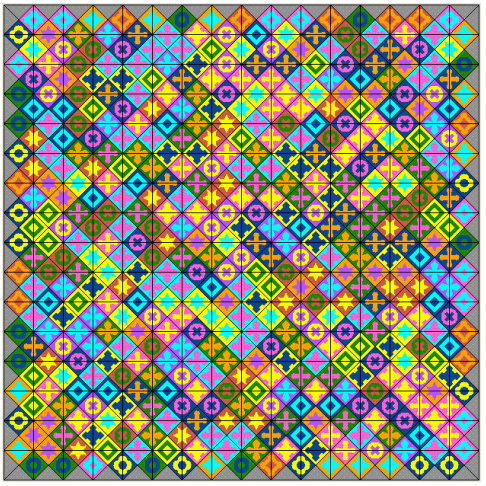
IMT Nord Europe
École Mines-Télécom
IMT-Université de Lille



MAD - Optimisation combinatoire

Projet Eternity

Groupe :

- Adrien Wils
- Mathieu Philippot

Date de remise : Lundi 21 Avril 2024

Table des matières

1. Lancement du code

Afin de pouvoir exécuter notre programme, l'environnement doit contenir matplotlib, numpy, random, typing, math, abc. Toutes ces librairies font partie de Anaconda ou de la librairie standard python.
La seule modification nécessaire à effectuer dans notre programme concerne le chemin d'accès vers le bench. Cette modification est encadrée par des commentaires dans le fichier **"Solution.py"**, qui exécute le main.

Notre code est fourni dans sa version finale, version qui exécute 10 runs (~5min) avant de les comparer et fournir les statistiques demandées. Vous pouvez cependant choisir le nombre de run en modifiant la variable nb_runs au même endroit que précédemment.

2. Choix algorithmiques

Notre première démarche pour résoudre le puzzle Eternity a été de sélectionner l'algorithme le plus approprié. Nous avons d'abord considéré la recherche locale pour son approche intuitive, mais nous avons vite basculé vers l'algorithme génétique en raison de sa capacité à explorer un espace de solution plus large, en s'inspirant de mécanismes de l'évolution naturelle. Nous maximisons donc nos chances de trouver une solution proche de l'optimal grâce à la diversité qu'apporte l'algorithme génétique.
Voici en détail les choix que nous avons effectués pour les principales composantes de l'algorithme génétique.

2.1 Crossover et tournoi

Le crossover représente l'ADN de l'algorithme génétique, son choix est donc crucial. Pour cette raison, nous avons multiplié les recherches et réflexions avant de converger vers notre solution actuelle :

Le choix du croisement multi-points a été motivé par sa capacité à favoriser la diversité génétique au sein de la population d'individus. En effet, en sélectionnant plusieurs points de coupure sur la grille, nous permettons aux parents d'échanger des sections de leur ADN de manière plus complexe, favorisant ainsi une exploration plus étendue de l'espace des solutions. De plus, nous avons fait le choix de conserver les bords et les coins afin de conserver la structure correctement établie lors du début de notre recherche (cf mutation). Enfin, pour assurer la cohérence de notre solution, nous éliminons les doublons créés lors du croisement, en les remplaçant par les pièces manquantes, c'est le rôle de la fonction "repair_child".

En conclusion, le choix du croisement multi-points avec conservation des bords et des coins des parents représente une approche équilibrée et efficace pour générer une diversité génétique tout en préservant les caractéristiques structurelles des solutions, ce qui en fait une option robuste pour notre algorithme génétique.

Pour choisir les parents lors du crossover, nous avons choisi une sélection par tournoi. Cette méthode permet d'introduire une part d'aléatoire tout en favorisant les solutions au score élevé.

2.2 Mutations - heuristiques

L'autre composante fondamentale de l'algorithme génétique est la mutation des solutions selon une probabilité "5% ?". Celles-ci permettent d'obtenir des solutions enfants qui se distinguent réellement des solutions parents, c'est donc une étape essentielle à la diversification/exploration des solutions. Afin d'influencer positivement cette recherche, nous avons mis en place diverses heuristiques, en fonction de notre avancée dans l'algorithme :

Pour les premières générations, nous avons décidé de ne modifier que les bords du puzzle car cela nous apporte rapidement une stabilité et un score intéressant.
Ensuite, nous utilisons des heuristiques intuitives et basiques afin d'améliorer nos solutions. Nous nous sommes basés sur notre intuition, ainsi que les articles mis à disposition sur MLS.

- Swap : On échange 2 pièces internes aléatoirement
- Rotation : On effectue une rotation sur 2 pièces internes aléatoires
- Swap&Rotate : On effectue la permutation de 2 pièces, puis leur rotation
- ThreeSwap : On permute et effectue la rotation de 3 pièces internes

On remarque cependant qu'arrivé à un score plus élevé, ces petits changements ont de moins en moins de chance d'améliorer le score. C'est un phénomène cohérent puisque les pièces commencent à matcher entre elles et effectuer un échange sur 2 pièces à plus de chance de casser une combinaison correcte qu'auparavant. Ainsi, nous avons décidé d'élargir notre heuristique à des régions entières du puzzle :

- RegionSwap : On effectue un échange entre 2 régions d'une taille sélectionnée de manière pseudo-aléatoire.
- RegionRotation : L'idée est la même avec une rotation sur un région entière

2.3 Recuit simulé

La méthode du recuit simulé est également une partie importante de notre programme car elle permet d'explorer de manière plus souple l'ensemble des solutions. Cette méthode va d'abord effectuer une copie de la solution trouvée avec le crossover et la dernière mutation. Puis, on va venir muter la copie en utilisant des heuristiques similaires mais propres au recuit simulé. La différence avec une simple mutation réside dans le fait que le recuit simulé accepte parfois des solutions de moindre qualité et permet donc à l'algorithme d'éviter de converger prématurément vers une solution bloquante et de faible qualité.

2.4 Autres fonctions

Nous avons ainsi expliqué la majorité des points techniques de notre algorithme génétique. Il ne reste plus qu'à tout exécuter grâce à la fonction "**genetic_algorithm**" qui assemble les fonctions entre elles. Voici le déroulé :

- Initialisation : On initialise le puzzle de manière aléatoire en respectant les contraintes sur les bords et coins.
- Evaluation : On évalue nos solutions quasiment à modification, et on stocke les solutions qui ont les meilleurs scores. Il est pertinent de stocker de stocker la meilleure solution puisque nous pourrions la perdre en acceptant une solution de moindre qualité avec le recuit simulée par exemple.
- Tournoi : On va sélectionner des parents parmi l'ensemble de notre population grâce à la méthode du tournoi. Cela favorise l'élitisme, tout en gardant une part d'aléatoire.
- Crossover : Création des enfants à partir de notre croisement multi points.
- Mutations : On explore de nouvelles solutions avec les mutations et le recuit simulé.
- Critère d'arrêt : L'algorithme s'arrête lorsque le critère est rencontré. Dans notre cas, il s'agit d'un nombre d'itération, mais nous aurions pu choisir un score à atteindre par exemple.

- Affichage : On affiche nos résultats et statistiques grâce à matplotlib puis, on affiche également le puzzle dans sa meilleure disposition.

3. Démarche expérimentale

Une fois notre premier algorithme implémenté et fonctionnel, nous avons décidé de partir en quête des 2 millions d'euros en optimisant notre algorithme !
Bien que nous soyons encore loin du score parfait, nos expériences et modifications sur les fonctions et les paramètres nous ont permis de doubler notre score petit à petit.

3.1 Amélioration des différentes "boîtes"

La première grande étape a été l'amélioration concrète de notre algorithme. Nous avons ainsi tout au long du projet testé de nouvelles idées pour les différentes "boîtes" de notre algorithme.

En ce qui concerne le croisement, nous avons commencé avec un crossover uniforme. Plus simple à intégrer, il était toutefois moins efficace, en termes de qualité comme de vitesse d'exécution. Nous avons convergé vers notre solution actuelle, un crossover multi-points et qui conserve les bords/coins, car nous octroyons en temps de calcul suffisamment grand pour que ceux-ci soient corrects.

Ensuite, ce sont les mutations qui ont le plus retenues notre attention. Le choix des heuristiques est primordial car il permet de guider l'algorithme vers un meilleur score. Nous nous sommes d'abord inspiré de notre manière, en tant qu'humain, de résoudre le puzzle, avec premières heuristiques naïves (swap, rotation) voir totalement inefficace (Swap 2 pièces de même couleur, qui pourrait juste être inversée par exemple). Ainsi, notre algorithme s'améliorait mais convergeait très rapidement vers une solution faible (~180). Nous avons fait tourné des parties avec l'interface graphique afin de comprendre le problème et, avons conclu qu'il fallait en fait utiliser la puissance fournie par l'ordinateur et une approche moins humaine. En effet, lorsque le puzzle commence à avoir un score élevé, la probabilité qu'un swap

fasse perdre du score devient trop haute pour considérer cette option, nous avons donc décidé de mouvoir des régions entières (de taille pseudo aléatoire) et ainsi réduire notre exposition aux minima locaux.

Enfin, nous avons simplement effectué beaucoup de simulation en variant nos méthodes. Nous avons ainsi privilégié certaines heuristiques au profit d'autres, que ce soit pour la mutation où le recuit simulé. Pour avoir un score significatif, nous lançons toujours sur 10 runs avec 100k évaluations.

3.2 Variations des paramètres

La dernière étape a été l'ajustement des paramètres afin d'optimiser l'algorithme.

Nous avons d'abord modifié le couple (taille population, nombre de générations) pour contrôler la diversité et la pression sélective.

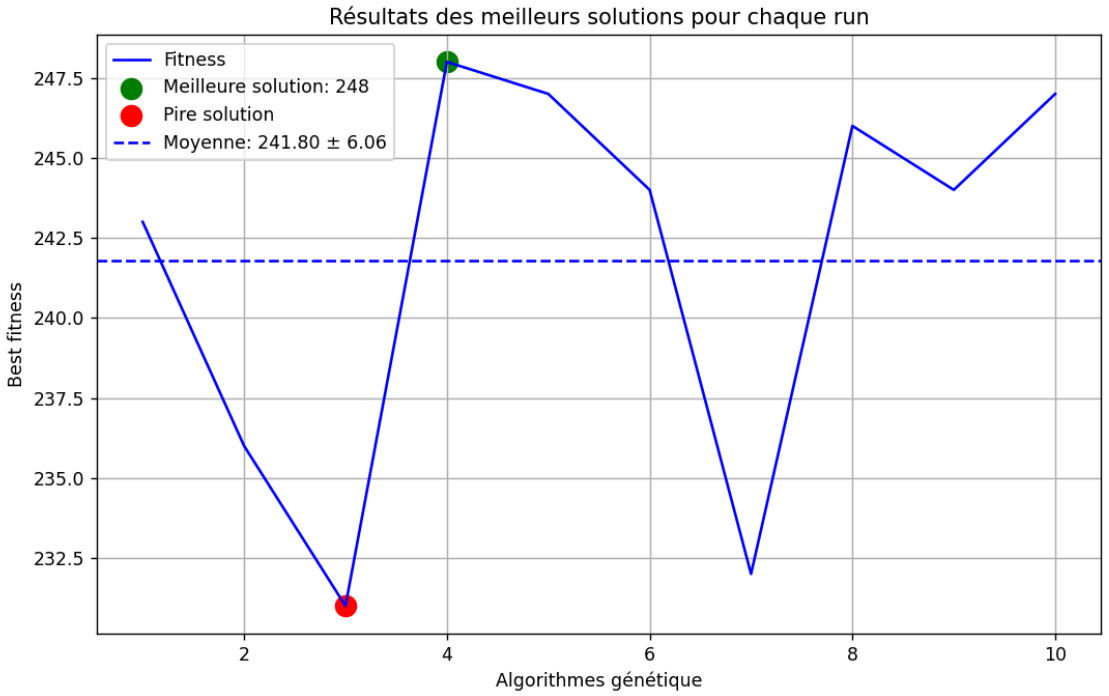
Ensuite, le réglage du taux de refroidissement et de la température initiale dans le recuit simulé a permis d'éviter les optimaux locaux. Nous avons également adapté le taux de mutation pour maintenir la diversification de la population.

Enfin, le choix de la taille du tournoi dans la sélection des parents a été ajusté pour influencer la pression sélective. Nous voulions à la fois donner une chance importante aux meilleures solutions d'être sélectionnées, tout en gardant une part d'aléatoire. En effet, l'aléatoire dans ces processus a permis une exploration efficace de l'espace des solutions, conduisant à une amélioration notable des performances globales de l'algorithme.

4. Résultats

4.1 Statistiques

En limitant notre algorithme aux 100 000 premières évaluations nous obtenons un score moyen de 242 avec un écart type de 6. Voici le graphique qui résume nos 10 runs :



Notre algorithme converge en général vers un minimum local, la différence n'est donc pas très grande lorsque nous n'imposons aucune limite. En effet, en augmentant la taille de la population et le nombre de générations, nous avons une moyenne autour de 255, avec 280 comme meilleur score obtenu.

4.2 Pistes d'amélioration

La principale difficulté rencontrée par notre algorithme est les minima locaux. Voici donc nos pistes d'amélioration que nous n'avons soit pas eu le temps, soit par réussi à implémenter.

- Nouvelles heuristiques : Il serait intéressant de trouver une heuristique qui permettent de mieux “choisir” les pièces à modifier en analysant directement le plateau. Nous avions aussi pensé à utiliser une heuristique d’optimalité partielle, qui permettrait de résoudre un sous-puzzle de petite taille.
- Avoir plusieurs phases : Cette approche serait intéressante pour éviter les extrema locaux car elle laisse la possibilité de faire tourner l’algorithme avec un ensemble de méthodes/paramètres puis, de reprendre une solution correcte mais bornée avec un autre ensemble de méthodes et paramètres qui lui, ne sera pas forcément à un minimum local pour cette solution donnée.
- Enfin, il est toujours possible d’optimiser davantage le choix des paramètres, en effectuant plus de run, et en corrélant les paramètres entre eux