INF6102 - Métaheuristiques appliquées au génie informatique

Hiver 2025

Projet
Eternity II

Auriane Peter–Hemon - 2310513
Félix Lamarche - 2077446

16 avril 2025

Table des matières

1 Introduction

Le problème Eternity II est un jeu de plateau dont le but est de placer chacune des tuiles carrées composées de quatre couleurs différentes tel que chacune ait ses arêtes en contact avec une arête d'une autre tuile de la même couleur. Le puzzle original est composé d'un tableau de 16x16 tuiles et n'a toujours pas été résolu par quiconque. Alors, nous allons plutôt tenter d'obtenir une solution ayant le moins de conflits : le nombre d'arêtes en contact avec deux couleurs différentes.
L'espace de recherche de ce problème, selon la taille n d'un plateau $N \times N$, est de $4^{N^2} \times (N^2)!$, puisqu'il y a au total N^2 tuiles pouvant chacune être orientées de 4 façons différentes pour 4^{N^2} agencements de rotations différentes et $(N^2)!$ emplacements différents pour les tuiles. Une recherche exhaustive se déroulerait en temps non polynomial et ne serait pas possible pour des instances non triviales. De plus, ce problème est symétrique et comporte quatre solutions différentes, car la première pièce placée a quatre rotations différentes possibles. Ainsi, une solution valide peut être rotater pour produire une autre solution valide.

2 Solveur heuristique

Dans un premier temps, nous avons mis en place le solveur heuristique simple suivant : on place les pièces une par une et, pour chaque position sur le plateau, on choisit parmi toutes les pièces restantes celle qui minimise le nombre de conflits additionnels avec les pièces déjà posées. Si plusieurs pièces minimisent ce nombre de conflits créés, une pièce est choisie aléatoirement d'une façon uniforme parmi celles-ci.
On remplit le plateau en suivant l'ordre d'une spirale, en commençant par le coin en bas à gauche. De cette manière, les bords sont remplis avant l'intérieur pour prioriser le placement des pièces de coins et de bords pour éviter de placer des pièces d'intérieur à ces emplacements.
De plus, pour s'assurer que les pièces de bords soient placées sur les bords et que les pièces de coin soient placées dans les coins, on ajoute une forte pénalité si un type de pièce est tenté d'être placé dans un emplacement impossible (par exemple, une pièce de l'intérieur dans un bord).

13	12	11	10	9
14	23	22	21	8
15	24	25	20	7
16	17	18	19	6
1	2	3	4	5

FIGURE 1 – Ordre de placement des pièces pour une instance de taille 5×5.

3 Solveur de recherche locale

Pour améliorer les solutions obtenues par notre heristique, on développe une méthode de recherche locale. Pour cela, on reprend l'heuristique d'échange 2-opt.

La modélisation est la suivante :
- **Solution initiale :** solution obtenue grâce à notre solveur heuristique décrit précedemment.
- **Voisinage :** Les voisins d'une solution sont les solutions où deux pièces ont été échangées et rotatées (cela peut être une pièce échangée et rotatée avec elle-même).
- **Évaluation** : La valeur de chaque voisin correspond au nombre de conflits restants dans la solution.
- **Sélection** : On sélectionne le meilleur voisin améliorant, si plusieurs ont la même valeur, une sélection aléatoire suivant une distribution uniforme est utilisée.
- **Critère d'arrêt** : On s'arrête lorsqu'il n'y a plus de voisin améliorant.

Taille du voisinage : On teste toutes les permutations et tous les échanges possibles. Le cardinal du voisinage est donc :

$$card(V) = O(N^4)$$

Le voisinage est connecté, puisqu'à partir de n'importe quelle solution, il est possible d'atteindre la solution optimale. En effet, pour pour chacune des pièces , on peut l'échanger avec la tuile se retrouvant à sa position et à sa rotation approprié dans la solution faisable avec l'échange 2-opt.

La recherche est incomplète et perturbative. La présence de conflits est une contrainte molle, et les pièces étant toutes utilisées est une contrainte dure.
En pratique, pour le voisinage, au lieu d'évaluer chaque paire de pièces avant de faire une sélection et faire un échange, chacun des emplacements des pièces sont choisis les uns après les autres d'une façon aléatoire, et le meilleur échange pour cet emplacement est choisi, si cet échange diminue le nombre de conflits existant. Ainsi, explorer le voisinage est beaucoup plus efficace et ajoute un peu de diversité avec les choix aléatoires. Cependant, la recherche locale ne s'arrête que s'il n'y a plus d'échanges diminuant le nombre de conflits dans l'entièreté du voisinage.

De plus, on ajoute un mécanisme de redémarrage. Après avoir atteint un minimum local lors de la recherche locale et tant que le temps d'exécution n'est pas terminé, on génère une nouvelle solution initiale à partir de notre solveur heuristique, qui a de l'aléatoire, pour redémarrer une nouvelle recherche locale, et ce tant qu'il y a du temps d'exécution restant ou jusqu'à ce qu'une solution faisable soit trouvée.

4 Solveur avancé

4.1 Recherche locale itérée

Ce solveur avancé se base sur l'algorithme de recherche locale précédemment expliqué en y ajoutant des fonctionnalités.

La modélisation est la suivante :
- **Solution initiale :** Meilleure solution obtenue du solveur heuristique dans un temps limite.
- **Voisinage :** Swap k-opt séquentiel avec rotations des pièces.
- **Évaluation** : La valeur de chaque voisin correspond au nombre de conflits restants dans la solution.
- **Sélection** : On sélectionne le meilleur voisin améliorant parmi la suite d'échange séquentiel de k-opt
- **Critère d'arrêt** : On s'arrête lorsqu'il n'y a plus de voisin améliorant.

Taille du voisinage : On teste toutes les permutations et tous les échanges possibles pour chaque échange successif de pièces. La taille asymptotique du voisinage pour un échange de k pièces est :

$$card(V) = O(N^{2k})$$

Le choix de la solution initiale se trouve être la solution heuristique générée ayant le moins de conflits. Notre heuristique a une portion d'aléatoire ce qui permet de générer plusieurs solutions initiales différentes. Ainsi, nous pouvons chercher la meilleure solution initiale selon un temps de recherche dicté.

Le voisinage de cette recherche est connecté en suivant le même raisonnement décrit précedemment pour l'algorithme de recherche locale. La recherche est incomplète et perturbative. La présence de chaque pièce dans la solution est une contrainte dure et la présence de conflits est une contrainte molle.

Le voisinage est un échange de k pièces successives tel que la pièce P_1 choisie à la position (i,j) sera échangé de position avec la pièce P_2 réduisant le plus de conflits pour cette pièce P_1. Si plusieurs pièces réduisent le même nombre de conflits pour cette pièce P_1, l'échange permettant réduisant le plus de conflits pour la pièce P_2 sera sélectionné. Dans le cas de plusieurs choix égaux, un échange aléatoire selon une distribution normale sera effectué parmi ceux-ci. L'échange valorise la réduction du nombre de conflits de la pièce P_1, puisque l'échange successif échangera P_2 avec une nouvelle pièce, si un échange bénéfique existe. Ainsi, la réduction du nombre de conflits pour la pièce P_2 est moins significatif. Les échanges successifs continuent

jusqu'à ce qu'aucun échange améliorant n'existe pour la dernière pièce P_n. Puis, si un nombre d'échanges intermédiaires de cet échange successif réduisait davantage le nombre de conflits que le k-opt fait maximalement, les échanges augmentant le nombre de conflits sont inversés afin de retourner la solution réduisant au maximum le nombre total de conflits. Également, pour éviter des cycles, les échanges successifs ne peuvent échanger de positions avec une pièce ayant déjà été déplacée dans ce voisinage.

Une fois qu'un minimum local est atteint, il y a diverses façons de tenter d'en sortir. Il y a un système d'ILS (recherche locale itérée) qui consiste en une perturbation de la meilleure solution courante. La perturbation se fait selon un échange aléatoire d'un nombre de pièces aléatoires. Un nombre de pièces se trouvant entre 5% et 50% du nombre de pièces totales sera sélectionné aléatoirement selon une distribution uniforme. Parmi celles-ci un nombre de pièces ayant des conflits et un nombre n'ayant pas de conflits seront sélectionnées. Le ratio de pièces ayant des conflits est choisi aléatoirement dans l'intervale de $[0.20, 0.50]$ d'une façon uniforme. S'il n'y a pas assez de pièces à conflit, des pièces n'ayant pas de conflits seront plutôt choisies. Ce ratio existe afin de diversifier la perturbation tout en s'assurant de perturber des pièces ayant des conflits, sinon la perturbation pourrait facilement mener à la même solution après la recherche locale. Les ratios aléatoires ont été choisis expérimentalement après avoir observé de bons résultats parmi ces intervalles, et le choix aléatoire permet de diversifier la perturbation. La perturbation consiste à l'échange de positions des différentes pièces choisies telles que la pièce P_1 se retrouve à la position de la pièce P_2 et ainsi de suite.

De plus, un mécasnime de redémarrage existe. S'il n'y a pas eu d'améliorations après un certain nombre d'itérations, la recherche va redémarrer. Ce redémarrage a 50% de chance d'essayer une nouvelle recherche à partir d'une solution initiale de l'heuristique et a 50 % de chance de redémarrer à partir de la meilleure solution trouvée jusqu'à présent. Ainsi, il y a de la diversification et de l'intensification en générant de nouvelles solutions initiales sans complètement abandonner la meilleure solution trouvée jusqu'à présent.

4.2 Large Neighborhood Search

Nous avons essayé d'implémenter un Largee Neighboorhood Search. Les paramètres utilisés sont les suivants :

- **Fonction de destruction :** pour intensifier la recherche, on détruit uniquement des cases ayant des conflits.
- **Pourcentage de destruction :** on commence avec un pourcentage de destruction de 30% qui diminue progressivement au fur et à mesure que la solution s'améliore.
- **Fonction de réparation :** on répare la solution en utilisant la même heuristique que précédemment : pour chaque emplacement vide, on sélectionne de manière aléatoire une des pièces induisant le moins de conflits.

De même, un système de redémarrage existe après un certain nombre d'itéraitions sans amélioration. La nouvelle solution initiale a 50% de chances d'être la meilleure solution obtenue et 50% de chances d'être une solution aléatoire.

5 Résultats

On obtient les résultats suivants :

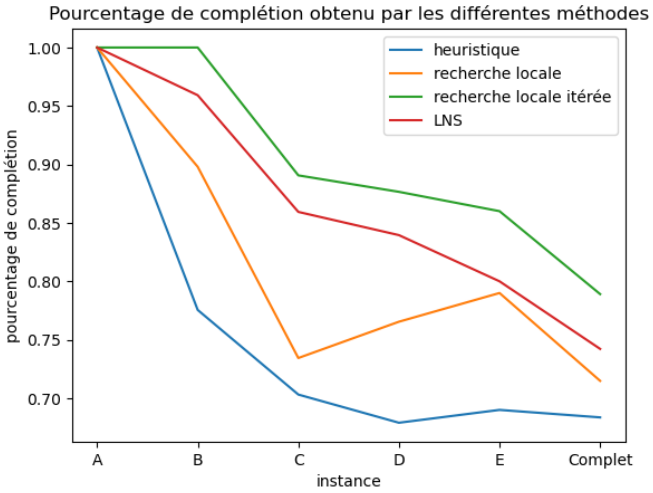

FIGURE 2 – Meilleurs résultats obtenus pour chaque méthode

Les meilleurs résultats sont obtenus par la méthode de recherche locale itérée. La méthode LNS s'avère assez décevante : si elle permet d'obtenir de meilleurs résultats qu'une simple recherche locale, elle est cependant moins efficace qu'une recherche locale itérée. Il est probable que nous ayons mal géré le compromis entre intensification et diversification, ce qui nous amène à rester coincé dans des minima locaux.

Les résultats détaillés pour chaque méthode sont les suivants tels que les résultats sont le nombre de conflits :

Instances	**A**	**B**	**C**	**D**	**E**	**Complet**
Meilleur résultat	0	11	19	26	31	81
Résultat moyen	4.12	17	27.42	34.48	38.52	97.66
Ecart type	4.85	2.99	3.48	4.06	3.74	7.09
Tps d'exécution moyen (s)	2.1e-3	1.6e-2	0.03	0.04	0.06	0.40

TABLE 1 – Résultats solveur heuristique (50 itérations)

Instances	**A**	**B**	**C**	**D**	**E**	**Complet**
Meilleur résultat	0	5	17	19	21	73
Résultat moyen	0	12.07	26.19	29.71	36.91	105.41
Ecart type	0	3.13	3.58	3.77	4.35	7.27
Nb restart	0	1809	967	572	341	39

TABLE 2 – Résultats recherche locale (10 min)

Instances	**A**	**B**	**C**	**D**	**E**	**Complet**
Meilleur résultat	0	0	7	10	14	54
Temps pour l'obtenir	0	2401.90	2228.35	1379.88	1920.83	3547.39

TABLE 3 – Résultats ILS (1h)

Instances	**A**	**B**	**C**	**D**	**E**	**Complet**
Meilleur résultat	0	2	9	13	20	66
Temps pour l'obtenir (s)	0.0	335.91	300.05	2418.77	263.90	1270.94

TABLE 4 – Résultats LNS (1h)

6 Analyse de complexité

Les complexités sont selon N, la taille d'un tableau de $N \times N$.

Obtenir le nombre de conflits pour le placement d'une pièce :
`get_conflict_count(solution, x, y, piece)` : $O(1)$.

Échanger deux pièces d'emplacement :
`swap_pieces(x1, y1, piece1, x2, y2, piece2` : $O(1)$.

Solveur heuristique :
`solve_heuristic(puzzle)` : $O(N^4)$.
Pour chaque position sur le tableau, chaque conflit de chaque rotation de chaque pièce est évaluée afin d'obtenir les meilleurs choix de pièces.

Obtenir le meilleur choix d'échange pour une pièce :
`get_best_swap(x, y, piece)` : $O(N^2)$.
Pour obtenir un des échanges réduisant le plus le nombre de conflits, chaque pièce est évalué, soit N^2 échanges possibles.

Résoudre le problème selon une recherche locale simple :
`solve(puzzle)` : $O(N^4)$.
Chacune des N^2 pièces doit rechercher son meilleur échange avec `get_best_swap()` et cela se produit tant qu'il y a des échanges pouvant réduire le nombre de conflits présents.

Faire un échange de k-opt :
`do_lkh_swap(x, y, piece)` : $O(N^4)$.
Pour trouver l'échange optimal d'une pièce, c'est un temps $O(N^2)$ et il y a jusqu'à N^2 pièces pour des échanges successifs.

Résoudre le problème selon l'algorithme de recherche locale avancée :
`solve_advanced(puzzle)` : $O(N^4)$.
Pour chacune des pièces, il y a une possibilité d'appeler `do_lkh_swap()` afin de faire des échanges, cependant plus la fonction fait d'échanges, moins d'échanges futurs seront faits. En effet, si la pièce a bien été placée lors du premier appel de `do_lkh_swap()`, à son appel, elle ne fera pas d'échange et ne bougera donc pas. La complexité asymptotiques de cette fonction reste donc de l'ordre de $O(N^4)$.

Détruire une partie de la solution : `destroy_solution_conflicts_only(puzzle)` : $O(N^2)$.
Pour évoir les pièces en conflit, on appelle `get_conflict_pieces(puzzle)` qui itère sur toutes les pièces pour savoir si elles sont en conflit

Réparer la solution : `destroy_solution_conflicts_only(puzzle)` : $O(N^4)$.
Idem que le solveur heuristique, mais on évalue moins de pièces.

Résoudre le problème selon l'algorithme LNS :
`solve_lns_(puzzle)` : $O(N^4)$.
On détruit puis on répare la solution