INF6102 – Métaheuristiques appliquées au génie informatique

Projet - Eternity II

Guillaume Thibault
g.thibault@polymtl.ca

Guillaume Blanché
guillaume.blanche@polymtl.ca

23 avril 2023

1 Eternity II

Eternity II est un puzzle publié en 2007 afin de succéder à Eternity I, un ancien problème de satisfaction combinatoire qui avait été résolu par la communauté. Ce problème a été rendu célèbre par sa difficulté de résolution et son enjeu : une somme de 2 millions de dollars avait été promise à toute personne trouvant une solution au jeu mais à la clôture de la compétition en 2010, aucune solution valide n'avait été trouvée. La tâche a accomplir est de placer chacune des 256 pièces dans une grille de taille 16x16 de façon à faire correspondre les couleurs de chacun des 4 côtés des pièces.

Nous pouvons dès lors donner une approximation de la taille de l'espace de recherche en fonction de la taille du puzzle. Pour un puzzle de taille n, en comptant toutes les permutations entre les n^2 pièces et les 4 orientations possibles par pièce, on obtient au total : $(n^2)! \cdot 4^{n^2}$ solutions possibles. Pour le puzzle complet de taille 16×16, on a alors environ 10^{154} solutions possibles, ce qui exclue dès maintenant l'utilisation de toute approche de recherche complète pour résoudre le problème.

Par ailleurs, le problème est connu pour être NP-complet, ce qui signifie qu'il n'existe pas d'algorithme efficace connu pour trouver une solution optimale, et le problème est susceptible de devenir de plus en plus difficile à mesure que la taille de la grille et le nombre de pièces de puzzle augmentent. Par conséquent, des méthodes heuristiques et métaheuristiques sont souvent utilisées pour rechercher de bonnes solutions en un temps raisonnable.

2 Méthodes de résolution

2.1 Phase 1 : Solveur heuristique

Construction de la solution Notre solveur heuristique se base sur une construction progressive minimisant à chaque pièce ajoutée le nombre de conflits. Notre création d'une solution heuristique se déroule en 3 étapes principales : le placement des coins, puis le placement des côtés, et enfin le placement des pièces intérieures.

Une première difficulté à résoudre consiste à choisir comment seront placés les coins (leur orientation est naturellement donnée par les deux côtés gris des pièces qui sont dirigés vers l'extérieur du puzzle). Il existe 4 configurations de géométries différentes, qui ne peuvent pas être obtenues à partir de la simple rotation du puzzle. Notre première tâche consiste alors à déterminer laquelle de ces 4 configurations est la meilleure, ce que nous réalisons de façon exhaustive. Ensuite, à partir d'une géométrie des coins donnés, les côtés du puzzle sont placés de façon à minimiser le nombre de conflits sur le bord du puzzle. Nous générons les côtés avec une approche gloutonne : on procède pièce par pièce en déterminant la pièce qui ajoute le moins de conflits à la construction en cours. Nous pouvons noter que là aussi, l'orientation des côtés est naturellement donnée par leur position sur le puzzle : la couleur grise est toujours orientée vers l'extérieur.

Enfin, une fois le bord du puzzle complètement construit, et toujours pour une géométrie de coins de départ donnée, on place les pièces centrales de façon gloutonne selon le même processus que pour le placement des côtés. On construit donc le puzzle pièce par pièce en prenant là aussi celle qui ajoute le moins de conflits à la construction. Contrairement aux étapes précédentes, nous devons ici générer toutes les rotations possibles de chaque pièce pour en trouver la meilleure orientation. Cependant,

nous ajoutons ici un autre mécanisme en cas d'égalité entre plusieurs possibilités de pièce pour une position. Afin de réunir les pièces contenant une même couleur sur une région limitée du puzzle, nous privilégions les pièces qui n'utilisent pas de nouvelles couleurs dans la construction en cours en cas d'égalité. Cette technique s'est avérée efficace puisque ce mécanisme a fait gagné 7 conflits dans la construction heuristique du puzzle complet, passant d'un coût de 92 à 85.

Maintenant que la construction a été réalisée pour une géométrie des coins donnée, nous pouvons réitérer pour chacune des autres géométries possibles afin de trouver celle qui réduit le coût global de la construction.

Discussion des choix de constructions Un choix de conception majeur dans notre solveur heuristique qui n'a pas encore été discuté est l'ordre des positions auxquelles sont attribuées les pièces. Nous avons implémenté deux possibilités pour cela : d'une part une construction ligne par ligne, et d'autre part une construction couche par couche. Nous verrons qu'en pratique, si les deux approches tendent à obtenir des scores relativement proches, les structures des solutions obtenues par chacune de ces deux approches sont très différentes.

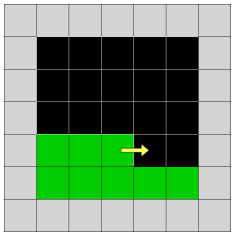


FIGURE 1 – Construction en lignes

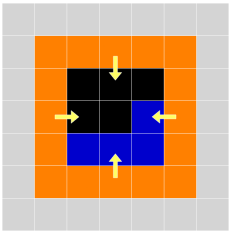


FIGURE 2 – Construction en couches

Résultats À partir des solutions illustrées dans les figures 7 et 8, nous pouvons faire certaines remarques. Les deux méthodes obtiennent tout d'abord des scores assez similaires, il n'est donc pas possible de dire à priori si l'une des deux approches est meilleure. Pour les instance 7×7 et 16×16, la construction en lignes obtient un meilleur score, alors que la construction par couches est la meilleure pour les instances de taille intermédiaire : 8×8, 9 *times*9 et 10×10. Nous pouvons ensuite discuter quelle approche sera la plus intéressante pour réaliser une recherche locale à partir d'une solution heuristique. L'objectif sera alors de déconstruire la construction pour s'écarter du minima local et ainsi réduire le nombre de conflits.

Dans la construction en ligne, les conflits sont davantage concentrés sur le bord supérieur du puzzle puisqu'à cette étape il ne reste que peu de choix de pièces possibles pour achever la construction. Nous pouvons de plus observer que la majorité des conflits générés par cette approche sont du type NORD-SUD. Enfin, les conflits sont répartis sur un nombre relativement important de pièces : la plupart des pièces en conflit dans cette approches ont au maximum deux conflits.

Pour la construction par couche, les conflits sont concentrés au centre du puzzle, c'est-à-dire là où les dernières pièces ont été placées. Cependant, ici il n'est pas possible d'identifier clairement une orientation privilégiée pour les conflits puisque les conflits sont créés à partir de chacune des directions menant vers le centre du puzzle. De plus, les conflits ont tendance à être davantage concentrés sur un nombre réduit de pièces, et il est ici plus fréquent de voir des pièces avec 3 conflits dans cette approche.

Nous avons fait le choix d'utiliser la construction par couches dans les algorithmes de recherche locale décrits dans les sections suivantes car la propriété de concentration des conflits sur certaines pièces du puzzle est intéressante. Elle permet de réduire le nombre de mouvements locaux nécessaires pour abaisser le coup de la solution puisqu'ici, les échanges entre pièces en conflits auront une plus grande chance de succès à cause de la très mauvaise qualité du centre. Cependant, pour l'instance complète, la version en lignes a fourni de meilleurs résultats.

Algorithm 1 Solveur heuristique

```
for corner_geometry in corner_geometries do
    solution ← corner_geometry
    for edge_position in edge do
        solution[edge_position] = best_edge
    end for
    for inner_position in inner do
        solution[inner_position] = best_inner
    end for
    geometry_cost ← solution_cost
    if geometry_cost < best_geometry_cost then
        best_geometry_cost ← geometry_cost
    end if
end for
return solution_best_geometry
```

2.2 Phase 2 : Recherche locale par Simulated Annealing

Notre solveur de recherche locale implémente l'algorithme de simulated annealing. Cet algorithme inspiré de la métallurgie permet d'explorer un large espace de recherche, en se concentrant au fur et à mesure sur un espace de bonne solutions. Au début de la recherche, on s'autorise à explorer des voisinages de mauvaise qualité pour assurer une grande exploration, et en fin de recherche, l'algorithme s'intensifie en ne considérant que les solutions améliorantes.

On démarre notre recherche locale avec une solution initiale générée aléatoirement et une température élevée. Nous ajoutons seulement une contrainte pour générer la solution aléatoire : les côtés gris des pièces appartenant aux bords doivent être orientés correctement. De cette manière, l'algorithme converge plus rapidement vers les minima locaux, sans risquer de rater l'optimum. A chaque itération, une solution candidate est générée à partir d'une fonction de voisinage. Si la solution candidate est meilleure que la solution courante, elle est directement acceptée, et sinon elle est acceptée avec une probabilité qui dépend de la température courante et du coût de la solution. Pour diminuer la température au cours de la recherche, l'algorithme simulated annealing introduit de plus un paramètre α selon la loi $T_{new} = \alpha \cdot T_{old}$.

2.2.1 Fonction de voisinage : rotation et 2-swap

La fonction de voisinage de notre solveur de recherche locale commence par identifier toutes les pièces comportant au moins un conflit. Pour chacune de ces pièces, nous ajoutons chacune des 3 autres orientations possible dans le voisinage. Ensuite, on ajoute également au voisinage tous les 2-swap possibles entre des pièces en conflit, en conservant ici les orientations. Enfin, dans l'objectif de calculer plus rapidement la fonction de voisinage, nous fixons une taille maximale pour notre voisinage. Plus précisément, nous limitons le nombre de pièces en conflits considérées lors d'une itération du calcul du voisinage. Pour éviter de retomber dans les mêmes minima locaux, l'ordre de sélection des pièces en confits est rendu aléatoire. Ce voisinage est clairement connecté car si nous connaissions la solution optimale et en partant d'une solution quelconque, il suffirait de réaliser les 2-swap entre les positions des pièces sur la solution considérée avec les positions occupées dans la solution optimale. Les rotations permettent ensuite d'orienter correctement chacune des pièces pour finalement reconstruire complètement la solution optimale.

2.2.2 Fonction de validation : solution améliorante

Pour sélectionner le nouveau candidat dans le voisinage obtenu, nous commençons par chercher les voisins améliorant la solution courante. Si de telles solutions existent, nous choisissons aléatoirement notre candidat parmi celles-ci. Sinon, nous choisissons aléatoirement le candidat parmi le voisinage complet.

2.2.3 Stratégie de redémarrage

Nous introduisons enfin un redémarrage sur une solution aléatoire lorsque la température obtient des valeurs très faibles et que la recherche, bien que très intensifiée, ne parvient plus à trouver de meilleures solutions. En pratique, nous faisons redémarrer la recherche lorsque la probabilité de sélection est restée inférieure à un seuil donné depuis un nombre fixé d'itérations.

Algorithm 2 Recherche locale : Simulated annealing

```
solution ← random_solution
best_solution ← solution
T ← T_0
while temps non épuisé do
    N ← neighborhood(solution)
    candidate ← select_candidate(N)
    Δ ← cost(solution) − cost(candidate)
    if Δ < 0 then
        solution ← candidate
    else
        solution ← candidate avec la probabilité e^{−Δ/T}
    end if
    if cost(solution) < cost(best_solution) then
        best_solution ← solution
    end if
    T ← T · α
    if P = P_min then
        no_progress_counter = no_progress_counter + 1
        if no_progress_counter ≥ max_no_progress then
            solution ← random_solution          ▷ Redémarrage lorsque la probabilité a convergé
        end if
    end if
end while
return best_solution
```

2.3 Solveur avancé : Algorithme génétique

2.3.1 Motivation

Le choix de l'algorithme génétique a été motivé par plusieurs raisons. Tout d'abord, les algorithmes génétiques, et de manière générale les approches évolutionnaires ont largement été utilisées dans la littérature traitant du problème Eternity II, ce qui nous a servi de point de départ pour développer notre solveur. Les algorithmes génétiques sont ensuite généralement très performants pour résoudre des problèmes de grande taille, là où explorer des voisinages entiers est très coûteux. Enfin, l'algorithme génétique a la principale caractéristique de pouvoir combiner des solutions de bonne qualité pour en générer de nouvelles potentiellement meilleures, ce qui peut devenir particulièrement intéressant si l'on a déjà des bonnes solutions. Notre résolution du problème décompose la résolution du bord du puzzle et la résolution de l'intérieur en deux recherches locales séparées. Une fois le bord construit, la seconde recherche ne modifiera pas la construction du bord.

L'intérêt de cette séparation est que l'on peut facilement réduire la taille de l'espace de recherche pour chacune de ces deux étapes : au départ en se concentrant sur les pièces possédant la couleur grise puis ensuite sur les autres pièces. Cependant, la bordure générée peut aussi poser de nouvelles difficultés : si la construction de la bordure est mauvaise, on risque d'ajouter des contraintes impossible à satisfaire pleinement pour les pièces intérieures qui seront ajoutées ensuite.

2.3.2 Résolution de la bordure

Génération de la population Nous générons la population avec des solutions aléatoires selon la même procédure que dans l'algorithme de Simulated Annealing décrit précédemment : les orientations

des bords sont prédéterminées par leur position (par exemple si on souhaite placer une pièce avec un côté gris sur le bord du bas, le côté gris sera toujours orienté vers le bas). Nous introduisons également des solutions élites à partir de notre solveur heuristique. Si les solutions élites assurent très rapidement un bon score de la solution, le principal risque est de restreindre la recherche local autour du minima local trouvé dans le solveur heuristique et donc de manquer de diversification.

Crossover La fonction de crossover utilisée pour le solveur du bord du puzzle consiste à sélectionner certaines pièces et à réaliser un 2-swap sur ces deux pièces. On échange par ailleurs les orientations de ces deux pièces lors du 2-swap. De cette façon, le côté gris de chacune des pièces reste orienté vers le bord du puzzle. La sélection des pièces à échanger est réalisée de façon similaire à la sélection qui était faite dans la fonction de voisinage du simulated annealing précédemment décrite : on sélectionne d'abord chacune des pièces en conflits et on fait des 2-swap entre ces positions et d'autres positions du bord choisies aléatoires, et d'autre part on ajoute des 2-swap entre des pièces choisies aléatoirement afin d'introduire davantage de diversification à la recherche. Cette technique permet de trouver un bon compromis entre intensification et diversification de l'exploration de l'espace de recherche, en s'intéressant à la fois aux pièces ayant des conflits tout en explorant des voisinages diversifiés.

Notre algorithme génétique de résolution de bord n'utilise pas de fonction de mutation car la seule perturbation que nous souhaitons considérer ici est l'échange des positions du bord qui est déjà introduit par la fonction de crossover.

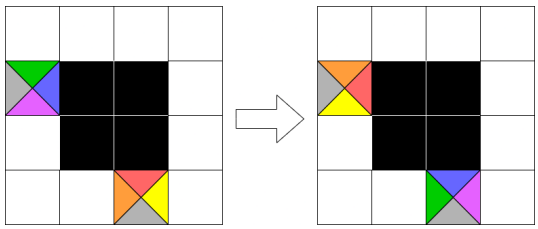


FIGURE 3 – Fonction de crossover appliquée sur le bord

Sélection La sélection des solutions de la population à conserver pour la génération suivante consiste à réaliser un tournoi. Formellement, à partir d'une population de taille initiale P_{init}, on considère un tournoi de taille $N_{tournoi}$ et un nombre de solutions acceptées N_{accept}. On sélectionne itérativement les individus de la population choisissant N_{accept} individus parmi les $N_{tournoi}$ formant le tournoi, jusqu'à obtenir la taille de population souhaitée.

2.3.3 Solveur pour l'intérieur du puzzle

Une fois que le temps dédié à la construction du bord est écoulé, la construction du centre peut débuter. La plupart des mécanismes utilisés ici sont relativement communs à ceux décrits lors de l'étape précédente, nous détaillerons donc simplement les différences notoires.

Génération de la population La population est ici aussi générée de façon aléatoire mais respecte la bordure qui a été générée auparavant. Les pièces qui ne contiennent pas la couleur grise sont donc assignées à des positions et des orientations aléatoires à l'intérieur de la construction.

Crossover La fonction de crossover utilisée dans le solveur de l'intérieur du puzzle est très similaire à celle décrite pour le bord. La principale différence introduite ici est l'utilisation de rotations aléatoires lors du 2-swap entre les pièces qui sont sélectionnées de la même façon qu'auparavant.

Pour choisir les pièces qui doivent être échangées, nous considérons d'une part, la totalité des positions impliquées dans un conflit, puis d'autre part un sous-ensemble des pièces intérieures choisies aléatoirement.

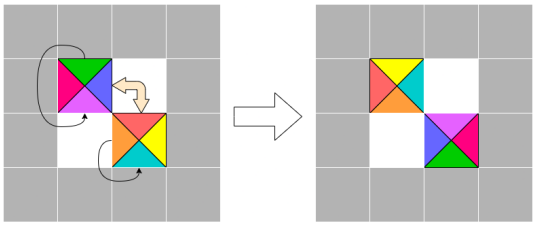


FIGURE 4 – Fonction de crossover appliquée dans la partie interne

Mutation Le solveur de l'intérieur du puzzle utilise un opérateur de mutation pour introduire davantage de diversification. Il s'agit de réaliser une rotation aléatoire sur une position de l'intérieur du puzzle sélectionnée aléatoirement avec une probabilité définie par un taux de mutation à fixer.

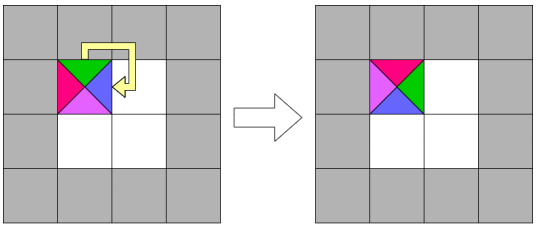


FIGURE 5 – Fonction de mutation

Recherche locale supplémentaire Il est nécessaire d'ajouter encore davantage de diversification à l'algorithme car, avec l'utilisation de solutions élites, le solveur peine à améliorer les solutions purement heuristique. Pour cela, nous réalisons une recherche locale à génération de l'algorithme génétique, sur base de la meilleure solution de la population courante. De cette façon, la recherche explore de nouveaux voisinages, à la manière de la Variable Neighborhood Descent. Nous avons utilisé ici la recherche locale du simulated annealing car elle permet justement d'apporter suffisamment de diversification en adaptant le choix de la température initiale, tout en garantissant de bonnes performances. De plus elle n'est pas très coûteuse en temps de calcul car elle n'explore pas la totalité du voisinage.

Hyperparamètres

Seed : 1234

Construction du bord :
- Temps de recherche : 60 secondes
- Taille de la population : 20
- Taille du tournoi : 10
- Nombre de solutions acceptées par tournoi : 5
- Nombre de générations maximal : 100
- Nombre de générations maximal sans amélioration : 10
- Nombre de solutions élites : 1

Construction de la partie interne :
- Temps de recherche : 19 minutes
- Taille de la population : 100
- Taille du tournoi : 100
- Nombre de solutions acceptées par tournoi : 30
- Taux de mutation : 0.01
- Durée de la recherche locale : 5 secondes
- Température initiale du simulated annealing : 10^{10}
- Nombre de générations maximal : 1000
- Nombre de générations maximal sans amélioration : 10
- Nombre de solutions élites : 1

Algorithm 3 Algorithme génétique

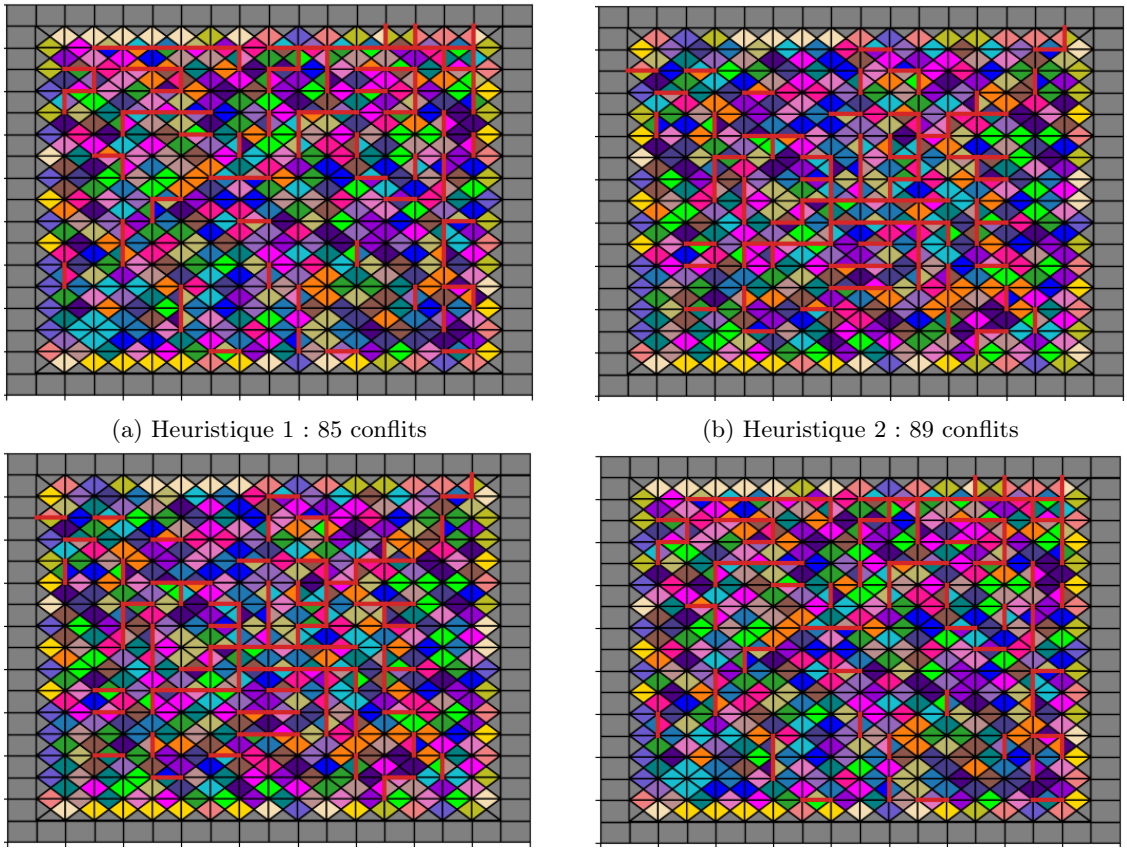
```
no_search_best_score = −∞
best_score = −∞
while temps non épuisé do
    population ← generate_population()
    generation = 0
    while generation ≤ num_generations do
        generation = generation + 1
        parents ← select_parents(population)
        elite ← select_elite(population)
        for parent ∈ parents do
            child ← crossover(parent)
            child ← mutation(child)
        end for
        population ← elite + tournament_selection(parents + children)
        best_gen_solution = select_best_individual(population)
        generation_score = score(best_gen_solution)
        if generation_score ≤ no_search_best_score then
            no_search_best_score = generation_score
            search_solution ← local_search(best_gen_solution)
            search_solution_score ← score(search_solution)
            population ← population + search_solution
            if search_solution_score < best_score then
                best_solution ← search_solution
                best_score ← search_solution_score
            end if
        else
            no_progress_counter = no_progress_counter + 1
        end if
        if no_progress_counter ≥ max_no_progress then        ▷ Redémarrage de la recherche
            break
        end if
    end while
end while
return best_solution
```

3 Résultats

3.1 Résultats sur les instances fournies

Nombre de conflits				
Instance	Heuristique1	Heuristique2	Simulated Annealing	GA
A	0	0	0	0
B	**14**	15	15	**14**
C	26	**20**	**20**	26
D	35	36	36	**34**
E	35	**33**	**33**	35
Complet	**85**	89	88	**85**

TABLE 1 – Scores obtenus sur les différentes instances sur les différents solveurs



(a) Heuristique 1 : 85 conflits

(b) Heuristique 2 : 89 conflits

(c) Simulated annealing : 88 conflits

(d) Algorithme génétique : 85 conflits

FIGURE 6 – Résultats visuels

4 Analyse

Après analyse des résultats, il est à noter que, parmi les quatre algorithmes développés ayant donné les meilleurs résultats, tous on résolus le problème A, mais aucun n'a pu résoudre les autres instances de manière optimale. Parmi les instances non résolues, deux des algorithmes ont produit des résultats identiques pour l'instance complète, deux autres ont donné les mêmes résultats pour l'instance E, et la même tendance a été observée pour les instances B et C. Pour l'instance D, seul l'algorithme génétique a permis d'obtenir le meilleur résultat.

Dans l'ensemble, ces résultats indiquent que l'algorithme génétique est le plus performant parmi ceux que nous avons développés. En effet, il a produit les meilleurs résultats (parmi ceux que nous avons observés avec nos implémentations) pour trois des instances, à savoir l'instance complète, l'instance D et l'instance B, en plus de permettre une construction optimale pour l'instance A, comme les autres algorithmes.

De plus, lors de l'observation des visualisations des solutions trouvées par nos quatre algorithmes, il est apparu que l'algorithme génétique présentait une tendance à générer la majorité de ses conflits avec la bordure en raison de l'utilisation de notre heurisitique en lignes. En revanche, les autres algorithmes ont présenté leurs conflits soit dans la partie centrale du casse-tête, soit dans la partie supérieure. En considérant ces résultats, nous sommes d'avis que l'approche utilisant l'algorithme génétique est la plus prometteuse pour une éventuelle continuation de la résolution de ce problème.

Un autre point important à souligner est que nos algorithmes ont été capables d'obtenir les résultats en seulement quelques itérations. En effet, nous avons alloué un temps de recherche de 20 minutes, mais les algorithmes ont rapidement convergé vers un minimum local, sans pouvoir en sortir par la suite. Bien que nous ayons tenté d'introduire des étapes supplémentaires à la recherche, telles que la recherche de voisinage variable et la recherche de voisinage large à partir des solutions trouvées par l'algorithme génétique, ces mécanismes n'ont pas été en mesure de sortir du minimum local.

Après une évaluation plus approfondie (voir annexe) de ces deux algorithmes, nous avons également constaté que les voisinages n'étaient pas suffisamment diversifiés. Par conséquent, nous considérons que l'amélioration de la méthode viendrait davantage d'une amélioration de la méthode de construction plutôt que de l'ajout d'une méthode de recherche locale.

5 Conclusion

Dans ce rapport, nous avons présenté quatre approches métaheuristiques différentes pour résoudre le puzzle Eternity II : tout d'abord un solveur heuristique, puis un algorithme de recherche locale simple utilisant le Simulated Annealing, et finalement des algorithmes plus avancés : les algorithmes génétique et VNS. Les quatre algorithmes ont été capables de trouver des solutions de haute qualité, avec des degrés d'efficacité variables.

Le solveur heuristique a été capable de trouver rapidement une bonne solution en utilisant une simple approche gourmande, mais il a eu du mal à trouver des solutions optimales en raison de la nature locale de sa recherche. Le simulated annealing, quant à lui, a permis d'explorer efficacement un espace de recherche plus grand afin de trouver des solutions de bonne qualité.

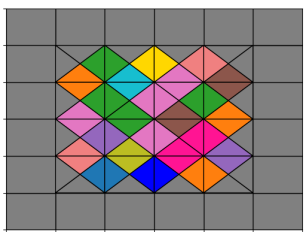
L'algorithme génétique s'est ensuite lui aussi révélé prometteur dans sa capacité à explorer efficacement l'espace de recherche et à améliorer nos solutions, en particulier lorsqu'il est associé à un opérateur de recherche locale et à des mécanismes de préservation de la diversité. Toutefois, il a nécessité un réglage minutieux des paramètres pour équilibrer l'exploration et l'exploitation et éviter une convergence prématurée.

Enfin, l'algorithme VNS a montré un grand potentiel dans sa capacité à combiner la recherche locale avec des stratégies de perturbation pour échapper efficacement aux optima locaux et trouver lui aussi des solutions de qualité.

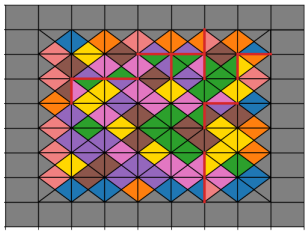
Dans l'ensemble, chacune de ces métaheuristiques a fourni une approche unique pour résoudre l'énigme d'Eternity II, avec ses propres forces et faiblesses. Dans la pratique, le choix de l'algorithme à utiliser peut dépendre de facteurs tels que la taille de l'espace de recherche. Pour résoudre l'instance A par exemple, l'heuristique seule a réussi a résoudre complètement le puzzle mais très rapidement, la complexité du puzzle rend cette approche inefficace pour espérer résoudre l'entièreté du puzzle.

6 Annexe

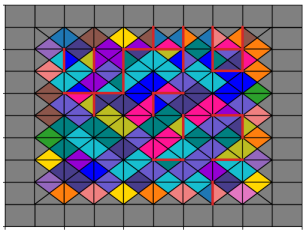
6.1 Construction heuristique



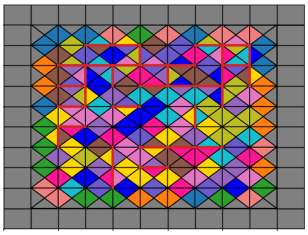
(a) A : 4x4



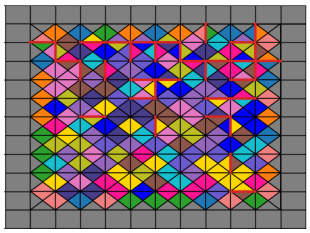
(b) B : 7x7



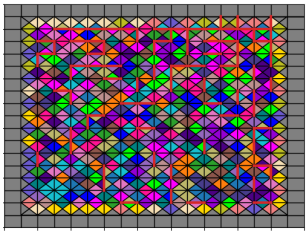
(c) C : 8x8



(d) D : 9x9

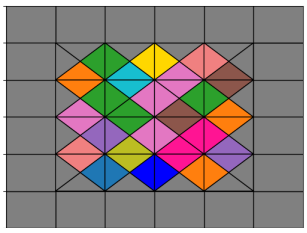


(e) E : 10x10

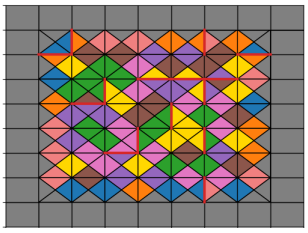


(f) Complet : 16x16

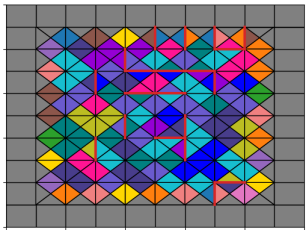
FIGURE 7 – Solutions générées par la construction en lignes



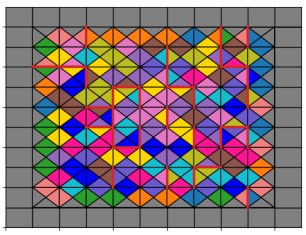
(a) A : 4x4



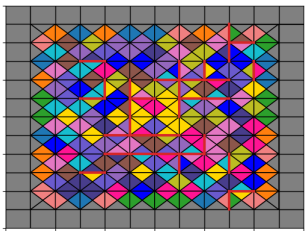
(b) B : 7x7



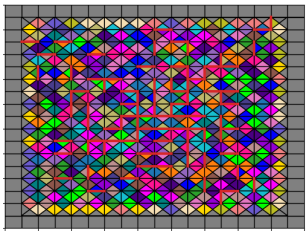
(c) C : 8x8



(d) D : 9x9



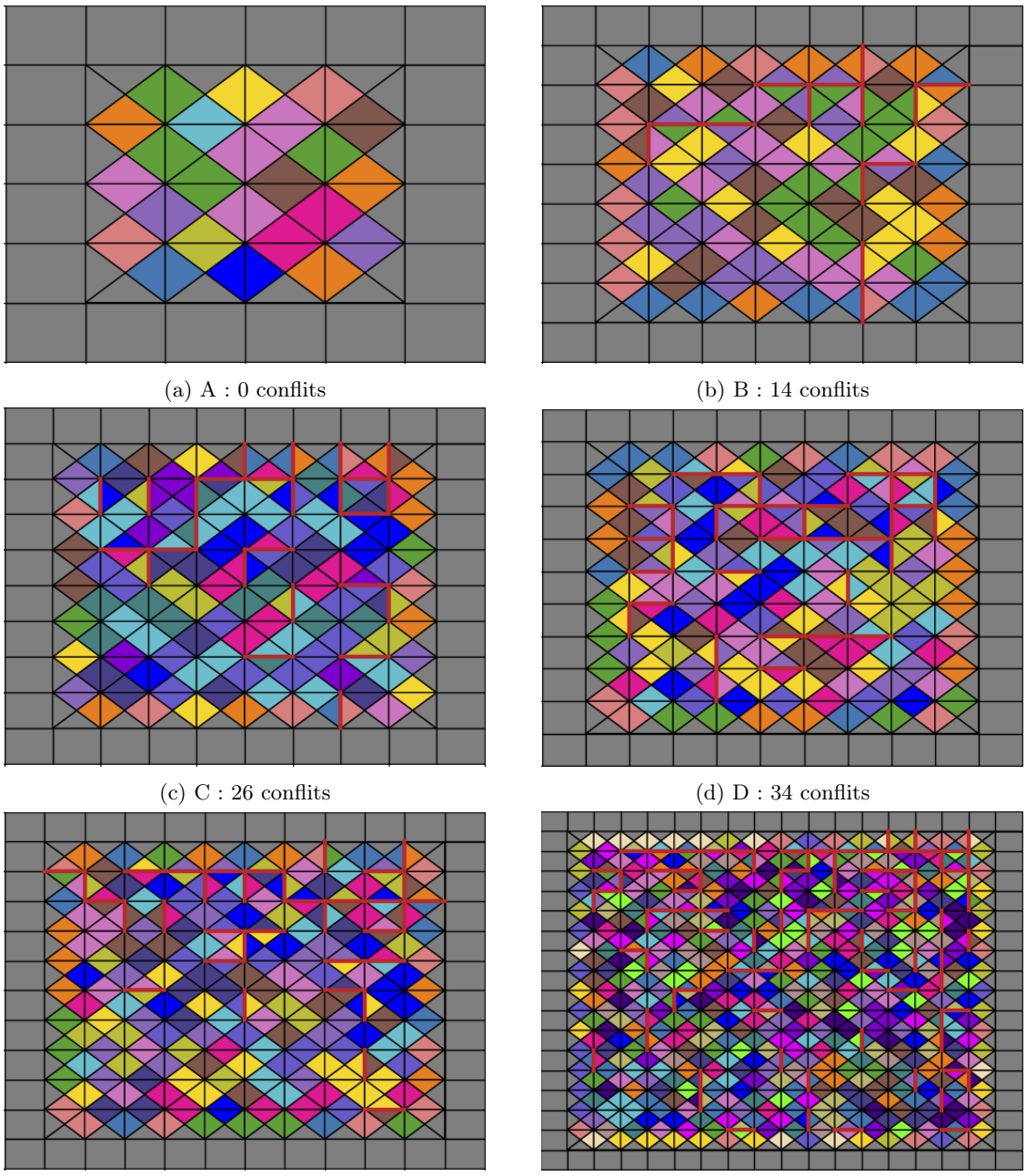
(e) E : 10x10



(f) Complet : 16x16

FIGURE 8 – Solutions générées par la construction en couches

6.2 Visualisation



(a) A : 0 conflits

(b) B : 14 conflits

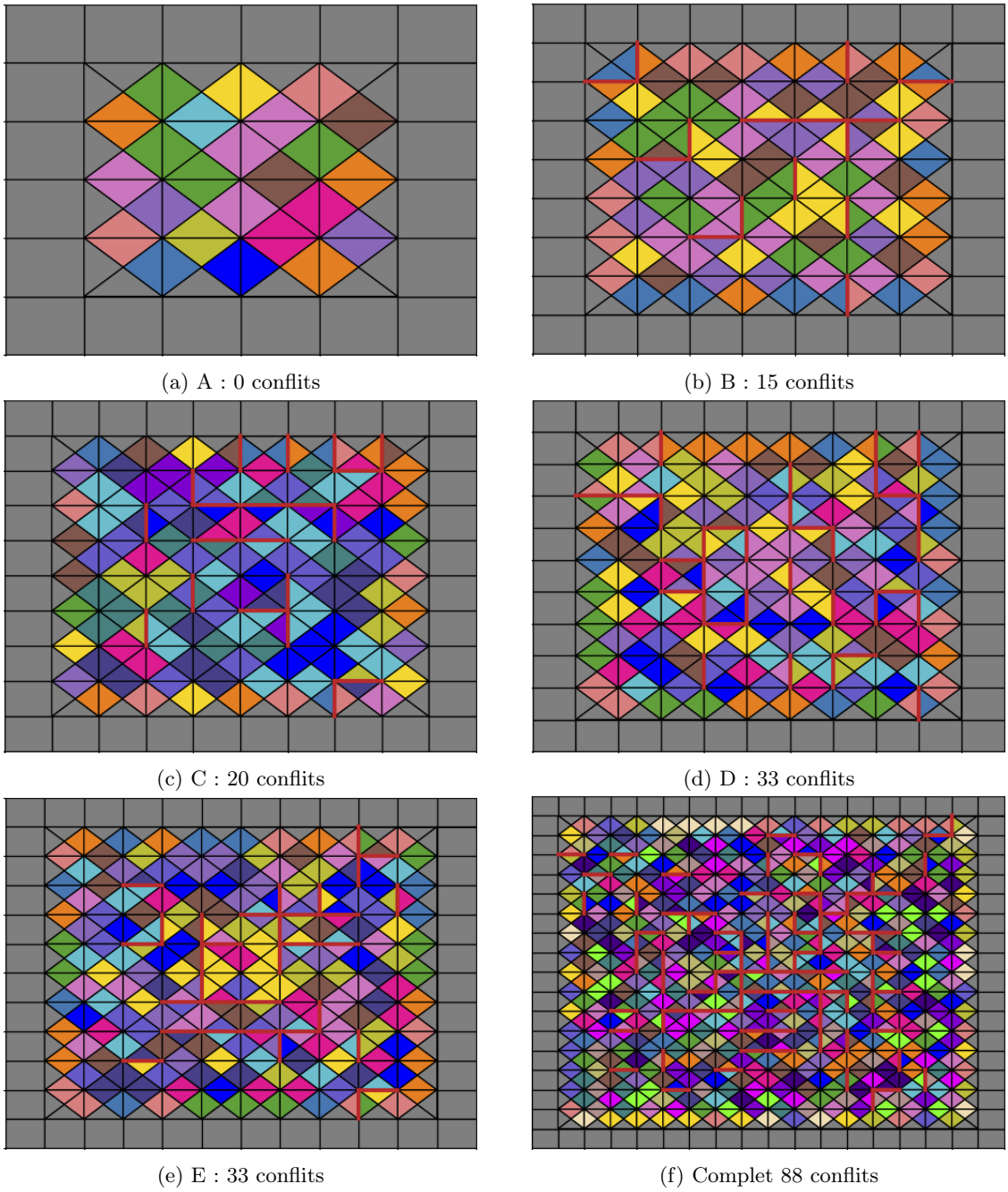
(c) C : 26 conflits

(d) D : 34 conflits

(e) E : 35 conflits

(f) Complet 85 conflits

FIGURE 9 – Résultats visuels - Algorithme génétique



(a) A : 0 conflits

(b) B : 15 conflits

(c) C : 20 conflits

(d) D : 33 conflits

(e) E : 33 conflits

(f) Complet 88 conflits

FIGURE 10 – Résultats visuels - Algorithme de recherche local avec Simulated annealing

6.3 VNS

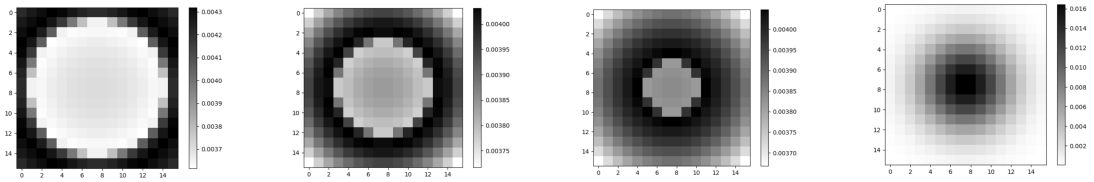
Parmi les algorithmes de recherche que nous avons essayé, l'un d'entre eux est la recherche par voisinage variable. L'objectif de cet algorithme est de procéder à des échanges entre les pièces dans un voisinage n fois, puis de passer au voisinage suivant. Afin d'améliorer la recherche, nous avons implémenté une méthode de redémarrage, qui consiste à altérer la solution autour d'une pièce et à redémarrer la recherche par voisinage variable. Cette méthode permet de déplacer les pièces à travers des voisinages et d'intégrer un mécanisme de diversification. Un autre élément important est l'intégration d'une méthode d'intensification dans l'algorithme VNS, qui consiste à effectuer de nombreux échanges dans chaque voisinage pour augmenter les chances de trouver une solution optimale.

Algorithm 4 Multi-start VNS

```
LEVEL_MAX ← board_widht
ITER_MAX ← 200
stop ← False
p ← 0
while p ≤ LEVEL_MAX and not stop dos
    it ← 0
    while it ≤ ITER_MAX and not stop do
        solution_new ← copy(solution)
        for _ in 0 to p + 2 do
            k ← select_random_neighborhood()
            solution_new ← shake(solution_new, k)          ▷ détruire la solution autour de k
        end for
        solution_new ← vns(solution_new)
        if getNbConflict(solution_new) < getNbConflict(solution) then
            solution ← solution_new
            p ← 0
            it ← 0
        end if
        it ← it + 1
        if time ≥ time_limit then
            stop ← True
        end if
    end while
    p ← p + 1
end while
return solution
```

Algorithm 5 VNS

```
NUMBER_SWAP ← 100
for neighborhood in prioritised_neighborhhods do
    for _ in 0 to NUMBER_SWAP do
        solution ← twoSwap(solution, neighborhood)
        solution ← rotate(solution, neighborhood)    ▷ Tourner une pièce dans le voisinage de façon optimal
    end for
end for
return solution
```



(a) Voisinage #1 (b) Voisinage #2 (c) Voisinage #3 (d) Voisinage #4

FIGURE 11 – Liste de voisinage utilisé par VNS : diagramme de probabilité de sélection

6.3.1 Résultats

Cet algorithme a été implémenté pour l'instance complète, car les autres instances ne pouvaient pas utiliser ce type de voisinage en raison de leur petite taille. Pour l'instance complète, l'algorithme a été testé à partir d'une solution où la bordure était presque optimale, en utilisant l'heuristique de construction de bordure précédemment expliquée, tandis que l'intérieur était construit de manière aléatoire. Les paramètres globaux utilisés étaient : $LEVEL\_MAX$ = la largeur du casse-tête et $ITER\_MAX = 200$. En utilisant cet algorithme, nous avons obtenu 155 conflits.

6.4 LNS

Parmi les algorithmes de recherche que nous avons testés, nous avons développé un algorithme de recherche de voisinage étendu. Ce dernier consiste à détruire la solution en utilisant une distribution de probabilité construite à chaque étape, en fonction du nombre de conflits de chaque pièce et de la proximité avec les pièces déjà enlevé. Ensuite, la solution est reconstruite en plaçant ces pièces selon une heuristique, afin d'éviter une recherche exhaustive, qui serait beaucoup trop longue compte tenu du nombre d'exécutions pour la décomposition et la reconstruction de la solution.

Algorithm 6 LNS

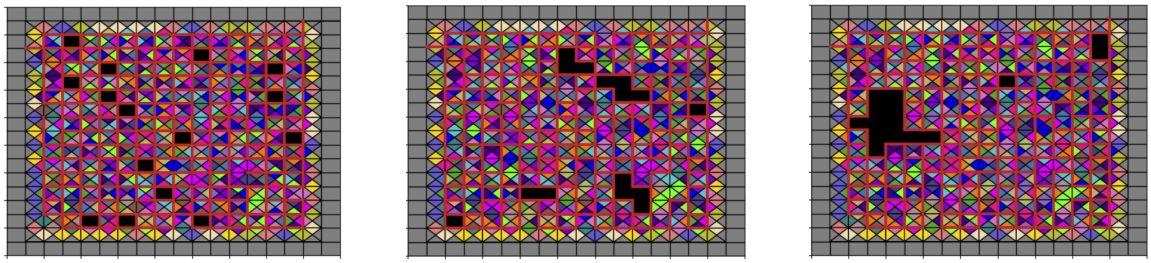
```
init tabu_queue
TABU_QUEUE_SIZE_MAX
neighborhood_size ← 5
while time ≤ TIME_LIMIT do
    solution_new ← rebuild(destroy(solution, neighborhood_size, tabu_queue))
    if getNbConflict(solution_new) < getNbConflict(solution) then
        solution ← solution_new
    end if
    while tabu_queue ≥ TABU_QUEUE_SIZE_MAX do
        pop(tabu_queue)
    end while
end while
return solution
```

Algorithm 7 Destroy

```
init removed_pieces
while size() < neighborhood_size do
    piece ← removePiece(solution)          ▷ au hasard en donnant la priorité aux pièces les plus
conflictuelle.
    if piece not in tabue_queue then
        removed_pieces ← removed_pieces + piece
        push(tabu_queue, piece)
    end if
end while
return solution, removed_pieces
```

Algorithm 8 rebuild

```
holes ← Suffle(holes)
for hole in holes do
    fit_best ← None
    piece_best ← None
    for piece in removed_pieces do
        fit ← findBestRotation(hole, piece
        if fit > fit_best then
            fit_best ← fit
            piece_best ← piece
        end if
    end for
    solution ← place(solution, hole, ,piece, fit)
    removed_pieces ← removed_pieces − piece
end for
return solution
```



(a) Voisinage sans adjacence (b) Voisinage avec adjacence #1 (c) Voisinage avec adjacence #1

FIGURE 12 – Exemple de voisinage

6.4.1 Résultats

Pour évaluer l'efficacité de cet algorithme et déterminer les paramètres optimaux, nous avons appliqué l'heuristique de construction de bordure, comme présenté précédemment et ensuite, nous avons mélangé l'intérieur du puzzle et finalement effectué la recherche LNS. Les paramètres possibles pour cet algorithme incluent la priorisation de la sélection de pièces adjacentes après la première pièce, la taille de la liste tabou, ainsi que la taille du voisinage.

LNS									
Paramètres									
Taille du voisinage	5	10	15	15	15	15	15	20	
Taille de la queue tabu	0	0	0	5	10	15	0	5	
Autoriser et prioriser adjacents	False	False	False	False	False	False	True	True	
Instance									Temps alloué (min)
A	0	0	0	0	0	0	0	0	2
B	23	18	23	22	20	19	20	21	2
C	33	30	30	29	29	29	27	29	2
D	46	50	48	44	47	47	42	44	2
E	54	61	57	56	55	57	57	52	2
Complet	189	202	176	185	185	186	195	206	20