en fonction du nombre de couleurs. En gras, les valeurs « pics » observées.

	record obtenu	difficulté moyenne
4×4, 4 couleurs	931	53
5×5, 4 couleurs	20304	2752
5 couleurs	15209	561
6 couleurs	10832	214
6×6, 6 couleurs	1973183	18904
7×7, 8 couleurs	13866018	156101
8×8, 8 couleurs	488081445	-
8×8, 10 couleurs	67613577	192861
10×10, 14 couleurs	10881092	746794

TAB. 3: Quelques valeurs moyennes et record des difficultés d'instances résolues avec le parcours en zig-zag.

5 Questions connexes

5.1 Isomorphismes entre jeux de tuiles

Comment déterminer si deux jeux de tuiles sont les mêmes à un échange de couleurs près ? Nous allons montrer que cette question est équivalente à la question de savoir si deux graphes donnés sont isomorphes. Le statut de ce problème est particulier car on sait qu'il est dans NP mais on ne sait pas s'il est NP-complet, ni même s'il est dans P.

Graph-isomorphism. Étant donnés deux graphes (E_1, V_1) et (E_2, V_2), on veut déterminer l'existence d'une application bijective $f : V_1 \longrightarrow V_2$ telle que pour tous $u, v \in V_1$,

$$uv \in E_1 \iff f(u)f(v) \in E_2.$$

Tileset-isomorphism. Étant donnés deux jeux de tuiles T_1 et T_2 (et leurs jeux de couleurs respectifs C_1 et C_2), on veut déterminer l'existence d'une application bijective $f : C_1 \longrightarrow C_2$ telle que

$$T_1 = f(T_2) = \{(f(c_1), f(c_2), f(c_3), f(c_4)) : (c_1, c_2, c_3, c_4) \in T_2\}.$$

L'égalité entre les tuiles se fait à rotation près et les jeux de tuiles ne sont pas forcément des instances d'ETER2.

Réductions. On réduit graph-isomorphism à tileset-isomorphism en codant la liste d'adjacence du graphe de départ avec des tuiles : à chaque nœud i correspond des tuiles contenant : la couleur N (pour l'orientation), l'entier i directement à droite de la couleur N, puis les entiers correspondant aux voisins du nœud i. À un seul nœud peuvent correspondre plusieurs tuiles si le nombre de voisins est supérieur à 2. La couleur x est utile lorsqu'il n'y a plus de voisins mais qu'il reste une arête d'une tuile à laquelle aucune couleur n'est assignée.

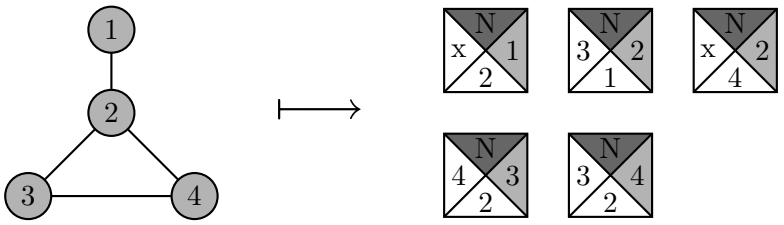


FIG. 7: Réduction de graph-isomorphism à tileset-isomorphism.

Comme les jeux de tuiles crées représentent exactement des listes d'adjacences de graphes, l'isomorphisme entre les graphes entraine l'isomorphisme entre les jeux de tuiles, et inversement.

L'autre sens est similaire : on code chaque tuile par un graphe « carré orienté » : les quatre coins codent les couleurs de la tuile et les lettres N et E servent à preserver l'orientation.

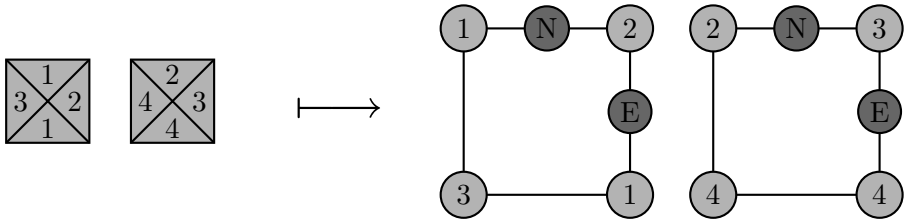


FIG. 8: Réduction de tileset-isomorphism à graph-isomorphism.

Deux tuiles isomorphes induisent deux graphes isomorphes, par construction. L'autre sens est presque aussi direct, mais il faut prendre garde au fait que deux tuiles ne peuvent pas être retournées, d'où l'utilité des couleurs N et E qui préservent l'orientation des tuiles.

Les deux réduction se font en temps polynomial : la première en temps proportionnel au nombre de sommets du graphe et la deuxième en temps proportionnel au nombre de tuiles.

5.2 Nombre de couleurs et configurations uniques

À taille fixée, quel est le nombre minimal de couleurs tel qu'il existe un jeu de tuiles sur ces couleurs ayant une unique configuration ?

Pour une configuration $m \times n$, ce nombre est inférieur ou égal à $m+n-2$: en assignant une couleur aux colonnes et aux lignes intérieures (comme sur la figure ci-dessous), on obtient un jeu de tuiles n'ayant qu'une seule solution, sur $m + n - 2$ couleurs intérieures.

Cette construction est valide ; on raisonne par récurrence. Fixer un coin impose le placement unique des tuiles sur la ligne et la colonne où le coin apparaît, à cause des bords. On obtient alors un pavage unique du contour de la grille, en plaçant le coin qui reste. On peut réitérer ce raisonnement jusqu'à ce que la configuration soit remplie car chaque étape fait apparaître un « nouvelle couleur de bord » à l'intérieur de la figure, qui est représentée par les quatre couleurs n'apparaissant que sur le contour de la grille intérieure.

Une autre construction permet de passer à $\max(m, n)$ couleurs : on colorie les $m - 1$ colonnes intérieures avec des couleurs distinctes, puis la même chose avec les $n - 1$ lignes intérieures (on peut utiliser le même ensemble de couleurs pour les lignes et les colonnes). En coloriant la diagonale d'une certaine manière, on obtient des configurations uniques, au moins pour les instances de taille inférieure à 10×10. L'appendice B contient un exemple d'une construction 10×8 utilisant ce procédé (figure 10).

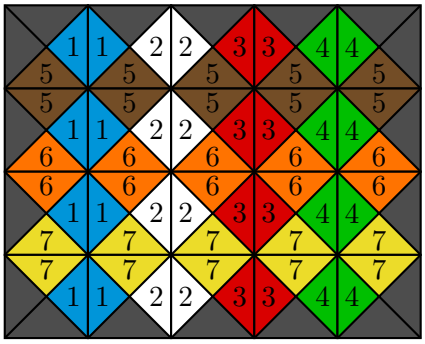


FIG. 9: Codage d'une configuration unique avec $m + n - 2$ couleurs.

Le tableau ci-dessous donne des bornes supérieures. Elles sont très probablement plus élevées que la vraie valeur, mais encore une fois le temps d'exploration d'une instance sur peu de couleurs nous empêche d'avoir des valeurs plus précises.

×	3	4	5	6	7	8	9	10
3	3	3	3	3	4	4	4	5
4	-	3	4	4	5	5	6	6
5	-	-	4	5	5	6	7	7
6	-	-	-	5	6	7	8	8
7	-	-	-	-	7	8	9	10
8	-	-	-	-	-	8	9	10
9	-	-	-	-	-	-	9	10
10	-	-	-	-	-	-	-	10

TAB. 4: Bornes supérieures du nombre minimal de couleurs nécéssaire pour coder une configuration unique, en fonction de la taille $m \times n$ de l'instance. Valeurs obtenues expérimentalement.

Une borne inférieure peut être obtenue en remarquant que le nombre de tuiles distinctes qu'il est possible de faire avec c couleurs est de $c + 4\binom{c}{2} + 6\binom{c}{3} + 6\binom{c}{4}$ tuiles intérieures, $c + 6\binom{c}{2} + 6\binom{c}{3}$, et toujours 4 coins à partir de 2 couleurs. Le tableau ci-dessous donne les premières bornes inférieures obtenues en comptant le nombre de bords et de tuiles intérieures (qui doivent être distincts pour préserver l'unicité de la configuration).

taille	$\leqslant 4 \times 4$	5×5 à 6×6	7×7 à 9×9	10×10 à 13×13
couleurs	2	3	4	5

TAB. 5: Bornes inférieures du nombre minimal de couleurs nécéssaire pour coder une configuration unique.

Liens avec le seuil de difficulté. Le pic de difficulté observé dans la section précédente s'accompagne d'une décroissance brutale du nombre de solutions. (On passse par exemple d'une moyenne de 7000 à 40 solutions lorsque le nombre de couleurs passe de 2 à 3 sur les configurations 4×4.)

On est donc tenté de conjecturer que le nombre de couleurs nécéssaires à coder une configuration unique est le même que le seuil à partir duquel la difficulté augmente d'un coup. Avoir une formule exacte ou même un équivalent asymptotique serait très intéressant, mais il est difficile de prouver que ce soit sur ce genre de questions.

6 Conclusion

Le travail effectué pendant ce stage est celui présenté dans ce document, à l'exception de la réduction (et de l'implémentation) de la résolution d'instances par Set-Cover (faites par Nicolas Ollinger), et des autres résultats dont la provenance est explicitement mentionnée.

Plusieurs questions intéressantes restent en suspens : à partir de combien de couleurs la transition de phase a-t-elle lieu ? (Et l'apparition de configurations uniques ?) Le fait de savoir qu'il existe une solution nous aide-t-il à résoudre une instance ? Quelle est la difficulté du puzzle 16×16 d'origine ?

Je tiens à remercier l'ensemble de l'équipe Escape, mes maîtres de stage, mes co-bureaux stagiaires ainsi que le personnel du LIF pour l'accueil chaleureux qui m'a été réservé, pour les discussions enrichissantes et pour les moyens matériels mis à ma disposition (ceci incluant un bureau, un ordinateur, un puzzle Eternity II, des glaces et du café).

Références

[1] *Site non-officiel français sur le jeu Eternity II*, `http://www.eternity2.fr/`.

[2] Erik D. Demaine and Martin L. Demaine, *Jigsaw puzzles, edge matching, and polyomino packing : Connections and complexity*, Graphs and Combinatorics **23 (supplement)** (2007), 195–208.

[3] Jean-Christophe Filliâtre, *DLX : Implantation de l'algorithme de D. Knuth Dancing Links*, `http://www.lri.fr/~filliatr/software.fr.html`.

[4] Michael. R. Garey and David S. Johnson, *Computers and intractability : A guide to the theory of np-completeness*, W. H. Freeman, 1979.

[5] Donald E. Knuth, *Dancing Links*, `http://arxiv.org/abs/cs/0011047`.

[6] Ville Lukkarila, *The square tiling problem is NP-complete for deterministic tile sets*, Tech. Report 754, TUCS, Mar 2006.

[7] Yasuhiko Takenaga and Toby Walsh, *Tetravex is np-complete*, Inf. Process. Lett. **99** (2006), no. 5, 171–174.

A Quelques programmes

Les implémentations en OCaml des algorithmes vus ci-dessus sont disponibles à l'adresse suivante : `http://timo.jolivet.free.fr/eter2/`. On y trouve :

- `solver` : un logiciel de résolution d'une instance donnée par l'utilisateur ;
- `dlxbenchmark`, `brutebenchmark`, `groupbenchmark` : permettent de tester et de comparer les différentes méthodes de résolution (comme par exemple de rechercher des instances à difficulté élevée ou d'étudier le comportement asymptotique du problème) ;
- `render` : un logiciel de rendu graphique de configurations valides ; le format de sortie est du code Ti*k*Z (qui a servi à faire les figures de ce rapport).

B Quelques images, en couleurs

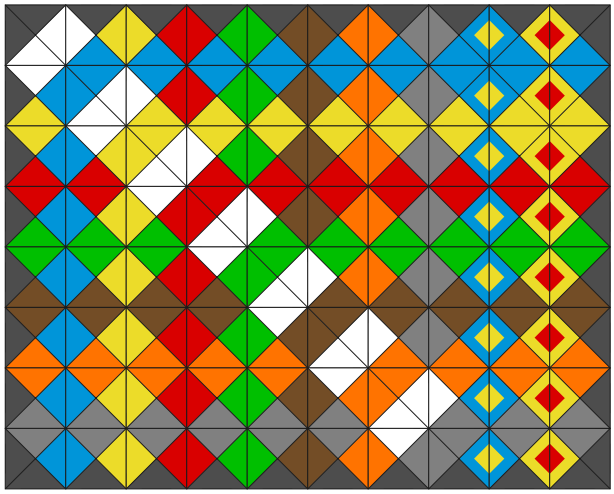


FIG. 10: Codage d'une configuration 10×8 unique avec 10 couleurs, selon la deuxième construction décrite.

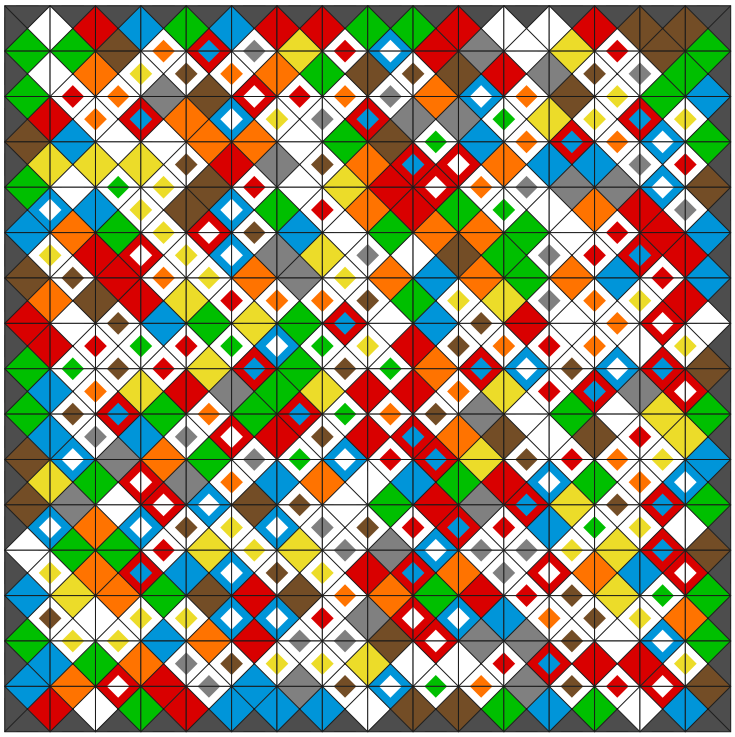
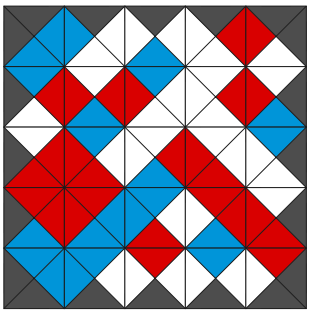
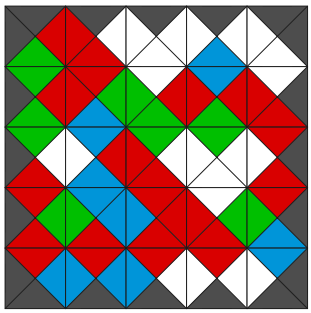


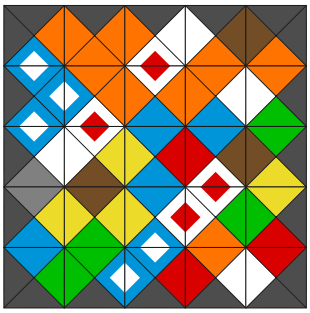
FIG. 11: Une instance similaire au puzzle Eternity II : 16×16, 17 couleurs intérieures et 5 couleurs de bordure ; probablement aussi difficile.



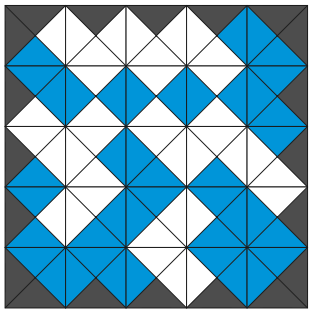
(a) 5×5, 3 couleurs, 6272 solutions, 4032797 nœuds



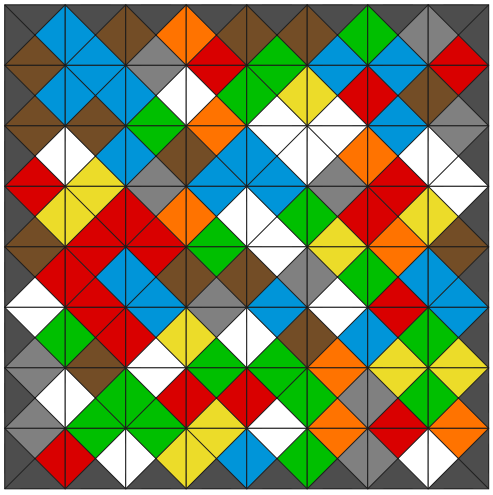
(b) 5×5, 4 couleurs, 1 solutions, 13638 nœuds



(c) 5×5, 10 couleurs, 1 solutions, 69 nœuds

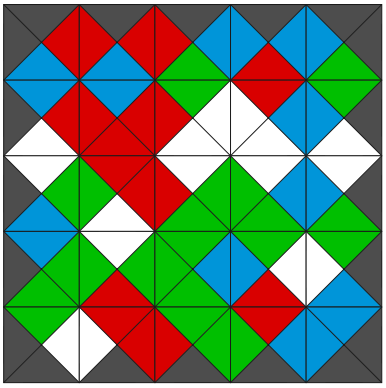


(d) 5×5, 2 couleurs, déjà trop long à résoudre

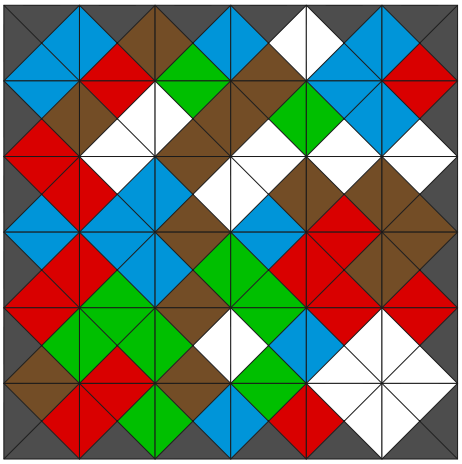


(e) 8×8, 8 couleurs, 2 solutions, 976162890 nœuds (!)

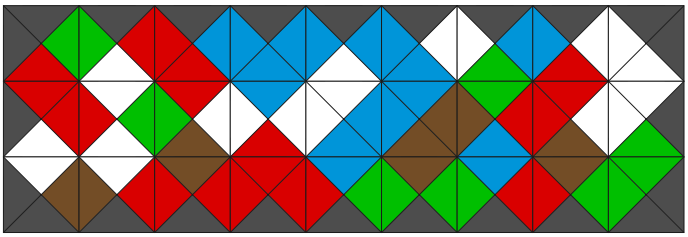
FIG. 12: Comparaison d'instances (algorithme zig-zag). L'écart du nombre de solutions entre (a) et (b) est intéressant.



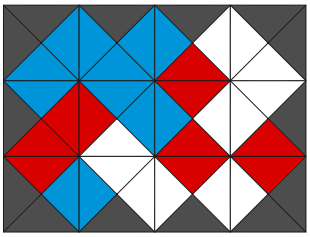
(a) 5×5, 4 couleurs



(b) 6×6, 5 couleurs



(c) 9×3, 5 couleurs



(d) 4×3, 3 couleurs

FIG. 13: Des configurations à solution unique.

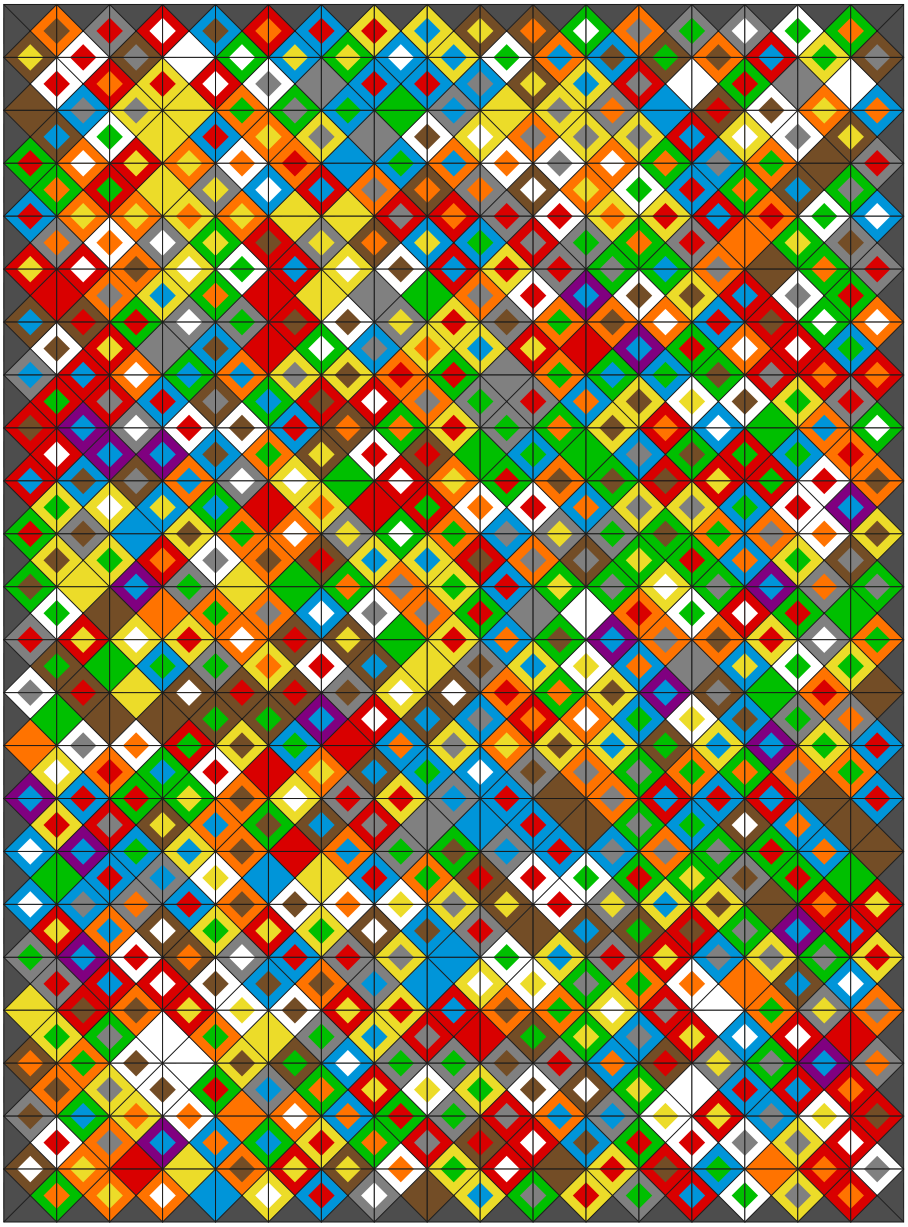


FIG. 14: Un joli tapis à 64 couleurs (et à solution unique).