Projet

Baptiste Prévost - 2231785
Guilhem Garnier - 2230323
April 24, 2023

Phase 1 : Solveur heuristique

Dans toutes les phases, nous avons arbitrairement fixé un coin en position 0 pour supprimer une symétrie de rotation. Dans notre heuristique de construction, nous considérons les positions restantes dans un ordre défini, et cherchons parmi les pièces restantes celle qui convient.

- Les *gloutons*, qui essaient à chaque étape de minimiser le nombre de nouveaux conflits générés
- Les *récursifs*, qui, dans le temps imparti, cherchent à compléter parfaitement le plateau, en revenant sur ses pas si aucune des pièces candidates ne conviennent.

Plusieurs ordres peuvent donc être considérés pour cette phase de construction. Voici ceux que nous avons considérés. De gauche à droite, on peut les nommer ”balayage”, ”double balayage”, ”spirale”, ”spirale inversée”, ”double spirale”.

13	14	15	16
9	10	11	12
5	6	7	8
1	2	3	4

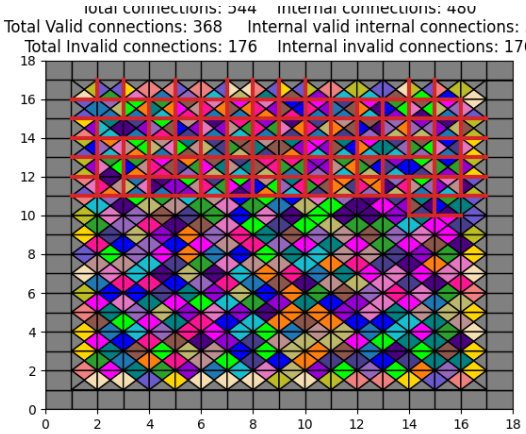
10	12	14	16
2	4	6	8
9	11	13	15
1	3	5	7

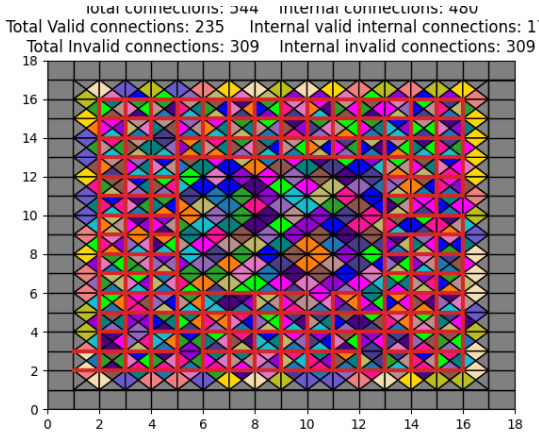
10	9	8	7
11	16	15	6
12	13	14	5
1	2	3	4

7	8	9	10
6	1	2	11
5	4	3	12
16	15	14	13

14	16	15	13
12	2	4	11
10	8	6	9
1	3	5	7

L’odre qui donne les meilleurs résultats semble être celui par balayage. Il permet d’obtenir en quelques secondes une grille avec environ $\frac{2}{3}$ du puzzle complet sans conflits, et le reste chaotique, ce qui semble être un bon point de départ pour nos algorithmes de recherche.
Cette méthode est suffisante pour obtenir l’optimum sur les instances A et B. Voici en exemple le résultat de l’heuristique sur l’instance complète, et une comparaison avec une construction en double spirale.





Phase 2 : Solveur de recherche locale

Notre fonction de recherche locale est un algorithme de *hill climbing* avec des redémarrages aléatoires. Nous utilisons une heuristique de construction récursive décrite plus haut, en mélangeant les pièces pour garantir plus de diversité lors des redémarrages.

Voisinages

Nous utilisons un voisinage proposé dans plusieurs papiers ([1], [2]) , qui consiste à sélectionner k pièces non-adjacentes, et trouver une permutation de ces pièces (en considérant leurs rotations) qui minimise le nombre de conflits. Du fait que ces pièces ne partagent aucun bord, on peut formuler la construction d'un voisin comme un problème d'assignation dans un graphe biparti valué. La bibliothèque *scipy.optimize* fournit avec la fonction *linear_sum_assignment* en $\mathcal{O}(k^3)$.
Le nombre de pièces du bord étant plus faible que celui de pièces intérieurs, on doit encourager la recherche dans ces deux régions. On distingue donc deux régions dans ce voisinage, en considérant des permutations de pièces du bords du puzzle, ou de pièces de l'intérieur. Le choix du voisinage est aléatoire.
En se basant sur les remarques de "Hybridization of CP and VLNS for Eternity II."([2]), on construit des voisins en cherchant à replacer $k = 16$ pièces non adjacentes pour l'intérieur, et $k = 24$ pour les pièces du bord.

Validité et sélection

Ce voisinage augmente fortement avec la valeur de k (sa taille est majorée par $\binom{n}{k}k!$, avec n la dimension du plateau). On génère alors à chaque itération un voisin aléatoire, en sélectionnant en priorité les pièces avec le plus de conflits.

Évaluation et acceptation

Avec ce voisinage, on peut facilement calculer le nombre de conflits du nouveau puzzle. Nous avons opté pour un critère simple, et un voisin est accepté si il présente un nombre de conflits inférieur ou égal à la solution actuelle.

Phase 3 : Solveur avancé

Une fois le voisinage clairement défini, l'algorithme de recherche locale est relativement simple. Nous avons donc implémenter plusieurs mécanismes pour améliorer nos résultats.

Diversification

Nous avons d'abord observé que notre recherche locale induisait trop d'intensification, en cherchant des voisins améliorants forts. Nous avons donc ajouté 3 autres voisinages simples, consistant à sélectionner au hasard 2 coins, bords ou pièces intérieures, et à les mélanger aléatoirement.
Il est intéressant d'observer que ces nouvelles fonctions permettent un voisinage mieux connecté, notamment au début de la recherche, et donc une recherche plus exhaustive.

Intensification

Malgré un espace de recherche très grand, et un voisinage considéré très large, la recherche arrive très rapidement dans des mimima locaux. Nous avons observé que le placement initial des coins pouvait fortement influer la qualité de la recherche, malgré une fonction de voisinage dédiées, surtout sur les

petites instances.

Nous avons donc formulé un problème de bandit manchot pour la sélection de la disposition des trois autres coins. On attribue à chaque disposition un certain score. Avant la phase de construction, on choisit une disposition des coins fixée : on prend avec une probabilité ϵ celle qui maximise notre score, et $1 - \epsilon$ une disposition au hasard. Après la fin de notre recherche, si la solution est la meilleure trouvée jusqu'ici, on incrémente le score de la disposition sélectionnée de 1.

Ci-dessous un tableau présentant un comparatif des résultats pour différentes valeurs d'ϵ sur l'instance complète, pour 25 recherches par valeur, avec notre algorithme de recherche locale.

ϵ	min	max	mean
0	55	68	60.3
0.2	47	75	59
0.4	52	64	55.3
0.6	48	71	59
0.8	52	66	58
1	55	66	59.4

Nous avons choisi de garder la valeur $\epsilon = 0.4$ parce qu'elle présentait la meilleure moyenne sur 25 recherches.

Heuristiques

Dans "A Guide-and-Observe Hyper-Heuristic Approach to the Eternity II Puzzle" [3], les auteurs proposent de débuter la recherche avec d'autres heuristiques d'évaluation lors du *hill climbing*, tout en conservant la meilleure solution (en terme de conflits) rencontrée pendant la recherche, et d'enfin effectuer une recherche locale comme décrite plus haut.

Les heuristiques considérées évaluent le nombre de carrés sans conflits de dimension c. Ces heuristiques encouragent la recherche à atteindre des solutions avec des régions complètes sans conflits. On peut observer qu'une solution optimale pour une de ces heuristiques est une solution optimale pour le problème initial.

Les auteurs proposaient de suivre pendant $\frac{3}{4}$ du temps cet objectif alternatif, puis poursuivre par une recherche locale à partir de la meilleure solution rencontrée. Nous avons pensé que ces différentes heuristiques pouvaient être utiles à différents instants de la recherche : plus la solution s'améliore, plus elle est capable de suivre une heuristique qui cherchent à construire de grands carrés. Nous avons donc proposé l'algorithme suivant :

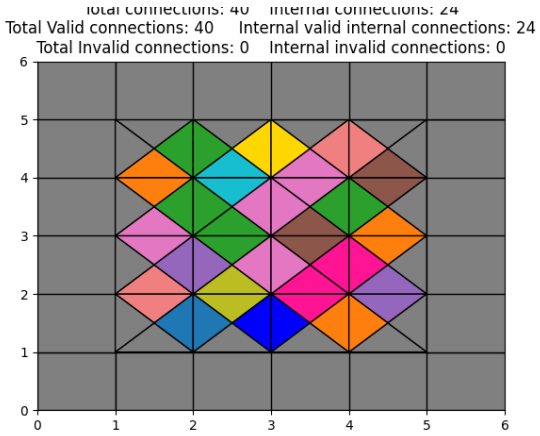
Algorithm 1 surrogateSearch

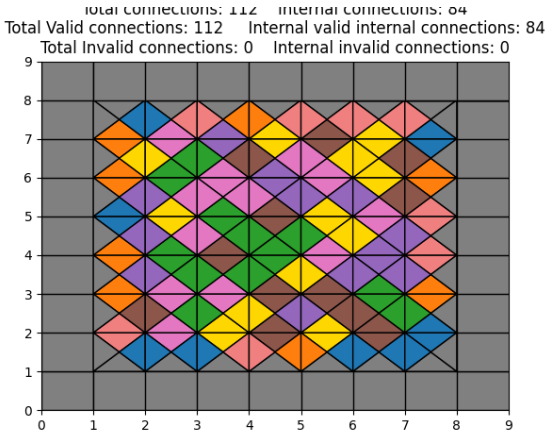
Require: $board$, $maxT$, c
  $totalConflicts \leftarrow$ conflicts($board$)
  $bestBoard \leftarrow board$
  $bestConflicts \leftarrow totalConflicts$
  $totalPerfectSquare \leftarrow$ perfectSquares($board, c$)
  for $t \in [1, maxT]$ **do**
    $neighbor \leftarrow$ randomNeighbor($board$)
    $neighborSquares \leftarrow$ perfectSquares($neighbor, c$)
    if $neighborSquares \leq totalPerfectSquare$ **then**
      $totalPerfectSquare \leftarrow neighborSquares$
      $board \leftarrow neighbor$
      $totalConflicts \leftarrow$ conflicts($board$)
      if $totalConflicts < bestConflits$ **then**
        $bestBoard \leftarrow board$
      end if
    end if
  end for
  return $bestBoard$

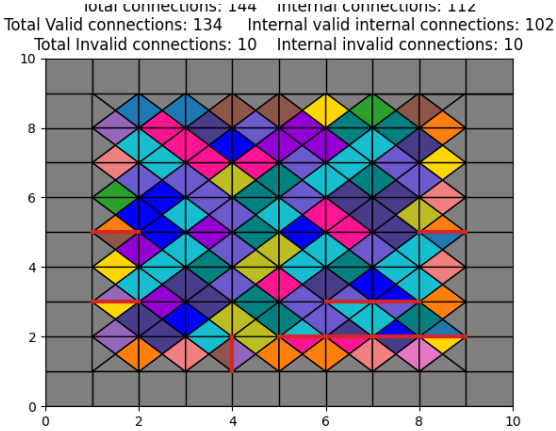
Algorithm 2 largeSearch

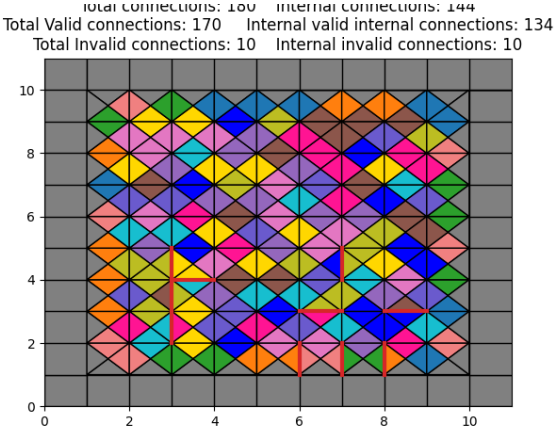
Require: $board$, $maxT$, c
  $squareSide \leftarrow$ side($board$)$/5$
  $bestBoard \leftarrow board$
  for $t \in [1, 10]$ **do**
    $bestSurrogateBoard \leftarrow$ surrogateSearch($board, squareSide$)
    if conflicts($bestSurrogateBoard$) $<$ conflicts($bestBoard$) **then**
      $bestBaord \leftarrow bestSurrogateBoard$
      $squareSide \leftarrow squareSide + 1$
    else
      $squareSide \leftarrow squareSide - 1$
      if $squareSide == 0$ **then**
        $squareSide \leftarrow$ side($board$)$/5$
      end if
    end if
  end for
  return $bestBoard$

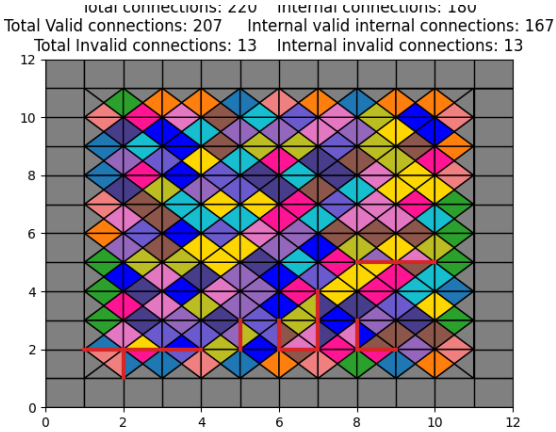
Phase 4 : Résultats

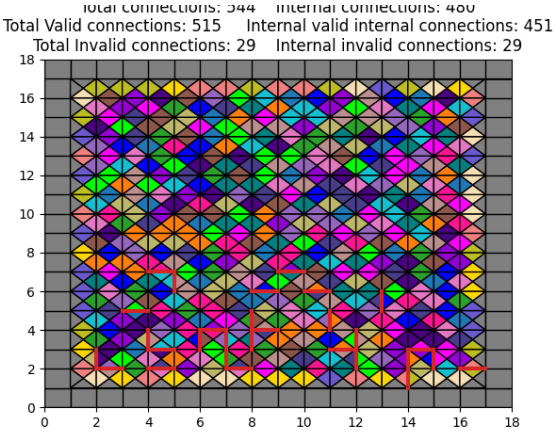












Pistes non retenues

Nous présentons ici plusieurs idées explorées lors de la conception de notre algorithe mais non retenues.

Pathrelinking

Pour le devoir 2 notamment, où nous avions utilisé un algorithme GRASP, le pathrelinking avait permis d'obtenir de très bonnes nouvelles solutions. L'idée du pathrelinking est de considérer deux solutions de très bonne qualité, et de chercher un chemin d'opérations simples qui les relie, en espérant trouver sur

la route une meilleure solution.

- La première étape est de considérer les positions des pièces qui ne sont pas identiques dans les deux solutions.
- On choisit ensuite une solution de début, et une solution de fin. Les résultats théoriques montrent qu'il est efficace d'interchanger ces rôles à chaque itération.
- Une itération consiste à forcer dans la solution de départ un élément de la solution de fin qui n'y figure pas encore.

Nous avons essayé deux méthodes de pathrelinking différentes.
Pour la première, on choisit une pièce de la solution de fin, et on repère sa position dans la solution initiale. Lorsqu'on la place dans la solution de début identiquement à la solution de fin, la pièce qui est ejectée est alors replacée à la position repérée, en minimisant les conflits créés.
Une autre idée était de créer des cycles. Au lieu de placer la pièce libérée à l'emplacement libéré, on la replace comme elle figure dans la solution de fin, ce qui libère une autre pièce, et on continue jusqu'à former un cycle de permutations.
Ces deux méthodes ont été longues à mettre en place, et ne fournissait aucun bon résultat. Les solutions semblent bien trop différentes pour être efficacement mélangées pendant un pathrelinking.

Perturbation

En voulant s'inspirer des algortihmes ILS ou ALNS, nous avons essayé des recherches avec des perturbations de plusieurs types :

- Mélanger au hasard k pièces du puzzle
- Mélanger au hasard une portion carrée du puzzle
- Mélanger au hasard une portion losange du puzzle
- Sélectionner le pire voisin possible dans le voisinage de notre méthode de recherche locale.

Ces méthodes semblaient longues à configurer, et ne nous ont pas donné de résultats suffisament bons pour continuer à les explorer.

Tabu

Un papier ([4]) proposait d'utiliser une recherche locale itérée avec tabou. Cependant, les voisinages utilisés étaient très relativement petits, avec seulement deux pièces échangées, ce qui permettait de concevoir facilement des listes tabou efficaces. Après quelques essais, nous avons conclus que notre espace de recherhe était bien trop grand pour justifier l'utilisation d'une liste tabou.

Conclusion

En nous appuyant sur plusieurs papiers de recherche, une bonne impltémentation et beaucoup d'exploration, nous avons pu obtenir de très bons résultats, notamment pour l'instance complète, en se référant à l'état de l'art et les solutions des années précédentes. Il peut être intéressant de noter que nos meilleurs résultats ont été souvent obtenus en relativement peu de temps. Par exemple, nous avons trouvé notre meilleur résultat sur l'instance complète deux fois, une fois pendant nos tests, en faisant tourner l'algorithme

pendant moins de 3 minutes 30, une autre fois à l'issu de 45 minutes de recherche. On peut donc imaginer qu'il reste une certaine marge de progression dans la calibration de l'algorithme, ou la diversification intelligente, pour conserver la richesse et la connectivité de nos voisinages tout en évitant les minima locaux.

References

[1] MILP and Max-Clique based heuristics for the Eternity II puzzle
Fabio Salassa, Wim Vancroonenburg, Tony Wauters, Federico Della Croce, Greet Vanden Berghe

[2] Hybridization of CP and VLNS for Eternity II
Pierre Schaus, Yves Deville

[3] A Guide-and-Observe Hyper-Heuristic Approach to the Eternity II Puzzle
Tony Wauters, Wim Vancroonenburg, Greet Vanden Berghe

[4] Solving Eternity-II puzzles with a tabu search algorithm
Wei-Sin Wang and Tsung-Che Chiang