Algorithmique et complexité M1

Université de Bourgogne - UFR Sciences et techniques

Eternity II - 1 étudiantt

Rémy Auloy

encadré par
Romain Raffin

Année 2022 - 2023

Contents

List of Figures

1 Introduction

1.1 Règle du jeu

Concernant les règles du puzzle EternityII, je me suis basé sur la page wikipédia du puzzle [2]. Le puzzle Eternity II est un jeu de placement de bord créé le 28 juillet 2007 par Christopher Monckton. Le puzzle initial se compose de 256 pièces carrées disposées sur une grille de 16 x 16 et soumises à la contrainte de correspondance des bords adjacents. Il est important de noter qu'il existe une couleur de bord spéciale qui impose la position de certaines pièces sur les bords de la grille (dans notre cas, la couleur grise). Les rotations de pièces sont également autorisées. Le puzzle initial se compose donc de 4 pièces de coin, 56 pièces de bord et 196 pièces centrales. Le nombre de configurations qui respectent les contraintes de bord est donc de $4! \times 56! \times 196! \times 4^{196}$

1.2 Généralisation et objectif du projet

Le puzzle Eternity II est un jeu de placement de bord consistant à assembler des pièces carrées de manière à ce que les bords adjacents correspondent. L'objectif de ce projet est de développer un algorithme permettant de générer des puzzles Eternity II solvables de manière aléatoire, de mélanger les pièces de ce puzzle, et de résoudre ce puzzle en utilisant le backtracking pour revenir à la solution initiale. À terme, l'objectif est de trouver une optimisation pour améliorer les performances de notre algorithme.

Mon projet se base sur une version simplifiée des pièces Eternity II, plus proche des MacMahon Squares [1]. Les contraintes de bord resteront les mêmes que celles d'Eternity II, à savoir quatre coins et un nombre de pièces de bord égal à $2(m + n - 4)$, où m et n sont les dimensions du puzzle. Le nombre de pièces intérieures sera quant à lui égal à $(n - 2) \times (m - 2)$. J'ai choisi d'utiliser 10 couleurs de pièces différentes, y compris la couleur grise.

Voici un exemple de piece :

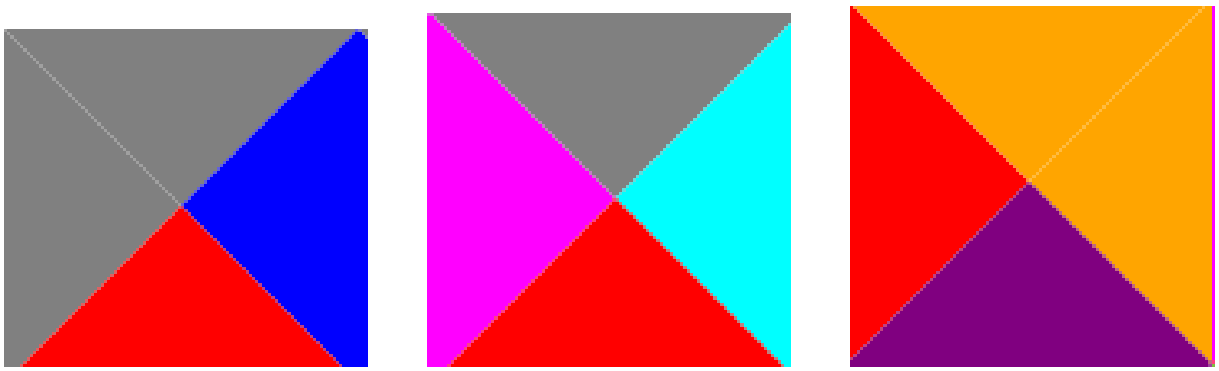

Figure 1: Pièce de coin : Pièce de bord : Pièce intérieurs

2 Génération aléatoire des puzzles

2.1 Choix des couleurs

Dans ce projet, j'ai décidé d'utiliser un total de 10 couleurs différentes, y compris la couleur grise qui sera utilisée pour les bords extérieurs et coin du puzzle. Les couleurs sont représentées par des entiers, et nous utilisons une liste pour stocker ces couleurs :

```
let colors = [1; 2 ; 3; 4; 5; 6; 7; 8; 9]
```

Nous utilisons également une hasht-able *color_table* pour stocker les associations entre les entiers et les couleurs réelles. Cette table permet à la fonction *piece_to_svg* d'accéder aux couleurs correspondantes lors de la génération du fichier SVG.

2.2 Algorithme de génération aléatoire

Pour générer un puzzle solvable aléatoirement, j'ai implémenté la fonction `generate_solvable_puzzle` qui prend en entrée les dimensions du puzzle et retourne un puzzle solvable. L'algorithme commence par initialiser un puzzle vide de dimensions `n` et `m` avec des cases de type `piece option`, où `None` représente une case vide et `Some piece` représente une case contenant une pièce.

La fonction `random_edge` est utilisée pour générer un bord aléatoire en choisissant une couleur au hasard dans la liste des couleurs. La fonction `random_piece` génère quant à elle une pièce aléatoire en assignant une couleur aléatoire à chacun de ses bords.

Ensuite, nous parcourons les cases du puzzle en utilisant des boucles imbriquées et assignons des pièces à chaque case en respectant les contraintes de bord suivantes :

- Si nous sommes sur le bord supérieur du puzzle, nous assignons la couleur grise au bord supérieur de la pièce.
- Si nous sommes sur le bord gauche du puzzle, nous assignons la couleur grise au bord gauche de la pièce.
- Si nous sommes sur le bord inférieur du puzzle, nous assignons la couleur grise au bord inférieur de la pièce.
- Si nous sommes sur le bord droit du puzzle, nous assignons la couleur grise au bord droit de la pièce.

De plus, j'assure que les bords adjacents correspondent en réutilisant les bords des pièces voisines lors de la génération d'une nouvelle pièce.

Algorithm 1 Algorithme de génération d'un puzzle solvable aléatoire

```
function GENERATE_SOLVABLE_PUZZLE(n, m)
    Créer puzzle vide de dimensions n × m
    for i ← 0 to n − 1 do
        for j ← 0 to m − 1 do
            Créer une pièce vide
            if i = 0 then
                Assigner la couleur grise au bord supérieur de la pièce
            else
                Copier la couleur du bord inférieur de la pièce (i − 1, j) sur le bord supérieur de la pièce
            end if
            if j = 0 then
                Assigner la couleur grise au bord gauche de la pièce
            else
                Copier la couleur du bord droit de la pièce (i, j − 1) sur le bord gauche de la pièce
            end if
            if i = n − 1 then
                Assigner la couleur grise au bord inférieur de la pièce
            else
                Assigner une couleur aléatoire au bord inférieur de la pièce
            end if
            if j = m − 1 then
                Assigner la couleur grise au bord droit de la pièce
            else
                Assigner une couleur aléatoire au bord droit de la pièce
            end if
            Ajouter la pièce au puzzle à la position (i, j)
        end for
    end for
    return puzzle
end function
```

Vous pouvez visualiser l'algorigramme en annexe. Voir 5.

2.3 Évaluation de la qualité des puzzles générés

Pour évaluer la qualité des puzzles générés, j'ai implémenté des fonctions d'affichage et de visualisation. La fonction `print_puzzle` permet d'afficher le puzzle dans la console en affichant les couleurs des bords de chaque pièce. Cela permet de vérifier rapidement si les contraintes de bord sont respectées.

Pour une visualisation plus intuitive, j'ai également implémenté une fonction pour générer un fichier SVG à partir d'un puzzle. La fonction `pieces_to_svg` génère le code SVG pour chaque pièce du puzzle en utilisant la fonction `piece_to_svg`. Ensuite, la fonction `print_svg` permet d'écrire le code SVG complet dans un fichier spécifié, qui peut être ouvert dans un navigateur web pour visualiser le puzzle.

Avec ces fonctions, nous pouvons vérifier que les puzzles générés sont solvables et qu'ils respectent les contraintes de bord. De plus, ces visualisations nous permettent d'évaluer la qualité des puzzles générés en termes de diversité et de complexité.

3 Mélange des pièces

Dans cette partie, je vais aborder le mélange des pièces du puzzle. Effectivement, une fois que nous avons un puzzle avec une solution, nous pouvons le mélanger pour le résoudre par la suite avec notre algorithme de backtracking. Pour cela, j'ai étudié différentes techniques de mélange et j'ai choisi celle qui me semblait la plus appropriée pour le projet. Je vais également évaluer la qualité des puzzles mélangés et expliquer pourquoi cette méthode a été retenue.

3.1 Techniques de mélange

Pour le mélange des pièces, j'ai pensé à plusieurs méthodes de mélange. La plus évidente selon moi est de mélanger les pièces en les permutant deux à deux de manière aléatoire. J'ai également pensé à effectuer une rotation aléatoire de la pièce avant de les permuter, ou encore à diviser le puzzle en plusieurs groupes de pièces et à déplacer ces groupes de manière aléatoire.

3.2 Choix de la méthode de mélange

En comparant ces méthodes, j'en suis arrivé à choisir la méthode de permutation des pièces deux à deux de manière aléatoire. Quels ont été mes points de comparaison ?

- La complexité
- La facilité de mise en oeuvre
- L'efficacité du mélange

3.2.1 Permutation aléatoire entre deux pièces

La méthode de permutation aléatoire entre deux pièces présente une complexité en $O(n * m)$, où n et m sont les dimensions du puzzle. Chaque élément est visité une fois et échangé avec une autre pièce sélectionnée aléatoirement. Cette méthode est relativement simple à mettre en œuvre, car il suffit de parcourir toutes les pièces et d'effectuer une opération d'échange. Cette méthode ne permet peut-être pas d'avoir un mélange parfait néanmoins celui-ci est généralement efficace et le nombre d'itérations sera relativement modéré, donc rapide.

3.2.2 Méthode des rotations

La méthode des rotations a une complexité en $O(n * m)$, car chaque élément doit être visité une fois pour effectuer la rotation. Toutefois, la rotation de certaines pièces peut nécessiter des calculs supplémentaires pour conserver la cohérence du puzzle. La mise en œuvre de cette méthode est plus complexe que la permutation aléatoire, car elle nécessite de prendre en compte la cohérence du puzzle et des pièces lors de la rotation. La rotation peut offrir un bon mélange, mais elle est limitée par le fait que les pièces restent dans leur position d'origine. La méthode peut ne pas être suffisante pour créer un mélange suffisamment désordonné. Et Si nous venions a effectuer une permutation aléatoire entre deux pièces après une rotation, cela serait pas viable car ajouterai de la complexité à la première méthode, cela sera donc moins performant. D'ailleurs j'ai essayé de faire cela malheureusement je n'arrivais pas à obtenir un algorithme qui résolvait correctement le mélange, mais nous y reviendrons plus tard.

3.2.3 Déplacement de groupe aléatoire

La méthode de déplacement de groupe aléatoire présente également une complexité en $O(n * m)$, car chaque élément doit être visité au moins une fois pour effectuer le déplacement. Cependant, la sélection et le déplacement de groupes peuvent également augmenter la complexité. Cette méthode est plus complexe à mettre en œuvre que la permutation aléatoire, car elle nécessite la sélection et la gestion de groupes de pièces. De plus, elle demande un traitement supplémentaire pour conserver la cohérence du puzzle lors du déplacement des groupes. Bien que cette méthode puisse fournir un mélange intéressant, elle est plus difficile à implémenter et à contrôler que les deux autres méthodes. Comme nous aimons faire au plus simple, je ne retiendrais pas cette méthode.

3.3 Évaluation de la qualité des puzzles mélangés

Après avoir comparé les trois méthodes de mélange, j'ai choisi la méthode de permutation aléatoire entre deux pièces pour les raisons suivantes:

- La complexité de cette méthode est en $O(n * m)$, ce qui est similaire aux autres méthodes, mais sans les calculs supplémentaires et la gestion de la cohérence du puzzle.

- La facilité de mise en œuvre est un point fort de cette méthode, car il suffit de parcourir les pièces et d'effectuer un échange aléatoire.
- L'efficacité du mélange est généralement bonne, même si elle ne garantit pas un mélange parfait. Dans la plupart des cas, un mélange raisonnable est obtenu en un nombre modéré d'itérations.

J'ai évalué la qualité des puzzles mélangés en analysant la répartition des pièces après le mélange. Dans la plupart des cas, les pièces étaient suffisamment mélangées pour offrir un défi intéressant lors de la résolution. Toutefois, pour des puzzles de petite taille comme des puzzles de 2×2, il est possible que le mélange résolve le puzzle, mais étant donné que notre sujet concerne la résolution de puzzles de 12×12, cela n'a pas été le cas et le mélange a correctement mélangé toutes les pièces. La simplicité de cette méthode et son efficacité sur les puzzles de grande taille font de cette méthode la meilleure pour notre projet.

En conclusion, la méthode de permutation aléatoire entre deux pièces s'est avérée être le meilleur choix pour mélanger efficacement et simplement les puzzles pour le projet.

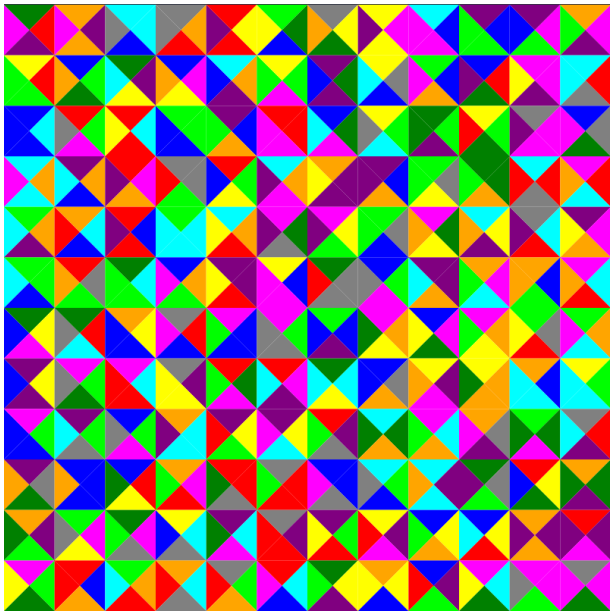

Figure 2: Exemple d'un puzzle 12x12 mélangé

4 Résolution par backtracking

4.1 Explication du backtracking

Le backtracking est un algorithme qui permet d'explorer toutes les solutions possibles en revenant sur ses pas dès qu'une solution partielle ne peut pas converger vers une solution complète. Cet algorithme est particulièrement utile pour résoudre des problèmes de recherche ou d'optimisation, ce qui convient parfaitement à notre cas de résolution de puzzle.

Le principe consiste à parcourir l'ensemble des solutions possibles en suivant un chemin spécifique, tel qu'un chemin de gauche à droite et de haut en bas. Cette approche facilitera la mise en œuvre de l'algorithme. D'autres itinéraires, tels que ceux en sens inverse ou en spirale, donneront des résultats équivalents, car chaque pièce comporte le même nombre de contraintes. De plus, il convient de noter que la somme des contraintes est la même pour tous les puzzles finaux.

L'algorithme de backtracking utilise une fonction récursive qui prend en entrée la solution partielle courante et renvoie la solution complète une fois qu'elle est trouvée. Cette fonction explore toutes les solutions possibles en ajoutant des décisions à la solution partielle en cours, puis en vérifiant si cette nouvelle solution partielle est valide. Si la solution partielle n'est pas valide, la fonction récursive revient en arrière pour essayer une autre décision. Le backtracking s'arrête lorsque toutes les solutions ont été explorées ou lorsque la solution complète est trouvée.

Cependant, le backtracking n'est pas une solution miracle et présente des inconvénients, notamment pour les puzzles de grandes tailles. En effet, le temps de résolution peut varier considérablement selon la complexité du puzzle. Parfois, il peut trouver une solution en quelques secondes, alors que d'autres fois, il peut prendre beaucoup plus de temps. De plus, le backtracking peut être coûteux en termes de mémoire, car il stocke toutes les solutions partielles précédentes. Néanmoins, des solutions existent, notamment l'utilisation d'heuristiques pour améliorer le temps de résolution de l'algorithme.

4.2 Implémentation et complexité de l'algorithme

4.2.1 Implémentation de l'algorithme

Voici le pseudo-code simplifié de mon algorithme de backtracking. Veuillez noter que j'ai supprimé certains éléments verbeux pour faciliter la compréhension. Dans la réalité, le code utilise le backtracking, qui est remplacé ici par l'opérateur $forall$, et utilise des fonctions internes à solve qui ne sont pas présentées ici.

Algorithm 2 Fonction solve qui résout le puzzle par backtracking naif

```
1: function SOLVE(puzzle, pieces, i, j)
2:     if i = n and j = 0 then
3:         return Some(puzzle)
4:     else
5:         calculer next_i et next_j
6:         for all piece ∈ pieces do
7:             if piece_fits(puzzle, piece, i, j) then
8:                 updated_puzzle ← puzzle ∪ {piece}
9:                 result ← solve(updated_puzzle, pieces \ {piece}, next_i, next_j)
10:                if result ≠ None then
11:                    return result
12:                end if
13:            end if
14:        end for
15:        return None
16:    end if
17: end function
```

Vous pouvez visualiser l'algorigramme en annexe. Voir 6.
Expliquons notre algorithme de backtracking simplifié :

Pour commencer, nous vérifions si la fonction *solve* a atteint la fin du puzzle en comparant les indices i et j avec les dimensions du puzzle n et m. Si c'est le cas, le puzzle complet est renvoyé en tant que solution.

Sinon, nous calculons la prochaine position à tester en déterminant $next_i$ et $next_j$, qui renvoie la case à droite si la ligne n'est pas finie, sinon la première case de la ligne suivante. Ensuite, nous parcourons chaque pièce restante de la liste pieces et vérifions si elle peut être placée à la position actuelle en utilisant la fonction $piece_fits$.

Si une pièce peut être placée, nous créons une copie du puzzle mis à jour en ajoutant la nouvelle pièce et appelons récursivement *solve* avec la liste de pièces restantes et la position suivante. Si la fonction *solve* renvoie une solution, nous la renvoyons également. Sinon, nous passons à la pièce suivante de la liste pieces.

Cet algorithme de backtracking simplifié fonctionne correctement sur les puzzles de petite taille. Cependant, pour des puzzles plus grands, tels que des puzzles de taille 10×10, 11×11 ou 12×12, l'algorithme devient très lent. Cela est dû au fait que plus le puzzle est grand, plus il y a de pièces et donc plus il y a de combinaisons à tester. Les limites de cet algorithme deviennent donc évidentes. Par exemple, pour un puzzle de taille 10×10, le nombre de combinaisons théoriquement possibles est de 10!, soit environ $9,33 \times 10^{157}$.

4.2.2 Complexité de l'algorithme

La complexité en temps de notre algorithme de résolution de puzzle solve est de l'ordre de $O((n \times m) \times k!)$. Cette complexité prend en compte le nombre total de cases du puzzle $(n \times m)$ ainsi que le nombre de pièces restantes à placer k.

Pour expliquer cette complexité, considérons que pour chaque case du puzzle, l'algorithme essaie de placer une pièce parmi les k pièces restantes. Ensuite, il y a $(k-1)$ pièces restantes pour la case suivante, puis $(k-2)$ pièces pour la case après, et ainsi de suite. Le nombre total de combinaisons de pièces à essayer est donc $k!$.

Ainsi, pour un puzzle de 10×10 avec 100 pièces, la complexité en temps de l'algorithme solve est de l'ordre de $O((100) \times 100!)$ qui est effectivement un très grand nombre. Cela montre que cet algorithme n'est pas optimal pour résoudre de grands puzzles en raison de sa complexité élevée.

Dans la pratique, l'algorithme ne testera pas toutes ces combinaisons, car il reviendra en arrière avec le backtracking lorsque certaines pièces ne correspondent pas. Cependant, cela illustre la complexité de l'algorithme et explique pourquoi il peut être lent pour des puzzles de grande taille. Pour résoudre de tels puzzles, des approches plus efficaces peuvent être nécessaires, telles que des heuristiques ou des algorithmes d'optimisation.

4.3 Évaluation des performances de l'algorithme de backtracking

Analysons les performances de notre algorithme de backtracking. Toutefois, avant de procéder à l'analyse, il convient de préciser que les résultats ci-dessous ont été obtenus à partir de la moyenne de 5 lancements de la résolution de puzzles de tailles diverses allant de 2×2 à 12×12.

taille	temps1	temps2	temps3	temps4	temps5	Moyenne
2x2	0	0	0	0	0	0
3x3	0	0	0	0	0	0
4x4	0	0	0	0	0	0
5x5	0	0	0	0	0	0
6x6	0	0	0	0	0	0
7x7	0	0	0	0,015625	0	0,003125
8x8	0	0	0	0	0	0
9x9	0,015625	0	0,015625	0,03125	0,015625	0,015625
10x10	0	0	0,015625	0,015625	0,015625	0,009375
11x11	0,265625	0,0625	0,671875	1,078125	0,78125	0,571875
12x12	52,796875	8,140625	113,484788	165,640625	57,859373	79,5844572

Table 1: Temps pour réaliser un puzzle de taille $n \times n$ avec notre algorithme.

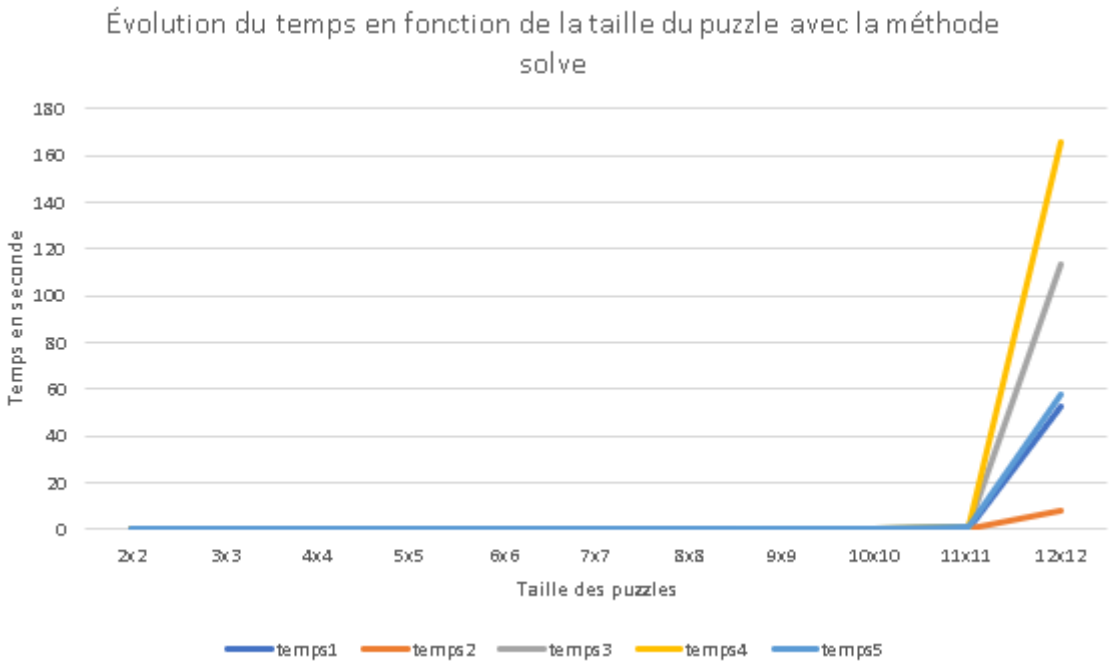

Figure 3: Évolution du temps en fonction de la taille du puzzle avec la méthode solve

Comme nous pouvions nous y attendre, plus le puzzle est grand, plus il met du temps à se résoudre, notamment pour les puzzles de taille 12×12. Il est également important de noter que, pour des puzzles allant de 6×6 à 12×12, il arrive que l'algorithme ne trouve pas de solution alors que celle-ci existe. En effet, notre fonction `generate_solvable_puzzle` génère une solution, et nous nous basons sur le mélange de ce résultat pour résoudre l'algorithme.

J'ai d'abord supposé que cela pouvait être dû à la complexité du mélange. Cependant, en théorie, cela ne devrait pas être le cas étant donné que nous ne faisons pas de rotation lors du mélange. Après avoir testé de relancer la résolution sur un nouveau mélange, si aucune solution n'a été trouvée lors de la première tentative, et ce, plusieurs fois, l'algorithme ne trouvait toujours pas de solution. De plus, il est surprenant de constater que cela peut arriver parfois, alors que d'autres fois, cela fonctionne très bien pour d'autres puzzles de la même taille.

j'en conclu donc que l'algorithme présente certaines limites en fonction des configurations du puzzle générées par `generate_solvable_puzzle`. Cette situation est également liée au fait que l'algorithme est naïf et fonctionne par force brute. Par conséquent, l'arbre des solutions possibles est très large, ce qui explique pourquoi, dans certains cas, l'algorithme ne parvient pas à trouver de solution, alors que dans d'autres cas, il y parvient.

Une autre raison possible est l'efficacité de la fonction de mélange, qui peut générer des configurations de puzzle plus ou moins difficiles à résoudre. Il est également possible que l'ordre dans lequel les pièces sont testées lors du backtracking ait un impact sur la performance de l'algorithme, certaines combinaisons de pièces pouvant conduire plus rapidement à une solution.

Pour palier a cela nous verrons plus tard que nous pouvons optimiser notre algorithme et le rendre plus intelligent.

5 Optimisation de l'algorithme

5.1 Principe du nouvel algorithme

Algorithm 3 Algorithme de résolution optimisé du puzzle

```
1:  function SOLVE_ALL(puzzle, pieces, i, j)
2:      n, m ← dimensions du puzzle
3:      corners, borders, inner ← séparer les pièces en coins, bords et intérieures
4:      function TRY_CANDIDATES(candidates, next_i, next_j)
5:          if candidates est vide then
6:              return None
7:          else
8:              candidate ← premier élément de candidates
9:              candidates_tail ← reste des éléments de candidates
10:             if candidate s'adapte au puzzle en position (i, j) then
11:                 updated_puzzle ← puzzle avec candidate placé en position (i, j)
12:                 new_remaining_pieces ← liste des pièces restantes sans candidate
13:                 solution ← SOLVE_ALL(updated_puzzle, new_remaining_pieces, next_i, next_j)
14:                 if solution existe then
15:                     return solution
16:                 else
17:                     return TRY_CANDIDATES(candidates_tail, next_i, next_j)
18:                 end if
19:             else
20:                 return TRY_CANDIDATES(candidates_tail, next_i, next_j)
21:             end if
22:         end if
23:     end function
24:     if i = n et j = 0 then
25:         return puzzle (solution trouvée)
26:     else
27:         next_i, next_j ← calculer les coordonnées de la position suivante
28:         if (i, j) est un coin then
29:             candidates ← corners
30:         else if (i, j) est un bord then
31:             candidates ← borders
32:         else
33:             candidates ← inner
34:         end if
35:         return TRY_CANDIDATES(candidates, next_i, next_j)
36:     end if
37: end function
```

Vous pouvez visualiser l'algorigramme en annexe. Voir 7.

L'algorithme *solve_all* est une optimisation de l'algorithme *solve*. Plutôt que de tester chaque pièce pour chaque emplacement, il utilise une approche plus efficace et moins naive. Tout d'abord, les pièces sont séparées en trois catégories : les coins, les bords et les pièces intérieures. Ensuite, pour chaque position du puzzle, seules les pièces correspondant à cette position en termes de catégorie sont testées. En d'autres termes, seules les pièces de la catégorie des coins sont essayées pour les positions correspondant aux coins, uniquement les pièces des bords pour les positions des bords, et uniquement les pièces intérieures pour les positions intérieures.

En comparaison de l'algorithme *solve*, la complexité de l'algorithme *solve_all* est plus difficile à déterminer en raison de la séparation des pièces en différentes catégories. Cependant, dans le pire des cas, la complexité de cet algorithme reste de l'ordre de $O(n \times m \times k!)$, car il doit parcourir toutes les pièces dans chaque catégorie. Néanmoins, en réduisant l'espace de travail, l'algorithme élimine de nombreuses combinaisons, ce qui devrait le rendre beaucoup plus rapide en théorie.

5.2 Évaluation de la performance du nouvel algorithme

Analysons les performances de notre nouvel algorithme optimisé. Tout comme pour l'algorithme de backtracking, les résultats ci-dessous ont été obtenus à partir de la moyenne des temps de résolution de puzzles de tailles diverses allant de 2×2 à 12×12. Cependant, contrairement à l'algorithme précédent, nous avons réalisé la moyenne sur 10 résolutions réussies pour chaque taille de puzzle, car ce nouvel algorithme est beaucoup plus rapide et trouve la solution plus efficacement.

taille	temps1	temps2	temps3	temps4	temps5	temps6	temps7	temps8	temps9	temps10	Moyenne
2x2	0	0	0	0	0	0	0	0	0	0	0
3x3	0	0	0	0	0	0	0	0	0	0,015625	0,0015625
4x4	0	0	0	0	0	0	0	0	0	0	0
5x5	0	0	0	0	0	0	0	0	0	0	0
6x6	0	0	0	0	0	0,015625	0	0	0	0	0,0015625
7x7	0	0	0	0	0	0	0	0	0	0	0
8x8	0	0	0	0	0	0	0	0	0	0	0
9x9	0	0,015625	0	0	0	0	0	0	0	0	0,0015625
10x10	0,015625	0	0	0,03125	0	0	0,015625	0	0	0	0,00625
11x11	0,015625	0	0,015625	0,015625	0,015625	0,015625	0	0	0,03125	0,0625	0,0171875
12x12	0,234375	1,03125	0,125	1,328125	0,1875	6.78125	0,078125	0,03125	0,0625	1,234375	0,479166667

Table 2: Temps pour réaliser un puzzle de taille $n \times n$ avec notre algorithme optimisé.

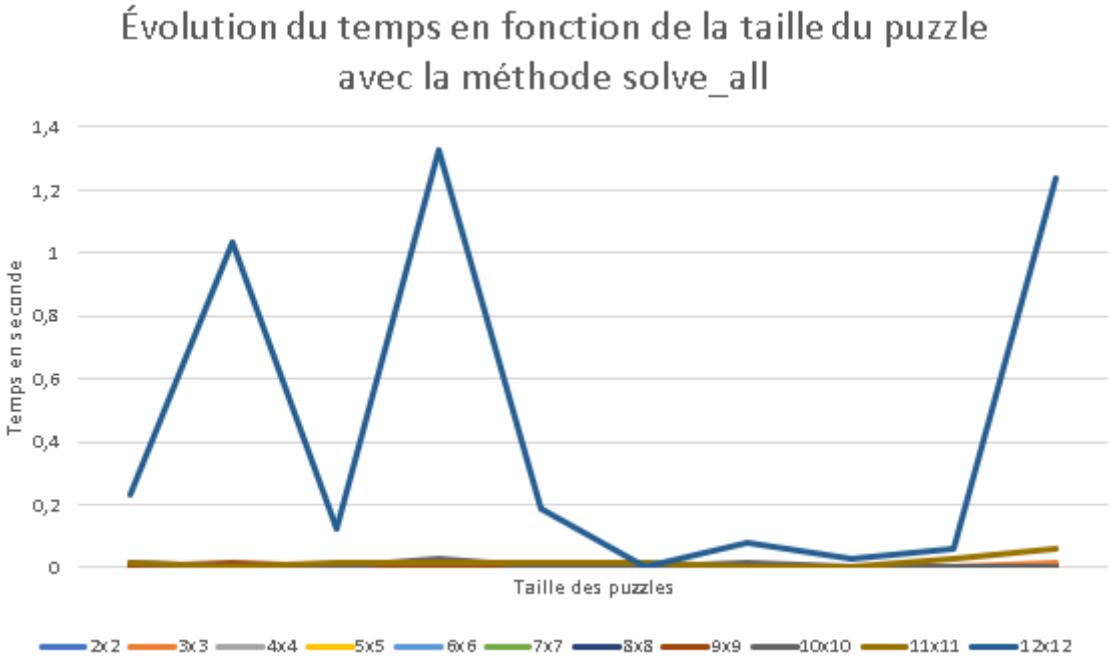

Figure 4: Évolution du temps en fonction de la taille du puzzle avec la méthode *solve_all*

Comme nous pouvons le constater dans le tableau, les temps de résolution pour le nouvel algorithme sont beaucoup moins grand que ceux de l'algorithme de backtracking naïf (voir tableau 1). En effet, les temps de résolution sont généralement proches de zéro pour les puzzles de petite taille et augmentent légèrement pour les puzzles plus grands. Même pour les puzzles de taille 12×12, l'algorithme optimisé résout les puzzles en un temps nettement inférieur à celui de l'algorithme naïf.

Ces améliorations sont principalement dues au fait que nous savons quel type de pièce placé pour la cellule en cours de traitement ce qui rend l'algorithme plus intelligent et moins dépendant de la force brute. Ainsi, l'algorithme est capable de résoudre efficacement des puzzles de tailles diverses, même ceux pour lesquels l'algorithme de backtracking naïf ne trouvait pas de solution.

Il est également important de souligner que, grâce à ces optimisations, l'algorithme est désormais capable de trouver systématiquement une solution pour les puzzles générés par la fonction *generate_solvable_puzzle*, ce qui n'était pas toujours le cas avec l'algorithme naïf.

En somme, ce nouvel algorithme optimisé présente des performances bien supérieures à celles de l'algorithme de backtracking naïf. Les temps de résolution sont fortement réduits et l'algorithme est capable de résoudre de manière plutot fiable des puzzles de différentes tailles. Cette amélioration est principalement due à l'introduction de techniques et de stratégies intelligentes qui permettent à l'algorithme de résoudre les puzzles de manière plus efficace, sans recourir à la force brute.

6 Conclusion

6.1 Résumé des résultats obtenus

Au cours de ce projet, j'ai développé un programme pour générer, mélanger et résoudre des puzzles EternityII. Les algorithmes ont été conçus pour gérer les différentes étapes du processus, notamment la génération de puzzles solvables, le mélange des pièces et la recherche d'une solution à partir du puzzle mélangé. Les résultats obtenus montrent que le programme est capable de résoudre efficacement les puzzles dans la majorité des cas.

6.2 Pistes d'amélioration

Plusieurs pistes d'amélioration peuvent être envisagées pour optimiser et donc améliorer le programme :

- Prendre en compte les rotations des pièces dans le $shuffle\_puzzle$ et les algorithmes de résolution : certes cela augmenterait la complexité, cependant cela rendrait le défi plus intéressant.
- Utiliser des heuristiques plus performantes : les méthodes actuelles pourraient être améliorées en intégrant des approches heuristiques plus avancées, ce qui pourrait permettre de résoudre des puzzles plus complexe et donc d'éviter les cas ou notre programme ne trouvait pas de solution.
- Diviser le programme en différents modules pour faciliter la maintenance, la modularité et la réutilisabilité du code, permettant ainsi une évolution plus fluide.

En conclusion, le programme développé au cours de ce projet offre une base solide pour la génération, le mélange et la résolution de puzzles Eternity II. Cependant, il faut prendre en compte que plus le puzzle est de grande taille, plus la résolution par backtracking sera compliquée. Il serait donc utile d'utiliser d'autres méthodes de résolution, comme par exemple la programmation linéaire ou les solveurs SAT, qui pourraient offrir de meilleures performances et une résolution plus efficace pour traiter des puzzles de dimensions plus importantes.

7 Annexes

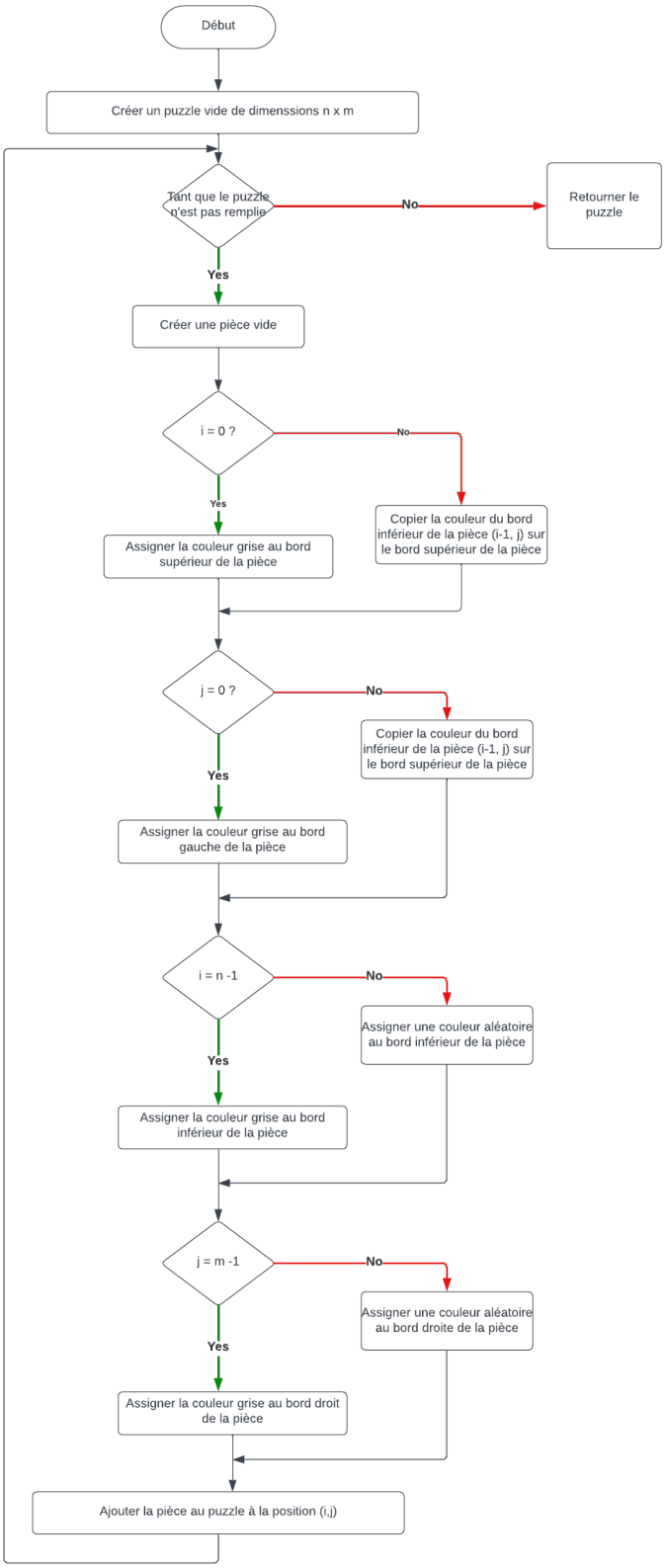

Figure 5: algorigramme de *generate_solvable_puzzle*

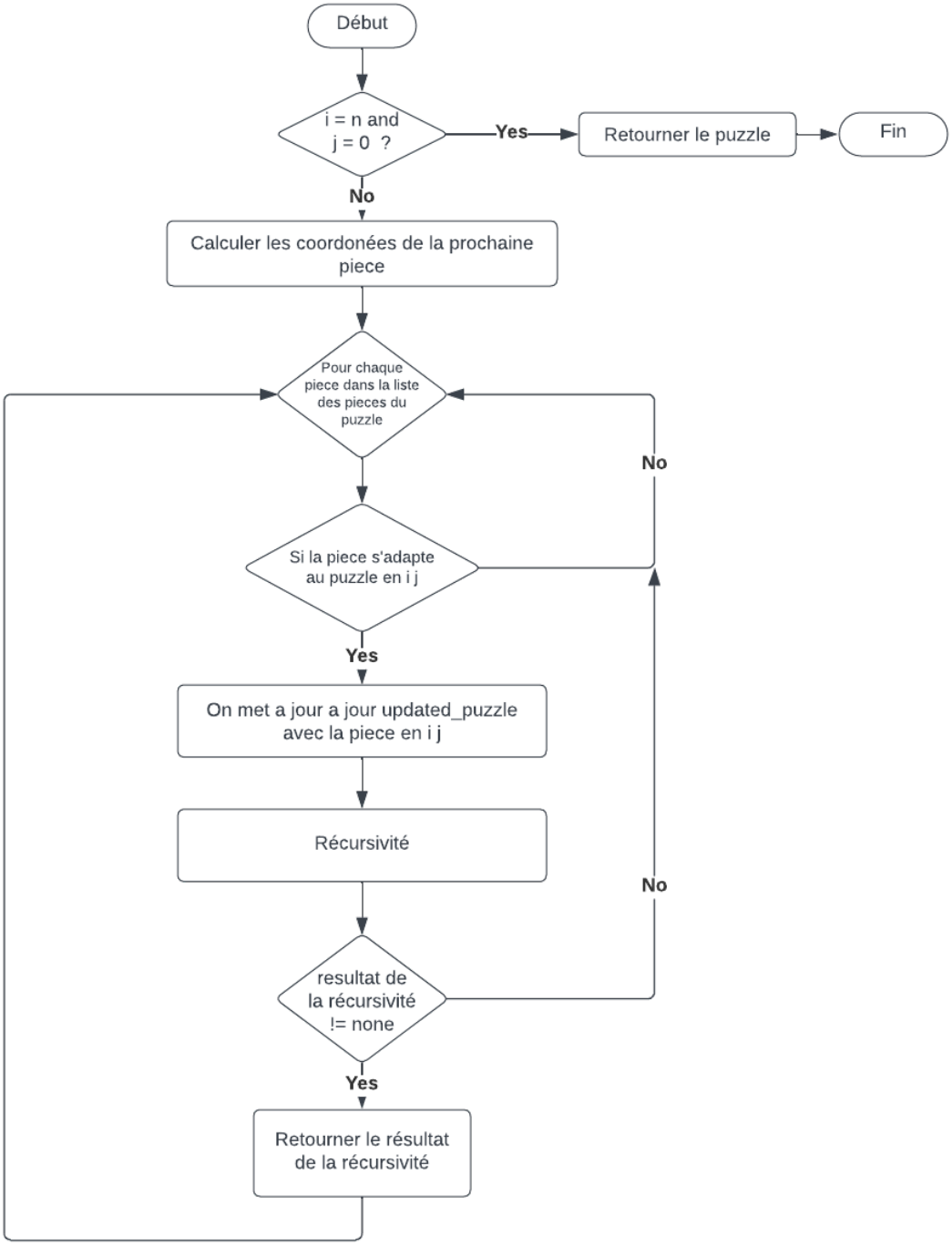

Figure 6: algorigramme de *solve*

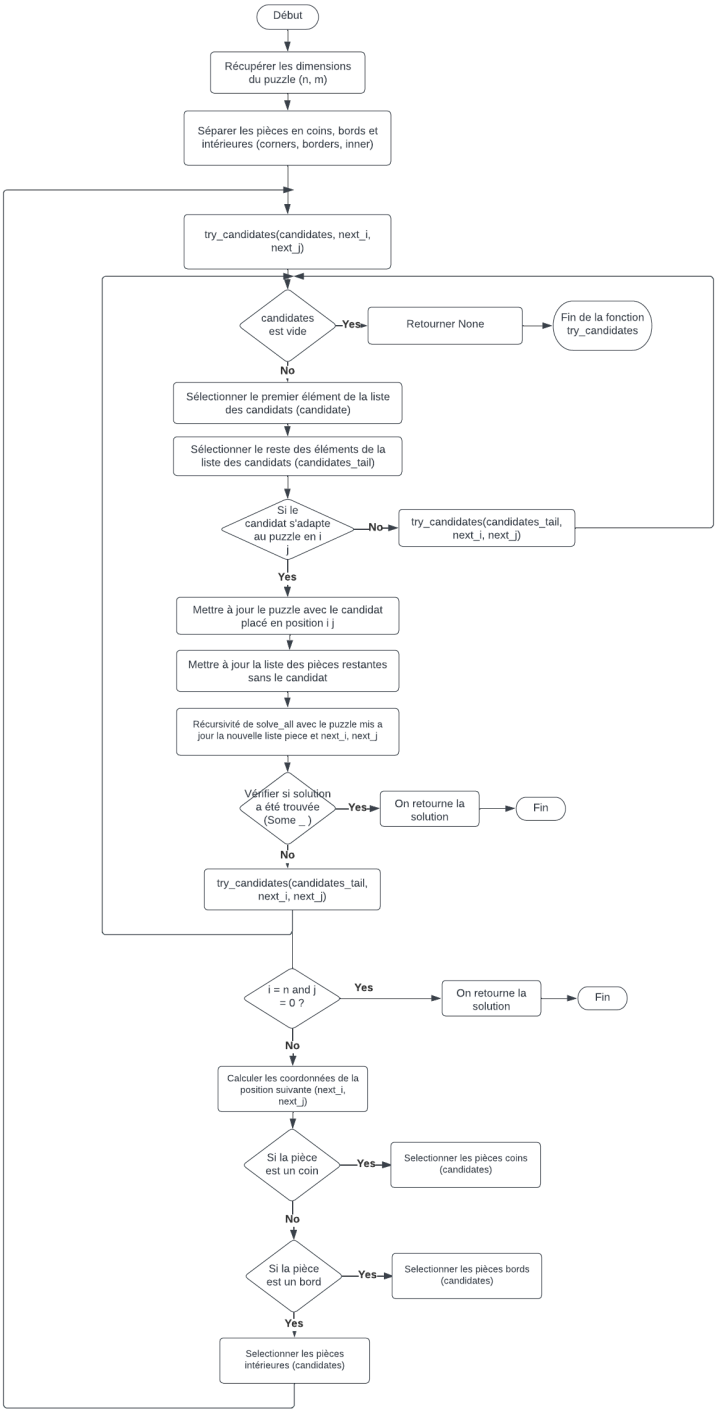

Figure 7: algorigramme de *solve_all*

References

[1] Wikepdia contributors. *MacMahon Squares*. 2023. URL: https://en.wikipedia.org/wiki/MacMahon_Squares.

[2] Wikipedia contributors. *Eternity II puzzle*. 2023. URL: https://en.wikipedia.org/wiki/Eternity_II_puzzle.