POLYTECHNIQUE MONTRÉAL

INF6102 - Métaheuristique

Projet - Eternity II

Signatures

Auteur(e)s	*Matricule*
Léo Valette	2307835
Armel Ngounou Tchawe	2238017

18 avril 2024

Table des matières

Liste des figures

Liste des tableaux

Liste des algorithmes

1 Introduction

Ce projet a pour but de résoudre, ou du moins de trouver la meilleure solution possible, à un puzzle nommé **Eternity II** [1].

La version originale de ce puzzle comprend 256 pièces divisées en quatre zones de couleurs. Les bords de deux pièces adjacentes doivent avoir la même couleur, sinon cela génère un conflit. Nous avons à notre disposition cinq instances réduites en plus du puzzle original, l'image 1.1 représente une de ces instances réduite ne comportant aucun conflit.

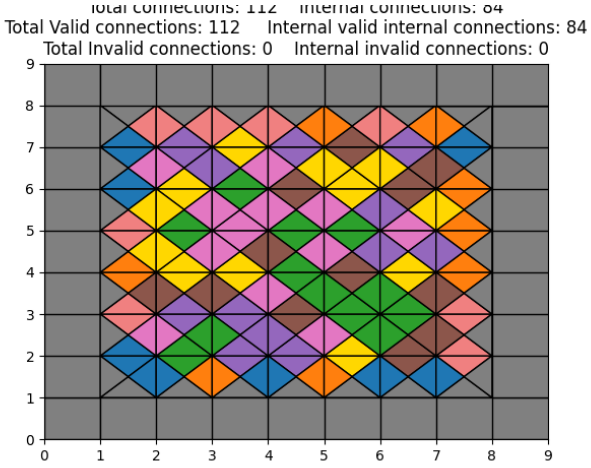

FIGURE 1.1 – Instance eternity_B résolue (7×7)

Pour une instance de taille T ($T \times T$ pièces), la taille de l'espace de recherche est $(\mathbf{T^2})! * \mathbf{4^{T^2}}$, soit pour $T = 16$ à peu près 1.15×10^{661} possibilités. Il est donc irréaliste d'utiliser des méthodes "force brute" pour résoudre le problème.

Le projet se divise en 3 phases. D'abord, nous avons implémenté un solveur se reposant uniquement sur une heuristique. Ensuite un solveur utilisant une recherche locale. Enfin, un solveur utilisant une/des métaheuristiques. Dans une dernière partie, nous analyserons les performances de nos différents modèles.

2 Solveur heuristique

Notre algorithme pour cette partie est très simple, nous plaçons les pièces une par une en choisissant celle qui génère le moins de conflit parmi celles disponibles. Nous testons donc toutes les rotations de chaque pièce restante à chaque étape de l'algorithme. En cas d'égalité, nous en choisissons une aléatoire parmi les meilleures. Cet algorithme a l'avantage d'être facile à implémenter et à améliorer.

12	19	20	21	22	23	18
11	38	39	43	46	48	17
10	33	37	40	44	47	16
9	29	32	36	41	45	15
8	26	28	31	35	42	14
7	24	25	27	30	34	13
0	1	2	3	4	5	6

FIGURE 2.1 – Ordre de placement des pièces pour une instance de taille réduite (7×7).

Au fur et à mesure de nos tests, nous avons implémenté plusieurs algorithmes en modifiant l'ordre de placement des pièces, car nous nous sommes rendu compte que les côtés devaient être placés en premier afin d'utiliser les pièces avec la couleur grise. Nous avons aussi ajouté à notre heuristique une forte pénalité en cas de conflit impliquant la couleur grise et/ou les bords. Le pseudocode de l'heuristique finale et du solveur final sont disponibles dans la partie Algorithmes.

L'ordre de placement final est visible dans la figure 2.1, on remarque que l'on place d'abord les bords avant de suivre les diagonales. Le but est d'utiliser au maximum les pièces déjà placées pour placer les nouvelles.

La figure 2.2 présente l'évolution des performances temporelles et de score des différentes versions de l'algorithme et de l'heuristique. Nous avons choisi les dernières versions, car elles génèrent de meilleurs résultats en moyenne en temps qui reste raisonnable, sachant que cette fonction ne sera pas appelée souvent.
On remarque que la version SolveurV1 + HeuristiqueV2 a trouvé une solution équivalente à la dernière version malgré un score moyen moindre, cela montre une plus grande diversité dans les solutions générées.

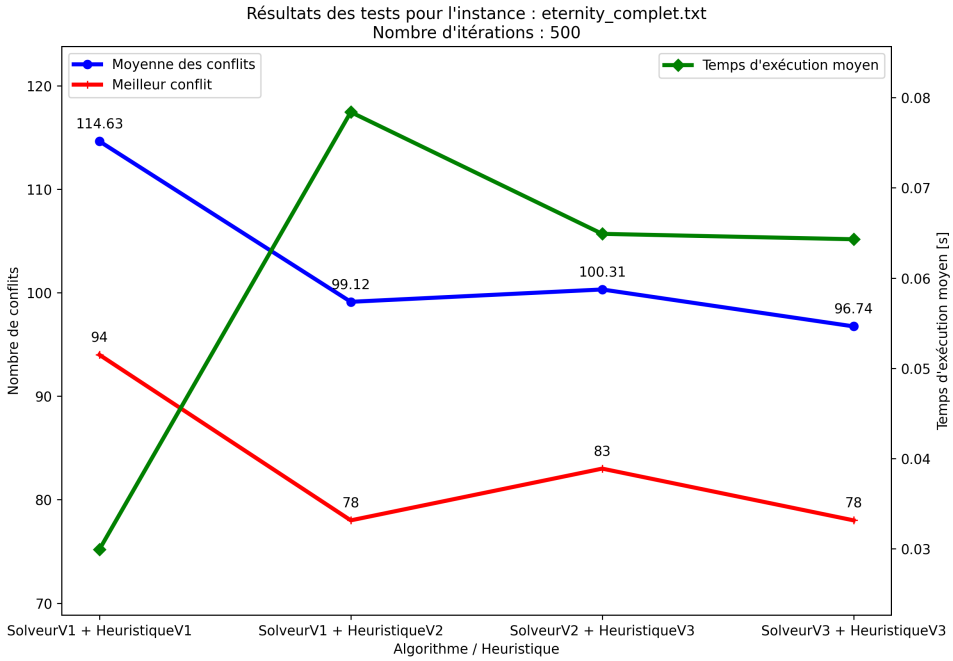

FIGURE 2.2 – Évolution des performances des différents algorithmes et heuristiques implémentés.

Pour ce qui est de la complexité de cet algorithme, elle dépend de plusieurs choses. Premièrement, l'heuristique parcourt chaque bord de la pièce à tester, donc une complexité constante ($O(4)$). Ensuite, pour chaque emplacement, on va choisir une pièce parmi les pièces restantes, on remarque que le nombre de pièces à évaluer décroit selon une suite arithmétique. La complexité du Solveur heuristique pour une instance de taille T ($T \times T$ pièces) est donc la suivante :

$$\begin{aligned} & O[4 \times (T^2 + (T^2 - 1) + (T^2 - 2)...1)] \\ = \quad & O(4 \times \frac{T^2 \times (T^2 + 1)}{2}) \\ \text{donc au final :} \quad & \mathbf{O(T^4)} \end{aligned} \tag{2.1}$$

Les résultats obtenus sur les différentes instances du problème sont disponibles dans la partie 6.1. On arrive à obtenir une solution valide pour l'instance A dans la plupart des essais. Pour les autres instances, nous n'avons pas trouvé de solution valide.

Les algorithmes conçus dans cette partie seront réutilisés dans la suite du projet afin de générer des solutions variées et de qualité.

3 Solveur recherche locale

Pour la partie recherche locale, nous avons d'abord choisi d'implémenter un recuit simulé. Nos tentatives se sont montrées infructueuses et nous avons finalement choisi un hill climbing avec restart. Nous ne détaillerons donc pas le recuit simulé, bien que le code soit disponible dans les fichiers joints au rapport.

3.1 Voisinage

La grande partie des efforts fournis pour ce solveur se trouve dans la conception de bons voisinages. Nous voulions un voisinage assez large pour avoir une bonne diversification, ce qui ralenti malheureusement la recherche pour les "grandes" instances.

Nous avons d'abord implémenté un voisinage permettant d'échanger 2 pièces de places tout en choisissant la meilleure rotation pour chacune de ces pièces. Comme cela était beaucoup trop lent, nous avons choisi de ne permettre le swap seulement si au moins une des deux pièces est en conflit. L'idée est que l'on veut se concentrer sur les pièces problématiques et que le swap de 2 pièces sans conflit à plus de chance d'en générer de nouveaux que d'améliorer la solution.

La dernière et meilleure version de ce voisinage comporte plus de restrictions, notamment liées aux pièces comportant du gris. On ne permet le déplacement d'une pièce d'angle (avec deux côtés gris) que dans un angle avec la bonne rotation. De même pour les pièces de bords (avec un côté gris), on ne permet que les déplacements vers un bord et on génère seulement la bonne rotation.

Ces améliorations ont permis de réduire la taille des voisinages générés tous en augmentant la proportion de voisins améliorants, comme montré dans la figure 3.1. Cela a permis d'accélérer la recherche par un facteur 10.

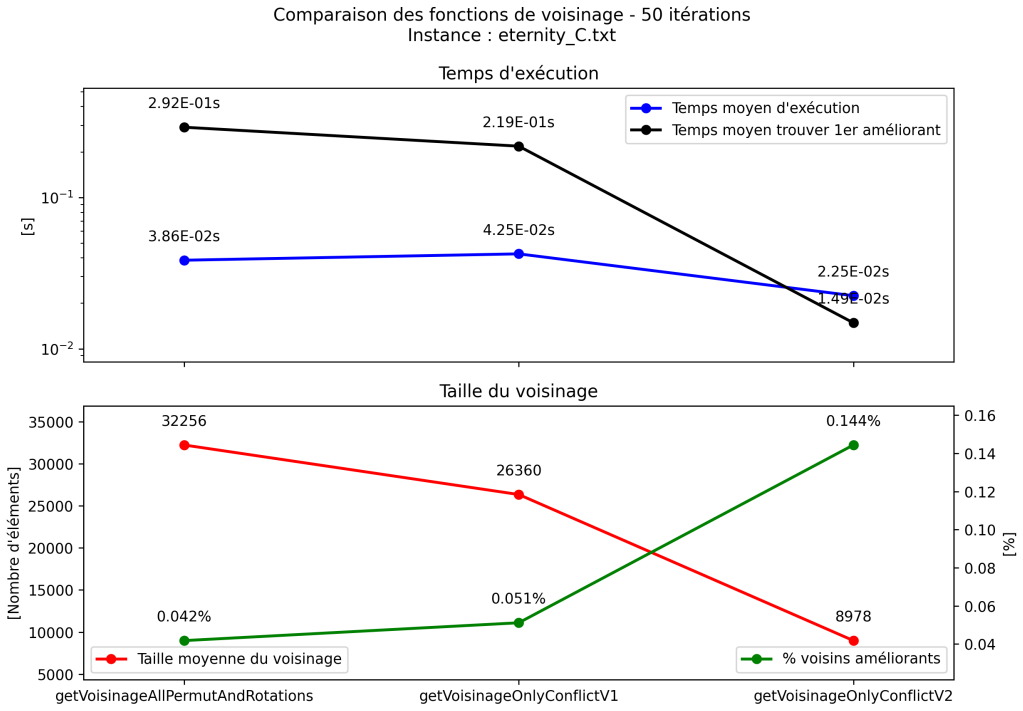

FIGURE 3.1 – Comparaison des différentes fonctions de voisinages implémentées.

On remarque aussi que la proportion de voisins améliorants est très faible, ce qui explique pourquoi le recuit simulé n'a pas fonctionné. En effet, la probabilité de sélectionner aléatoirement un voisin améliorant est quasi nulle. Pour inciter la sélection de bons voisins, il faut abaisser la température très proche de 0 et donc se rapprocher d'un hill climbing.

La taille du voisinage V dépend du nombre de case en conflit N et aussi de leur position. Dans le pire des cas, aucune case en conflit n'est située sur un bord. On peut donc toutes les déplacer à l'intérieur de la grille et générer toutes leurs rotations

$$|V_{max}| = N * (T-1)^2 * 4 = \mathbf{O(N * T^2)} \tag{3.1}$$

3.2 Autres caractéristiques et fonctions

Pour ce qui est de l'algorithme de recherche locale, nous avons implémenté un **Hill Climbing**.

a Fonction de sélection

La fonction de sélection parcourt le voisinage et retourne le premier voisin améliorant. Si aucun voisin n'améliore, alors on arrête la recherche. Dans le pire des cas, on parcourt tout le voisinage. La complexité de cette fonction pour un voisinage V est donc :

$$\mathbf{O}(|\mathbf{V}|) \tag{3.2}$$

b Fonction d'évaluation

La fonction d'évaluation compte le nombre de conflits dans la solution. Pour chaque case, on vérifie le bord SUD et EST. La complexité est donc :

$$O(T \times T \times 2) = \mathbf{O}(\mathbf{T^2}) \tag{3.3}$$

c Autres

Nous avons aussi ajouté un mécanisme de restart aléatoire. La solution initiale est générée avec le solveur heuristique de la partie 2. Le but est d'augmenter la diversification en optimisant des solutions très différentes, nous utilisons le solveur heuristique pour que la qualité des solutions initiale ne soit pas trop mauvaise.

L'algorithme du solveur *Recherchelocale* est disponible dans la partie 7.1. Les résultats obtenus avec ce solveur sont disponibles dans la partie 6.2. On observe une amélioration sur toutes les instances, à part la A où on avait déjà une solution valide avec le solveur heuristique.

4 Solveur métaheuristique

Pour cette partie du projet, nous avons voulu expérimenter avec les métaheuristiques de Large Neighboorhood Search (LNS) et Adaptive Large Neighboorhood Search (ALNS). L'idée derrière ces 2 métaheuristiques est de détruire une partie de la solution et de la reconstruire ensuite. Le choix des fonctions de destruction et de réparation est donc primordial.

Nous avons donc implémenté différentes fonctions de destruction et de réparation que nous décrivons dans les parties suivantes. Nous avons aussi choisi d'implémenter plusieurs fonctions d'acceptation afin de pouvoir tester différentes combinaisons.

4.1 Fonctions de destruction

Le rôle de la fonction de destruction est de dégrader la solution initiale en supprimant certaines valeurs. Nous fournissons à chacune de ces fonctions un paramètre *prctDestruct* contrôlant le pourcentage de la solution à détruire et elles retournent la solution dégradée, la liste des pièces supprimées et leurs positions.

a Destruction aléatoire

Nous supprimons aléatoirement un certain nombre de cases.

b Destruction avec une probabilité basée sur le nombre de conflits

Nous sélectionnons un sous-ensemble de pièces avec poids p_i qui varie selon le nombre de conflits de la pièce i et un paramètre additionnel $P_{conflit}$ qui donne plus ou moins d'importance aux conflits :

$$p_i = 1 + \texttt{nbConflit}(i) * P_{conflit} \tag{4.1}$$

Ces poids sont ensuite convertis en probabilité et on utilise une sélection roulette pour choisir les pièces à supprimer. L'idée est de supprimer en priorité des pièces en conflit tout en permettant aussi la suppression de quelques pièces aléatoires pour davantage faire varier la solution.

c Destruction des conflits seulement

On va sélectionner un sous-ensemble aléatoire des pièces en conflit et les supprimer. S'il y a moins de pièce en conflit que de pièce à supprimer, on va donc supprimer moins de pièce que demandé par *prctDestruct*. L'idée derrière cette fonction est de supprimer seulement des pièces en conflit afin de tenter de supprimer des conflits.

d Destruction de tous les conflits

Destruction de toutes les pièces en conflit sans prendre en compte *prctDestruct*. La logique est la même que pour la fonction précédente sauf que cette fois, on veut toujours supprimer toutes les pièces en conflit.
En pratique, pour des instances larges, le nombre de pièces en conflit est trop important en début de recherche et l'on détruit une trop grosse part de la solution pour la reconstruire de manière efficace.

4.2 Fonctions de réparation

Le rôle de cette fonction est de réparer la solution dégradée produite par la fonction de destruction afin d'obtenir une solution complète. Nous avons eu beaucoup de difficultés à trouver plusieurs façons efficaces de reconstruire une solution. Nous ne voulions pas utiliser une approche exacte, car cela nous aurait contraints à détruire trop peu de pièces. Nous voulions une reconstruction très rapide afin de pouvoir faire beaucoup d'itération et de redémarrage, c'est pour ça qu'une reconstruction heuristique nous semble être la meilleure solution.

a Réparation aléatoire

On replace les pièces supprimées dans un emplacement libre aléatoire avec une rotation aléatoire.

b Réparation heuristique - Emplacement first

On utilise l'heuristique finale du solveur heuristique (voir 2) pour replacer les pièces supprimées. Pour chaque emplacement, parcours toutes les pièces restantes et on choisit celle qui ajoute le moins de conflit. Pour ajouter de la diversité, la liste des emplacements et des pièces sont mélangés.

La complexité de cette fonction dépend du nombre d'emplacements libre n (égale au nombre de pièces à placer) :

$$\begin{aligned} \text{On reconnait une suite arithmétique : } & O(4 \times (n + (n-1) + (n-2)...1) \\ & = O(4 \times \frac{n \times (n+1)}{2}) \\ & = \mathbf{O(n^2)} \end{aligned} \tag{4.2}$$

c Réparation heuristique - Piece first

Même logique que la fonction précédente, sauf que l'on se place du point de vue de la pièce pour lui choisir le meilleur emplacement. La complexité est donc la même que précédemment.

4.3 Fonctions d'acceptation

Le rôle de la fonction d'acceptation est, comme son nom l'indique, d'accepter ou non la solution obtenue à l'issue de la phase de destruction/réparation. Ce choix peut se baser sur différents critères, la plupart du temps, on se base sur la différence de score entre la nouvelle solution et la solution actuelle.

a Tout accepter

$$\texttt{acceptAll()} \begin{cases} True & \text{toujours} \\ False & \text{jamais} \end{cases} \tag{4.3}$$

b Accepter les améliorations strictes

$$\texttt{acceptOnlyBetter}(oldCost, newCost) \begin{cases} True & \text{si } newCost < oldCost \\ False & \text{sinon} \end{cases} \tag{4.4}$$

c Accepter les solutions équivalentes et meilleures

$$\texttt{acceptSameOrBetter}(oldCost, newCost) \begin{cases} True & \text{si } newCost <= oldCost \\ False & \text{sinon} \end{cases} \tag{4.5}$$

d Accepter les dégradations avec un certain pourcentage

On accepte la solution améliorante avec une marge calculée d'après un pourcentage du score référence.

$$\texttt{acceptPrctWorst}(oldCost, newCost, prct) \begin{cases} True & \text{si } newCost < oldCost + prct * oldCost \\ False & \text{sinon} \end{cases} \tag{4.6}$$

4.4 Large Neighboorhood Search (LNS)

Nous avons implémenté un LNS afin de pouvoir évaluer la qualité de chacune de nos fonctions de réparation, destruction et acceptation. Cette fonction nous sert aussi à essayer différentes combinaisons de fonctions afin de trouver lesquelles fonctionnent le mieux ensemble.

Nous avons ajouté un mécanisme de restart variable, permettant de redémarrer soit d'une solution aléatoire, soit de la meilleure solution trouvée. On peut choisir des fonctions de destruction/réparation/acceptation différente suivant le type de restart. Le but est de trouver le bon équilibre entre diversification et intensification.

L'algorithme de notre Solveur LNS est disponible dans la partie 7.1.

a Meilleure configuration

Cet algorithme nous a donnée de très bons résultats avec cette combinaison de fonctions :

- **Destruction des conflits seulement**
- **Réparation heuristique - Emplacement first**
- **Tout accepter**

Cette configuration allie une bonne **diversification** avec notre fonction d'acceptation, tout en **intensifiant** la recherche sur les pièces problématiques avec notre fonction de destruction. La fonction de réparation considérant l'emplacement en premier a montré des résultats légèrement meilleurs, mais tout de même assez similaires à celle considérant les pièces en premier.

Pour le ratio entre les démarrages depuis la meilleure solution et les redémarrages aléatoires, nous sommes parvenus à la conclusion que 2 "aléatoire" pour un "meilleur" est surement le meilleur compromis.

Pour calibrer nos hyperparamètres, nous avons utilisé un système de log et lancé beaucoup de recherche en parallèle afin de récolter des données, que nous avons ensuite analysées pour déterminer les meilleures configurations. Cette méthode nous a donnée des idées pour de nouvelles fonctions et finalement nous a permis de sélectionner la meilleure configuration.

Les résultats obtenus avec cet algorithme sont disponibles dans la partie 6.3.

4.5 Adaptive Large Neighboorhood Search (ALNS)

L'Adaptive Large Neighboorhood Search est une évolution du LNS qui permet l'utilisation de plusieurs fonctions de destruction/réparation/acceptation au lieu d'une seule. Un mécanisme de sélection roulette et de mise à jour des poids assignés aux différentes fonctions, basé sur les résultats de chaque itération, permet de faire varier la configuration au fur et à mesure de la recherche. Le but est de pouvoir mieux s'adapter aux différences entre les instances et d'exploiter au maximum les différentes fonctions dans les contextes qui leur sont le plus avantageux.

Pour nous, cet algorithme a été un échec partiel. Nous avons réussi à l'implémenter et nous avons même ajouté un mécanisme de choix sur la fonction d'acceptation qui n'était pas présent dans l'algorithme du cours.

Malgré de nombreux essais et analyses, nous n'avons jamais réussi à surpasser les résultats obtenus avec le LNS "simple". Sur certaines instances, nous arrivons à des résultats similaires, mais souvent en plus de temps. Nous pensons que ces résultats décevants peuvent être dus à plusieurs facteurs :

1. **Le nombre trop important d'hyperparamètres** : l'ALNS demande de choisir des listes de fonctions, un facteur λ et des poids de mise à jour en plus de tous les autres hyperparamètres déjà présents. Le nombre de possibilités explose et la modification d'un seul hyperparamètres a une influence sur le choix de tous les autres. **Nous n'avons surement pas réussi à trouver une configuration suffisamment performante.**

2. **Le manque de diversité dans la reconstruction** : Nous n'avons pas réussi à trouver suffisamment d'idée de reconstruction différente pour que l'ALNS performe à son plus haut potentiel. **Nos 2 fonctions de reconstructions sont trop similaires** et la fonction random n'apporte pas grand-chose.

Nous restons persuadés que le choix d'un ALNS est particulièrement pertinent dans le contexte de ce projet. Mais comme les résultats ne sont pas au rendez-vous et pour ne pas surcharger davantage le rapport, nous n'approfondirons pas plus ce sujet. Le code source est disponible, mais nous ne fournissons pas l'algorithme dans ce rapport.

5 Analyse des performances

Commençons l'analyse des performances des différents solveurs par une comparaison des meilleurs résultats obtenus sur chaque instance :

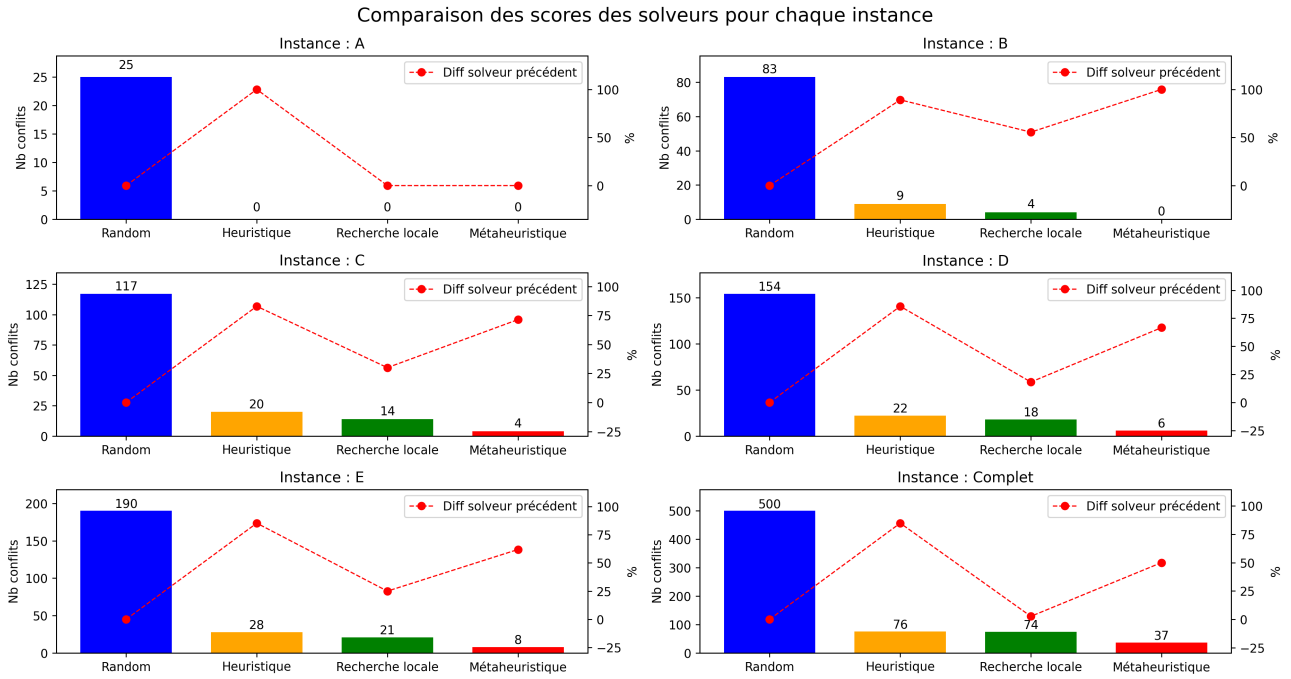

FIGURE 5.1 – Comparaison des meilleurs scores obtenus par les différents solveurs.

La figure 5.1 permet de visualiser facilement l'évolution des performances des différents solveurs. On remarque une très forte amélioration entre le solveur *Random* et le solveur *Heuristique* ce qui est attendu. Mais surtout, on peut constater que le solveur *Métaheuristique* est nettement plus performant que tous les autres. En moyenne, il améliore les solutions de plus de 50%. Alors qu'entre le solveur *Heuristique* et le solveur *Recherchelocale*, l'amélioration moyenne est autour des 20%.

Une comparaison des temps d'exécution des solveurs ne serait pas très pertinente, ils sont trop différents. Par exemple, pour l'instance complète, le solveur *Recherchelocale* fait 45 restarts et 474 itérations en 1 heure, alors que le solveur *Métaheuristique* fait 73 restarts et plus de 1 500 000 itérations pour le même temps d'exécutions. C'est normal, car les deux algorithmes n'utilisent pas les mêmes fonctions ou logiques de recherche et sont donc difficilement comparables sur ce point.
Sinon, une comparaison des temps d'exécution des fonctions de voisinages de la recherche locale est disponible ici.

Pour conclure ce rapport, nous avons trouvé pertinent d'analyser la complexité temporelle de notre algorithme LNS. Nous avons évalué sur plusieurs runs le temps d'exécution moyen d'une itération (destruction + reconstruction + acceptation) sur chaque instance. Le but est d'estimer grâce à une régression polynomiale l'évolution de la complexité temporelle en fonction de la taille de l'instance.

Sur la figure 5.2 on constate que c'est un polynôme de degrés 4 qui a le meilleur score R^2, la complexité serait donc $\mathbf{O(T^4)}$ où T est la taille de l'instance. Il faut toutefois relativiser ces résultats. En effet, nous avons ajusté le polynôme avec seulement 5 points, ce qui est très peu et on peut clairement remarque une sorte d'overfitting sur

la courbe du polynôme de degré 4. Il nous faudrait donc beaucoup plus d'instances de tailles différentes pour mesurer de manière fiable la complexité temporelle de cette manière.

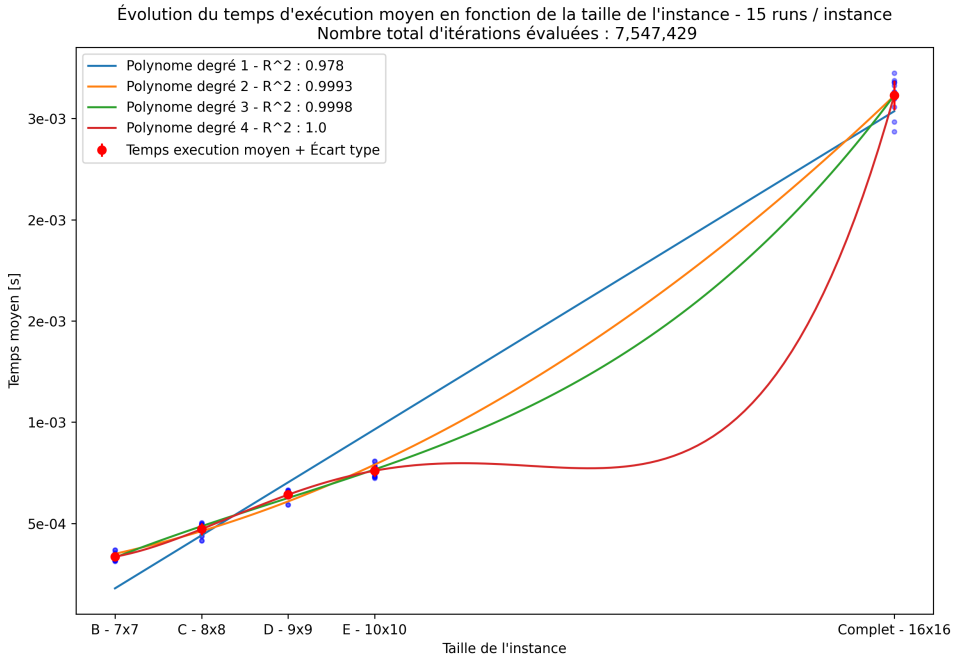

FIGURE 5.2 – Analyse de la complexité temporelle de l'algorithme LNS

Pour comparer, on peut rapidement faire une estimation de la complexité des différentes fonctions utilisée dans une itération :

- La **fonction de destruction** parcourt chaque pièce du plateau pour compter son nombre de conflits, on a donc une complexité $\mathbf{O(T^2)}$
- La **fonction de réparation** dépend du nombre n de pièces détruites, on a donc $n = 10\% \times T^2$ et donc $O((\frac{T^2}{10})^2) = \mathbf{O(T^4)}$ (voir b).
- La **fonction d'acceptation** nécessite de compter le nombre de conflits donc $\mathbf{O(T^2)}$ (voir b).

Finalement, on a une complexité temporelle $O(T^2 + T^4 + T^2) = \mathbf{O(T^4)}$ ce qui correspond à l'estimation précédente.

6 Résultats

Les résultats obtenus avec les différents solveurs du projet sont regroupés dans les tableaux suivants :

6.1 Solveur Heuristique

Instance	Meilleur score	Score moyen	Temps d'exec moyen [s]	Nb essai
A	**0**	3.155	3.0E-4	200
B	**9**	15.645	2.2E-3	200
C	**20**	27.695	4.1E-3	200
D	**22**	32.63	6.3E-3	200
E	**28**	38.16	9.3E-3	200
Complet	**76**	99.742	8.0E2	200

Tableau 6.1 – Résultats solveur heuristique

6.2 Solveur Recherche locale

Instance	Meilleur score (diff Heuristique)	Nb restart	μ / σ	Nb total iter	Temps alloué
A	**0** (-0)	0	0 / 0	0	15 min
B	**4** (-5)	5240	11.57 / 2.04	23750	15 min
C	**14** (-6)	1605	22.95 / 2.71	8757	15 min
D	**18** (-4)	657	27.11 / 2.96	3948	15 min
E	**22** (-6)	360	32.21 / 3.22	2292	15 min
Complet	**74** (-2)	45	87.2 / 5.04	474	1 h

Tableau 6.2 – Résultats solveur Recherche Locale. μ/σ : moyenne et écart type des scores à la fin des restart

6.3 Solveur Métaheuristique

Instance	Meilleur score (diff Recherche locale)	Nb restarts Best / Random	% destruction	Temps recherche
A	**0** (-0)	0 / 0	-	**-** / 1 h
B	**0** (-4)	255 / 511	10%	**46 min** / 1 h
C	**4** (-10)	194 / 389	10%	**1 h**
D	**6** (-12)	105 / 211	10%	**1 h**
E	**8** (-14)	86 / 172	10%	**1 h**
Complet	**37** (-37)	24 / 49	10%	**1 h**

Tableau 6.3 – Résultats solveur Métaheuristique

6.4 Points obtenus

Instance	Upper bound	Meilleur score	Score autograder
A	0	**0**	**1** / 1
B	0	**0**	**1** / 1
C	**0**	4	**0** / 1
D	**180**	6	**1.93** / 2
E	**220**	8	**1.93** / 2
Total	-	-	**5.86** / 7

Tableau 6.4 – Points obtenus

7 Annexe & Bibliographie

7.1 Algorithmes

Algorithm 1 Heuristique : nombre de conflits apportés par une pièce

Input : Instance du puzzle, Solution incomplète, Nouvelle pièce p
Output : Score S

$S = 0$
Pour chaque bord b de p :
  Si b est au bord de la grille **ET** b n'est **pas** gris :
    $S \leftarrow S + 10$
  Sinon Si b n'est **pas** au bord de la grille **ET** b est gris :
    $S \leftarrow S + 10$
  Sinon Si b voit une pièce adjacente **ET** b n'est **pas** de la même couleur que cette pièce :
    $S \leftarrow S + 1$
return S

Algorithm 2 Solveur heuristique

Input : Instance du problème, Fonction heuristique H
Output : Solution complète S

$S \leftarrow$ Solution vide
$P \leftarrow$ Liste des pièces ▷ Liste contenant les pièces à placer
Mélange de P
$i \leftarrow 0$ ▷ Index de la pièce a placer
while $len(P) > 0$ **do**
  $coord \leftarrow$ `orderPlacement`(i)
  $allPossibilities \leftarrow$ `getAllPlacementPossibilities`(p)
  $h^* \leftarrow 0$
  $p^* \leftarrow Null$
  for chaque $piece$ de $allPossibilities$ **do**
    $h \leftarrow H(coord, piece)$
    if $h < h^*$ **then**
      $h^* \leftarrow h$
      $p^* \leftarrow piece$
      if $h^* = 0$ **then**
        Sortie de la boucle **for**
      end if
    end if
  end for
  Ajout de p^* à S
end while
return S

Algorithm 3 Solveur recherche locale

Input : Instance du problème I, Fonction de voisinage N, Fonction de sélection S, Fonction de coût f, $maxTime$

Output : Solution optimisée

$s^* \leftarrow$ `solverHeuristique(`I`)`

while ($currentTime < maxTime$) **do** ▷ Boucle restart
  $s \leftarrow$ `solverHeuristique(`I`)` ▷ Génération nouvelle solution de départ
  $s^*_{current} \leftarrow s$

  while ($currentTime < remainingTime$) **do** ▷ Recherche locale
    $V \leftarrow N(s)$
    $s \leftarrow S(V)$
    if s est vide **then**
      Sortie de la boucle **while** de recherche locale
    end if

    if $f(s) < f(s^*_{current})$ **then**
      $s^*_{current} \leftarrow s$
      if $f(s^*_{current}) = 0$ **then** ▷ Si on a une solution optimale
        Sortie de la boucle **while** de recherche locale
      end if
    end if
  end while

  if $f(s^*_{current}) < f(s^*)$ **then** ▷ Mise à jour meilleure solution
    $s^* \leftarrow s^*_{current}$
    if $f(s^*) = 0$ **then** ▷ Si on a une solution optimale
      Sortie de la boucle **while** de recherche locale
    end if
  end if
end while

return s^*

Algorithm 4 Solveur LNS

Input : Instance du problème I, Fonction de destruction random d_r et best d_b, Fonction de réparation random r_r et best r_b, Fonction d'acceptation random a_r et best a_b, Fonction de coût f, Pourcentage de destruction $prctDestruct$, Ratio de restart avec la meilleure solution $ratioBest$, Temps maximum $maxTime$

Output : Solution optimisée

$s^* \leftarrow$ `solverHeuristique(`I`)`

while ($currentTime < maxTime$) **do** ▷ Boucle restart

  $restartBest \leftarrow$ Choix du restart en fonction de $ratioBest$

  $d, r, a, s \leftarrow \begin{cases} d_r, r_r, a_r, s^* & \text{si } restartBest = True \\ d_b, r_b, a_b, \texttt{solverHeuristique(}I\texttt{)} & \text{sinon} \end{cases}$ ▷ Initialisation du restart

  $s^*_{current} \leftarrow s$

  while ($currentTime < remainingTime$) **do** ▷ LNS

    $s' \leftarrow r(d(s))$ ▷ Destruction + Réparation

    if $f(s') = 0$ **then** ▷ Si on a une solution optimale

      $s^*_{current} \leftarrow s'$

      Sortie de la boucle **while** du LNS

    end if

    if $a(s')$ **then** ▷ Acceptation de la solution reconstruite

      $s \leftarrow s'$

    else

      Passage à la prochaine itération de la boucle **while** du LNS

    end if

    if $f(s) < f(s^*_{current})$ **then**

      $s^*_{current} \leftarrow s$

      if $f(s^*_{current}) = 0$ **then** ▷ Si on a une solution optimale

        Sortie de la boucle **while** de recherche locale

      end if

    end if

    if On dépasse le nombre maximum d'itération sans amélioration **then**

      Sortie de la boucle **while** de recherche locale

    end if

  end while

  if $f(s^*_{current}) \leq f(s^*)$ **then** ▷ On accepte les solutions équivalente pour faire varier la solution

    $s^* \leftarrow s^*_{current}$

    if $f(s^*) = 0$ **then** ▷ Si on a une solution optimale

      Sortie de la boucle **while** de recherche locale

    end if

  end if

end while

return s^*

7.2 Bibliographie

Références

[1] *Eternity II puzzle - Wikipedia*, https://en.wikipedia.org/wiki/Eternity_II_puzzle, (Accessed on 04/14/2024).