Projet Honshu - Lot C

Thomas KOWALSKI, Thibaut MILHAUD,
Matthis PILLONEL, Yoan ROCK

29 avril 2018

Table des matières

1 Introduction

Après avoir terminé le lot B, nous avons maintenant un jeu tout à fait fonctionnel, permettant :

— De charger une partie ;

— De joueur la partie ;

— D'en obtenir le score au fur et à mesure que le joueur place ses tuiles.

La partie suivante concerne la création d'un solveur. Nous en proposerons au moins deux.

2 Algorithme de résolution

2.1 Difficulté de la résolution

Soit k le nombre de tuiles à poser dans une partie et n la largeur de la grille de jeu.

Notons alors $\mathcal{I}_{n,k}$ le nombre d'issues possibles pour une situations initiale donnée et essayons de le majorer. il faut premièrement choisir l'ordre des tuiles ce qui fait $k!$ possibilités, ensuite à chaque fois que l'on pose une tuile il faut choisir sa position et son orientation nombre que l'on peut majorer grossièrement par $4n^2$.

Ainsi il apparait que $\mathcal{I}_{n,k} = O(k! \times n^{2k} \times 4^k)$, dès lors envisager de trouver la meilleure solution en les explorant toutes semble délirant. En effet pour avoir un ordre d'idée, avec $k = 12$ et $n = 9$, on obtient $k! \times n^{2k} \times 4^k = 6.4 \times 10^{38}$.

Malheureusement, étant donné que chaque tuile posée précédement influe sur le potentiel des suivantes, et que le score n'est pas forcement croissant, trouver la meilleure solution d'une autre manière semble impossible : il va falloir trouver des méthodes de résultion approchée.

2.2 Algorithme Glouton

Ce qui coute cher dans l'exploration totale de l'arbre des possibilités c'est d'avoir un k en exposant de $(2n)^2$ cela povient de la méthode par éssais successifs qu'il faut employer.

Pour enlever ce facteur de complexité, on va ici choisir de réfléchir sur un seul coup, c'est-à-dire sans coup d'avance.

On va chercher à chaque étape quel est le score maximal que l'on peut atteindre avec une tuile. Pour ce faire, on va donc parcourir les tuiles, les positions et les orientations disponibles pour trouver le meilleur triplet (t, p, o) et ensuite placer la tuile t en p orientée vers o.

Ansi, si l'on est à l'étape i de l'algorithme, on va réaliser au maximum $\mathcal{C}_i = 4(k-i)n^2$ tests. Et comme chaque choix est définitif il y aura précisement k étapes et donc on notant $\mathcal{C}$ un majorant de la complexité totale, on obtient :

$$\mathcal{C} = \sum_{i=0}^{k-1} \mathcal{C}_i = 4n^2 \sum_{i=0}^{k-1} (k-i) = 2k(k+1)n^2 = O(k^2 n^2)$$

Ce qui nous donne une complexité majorée par un polynome, autrement dit, un algorithme viable.

Pseudo-code

```
Tant qu'il reste des tuiles
    MeilleurScore = 0
    MeilleurChoix = (NULL, -1, -1, -1)
    Pour chaque Tuile
        Pour i de 1 à n
            Pour j de 1 à n
                Pour chaque rotation rot
                    Placer Tuile en (i, j)
                    Score = CalculerScore()
                    Si Score > MeilleurScore
                        MeilleurScore = Score
                        MeilleurChoix = (Tuile, i, j, rot)
                    Fin Si
                    Retirer la dernière tuile
                Fin Pour
            Fin Pour
        Fin Pour
    Fin Pour
    effectuer MeilleurChoix
    retirer MeilleurChoix[0] de la liste des tuiles
Fin Tant que
```

Cet algorithme est bien un algorithme glouton, puisqu'il cherche à maximiser le score total en maximisant le score à chaque pose de tuile.

Nous en avons réalisé deux implémentations légérement différentes. La première dans `librairies/O\_de\_un.c` nommée `meilleur\_score()` qui execute précisement l'algorithme décris ci-dessus lorsque le paramètre `limit` vaut 12.

Et la seconde dans `librairies/resolution.c` qui ne se préoccupe pas de choisir la bonne tuile et les place dans l'ordre initial ce qui fait passer la complexité de $O(k^2n^2)$ à $O(kn^2)$.

Cependant, en appliquant l'algorithme qui réordonne les tuile on obtient pour la partie de test un score de 58 contre 33 en conservant l'ordre initial.

2.3 Algorithme plus intelligent

Nous avons rapidement remarqué qu'une difficulté à prendre en compte pour le solveur était le fait que le score n'est pas croissant. De plus même en choisissant le meilleur coup lors de la pose d'une tuile, cela pouvait nous pénaliser pour les coups suivants.

Pour pallier ce problème, nous avons décidé d'écrire un solveur dont la politique serait l'expansion : nous sommes partis du principe que même si les façons de marquer des points sont variées, la plupart des cases rapportent des points, il faut donc en recouvrir le moins possible.

Cet algorithme fonctionne de la même façon que l'algorithme précédent, sauf que pour chaque tuile à poser, il tente d'abord de trouver une position qui maximise le score tout en ne recouvrant qu'une case du plateau.

Si ce n'est pas possible de ne recouvrir qu'une case, il applique le même algorithme que la méthode gloutonne, à savoir qu'il maximise le score sur l'ensemble de la grille, sans se soucier du nombre de cases recouvertes.

La complexité de ce nouvel algorithme est du même ordre que celui d'avant. Ce qui donne, si on place les tuiles dans l'ordre initial : $O(2 \times 4 \times k \times n^2) = O(k \times n^2)$.

En appliquant cette méthode au lieu de la même sans essayer d'optimiser l'espace occupé, le score donné par notre solveur passait de 33 à 47 ce qui est presque une multiplication par $1,5$.

Nous en avons conclu que cette méthode était très efficace par rapport à la première méthode imaginée.

2.4 Algorithme mixte

Enfin, la dernière idée que nous avons eu a été d'appliquer une méthode de résolution exacte à un problème plus simple : le même où les j premières tuiles ont déjà été posées.

Pour ce faire, nous avons choisi de placer les j premières tuiles grâce au premier algorithme glouton puis de placer récursivement les autres en explorant la fin de l'arbre. Ainsi on obtient une complexité en :

$$\mathcal{C} = O(j^2 n^2 + (k-j)! n^{2(k-j)} 4^{k-j})$$

On voit bien que cet algorithme ne va nous donner des résultats en un temps raisonnable que pour des petites valeurs de $(k-j)$ et dans ce cas, on trouve des bien meilleurs score qu'avec les précédents algorithmes.

Toutefois, dans l'optique d'un solveur pour un jeu, cet algorithme n'est pas envisageable dans la mesure où le temps d'exécution se compte en minutes même pour $(k-j) = 3$ ce qui est rédhibitoire.

3 Difficultés rencontrées

Il existe de nombreux algorithmes d'optimisation pour les problèmes combinatoires, choisir celui que l'on voulait mettre en application comme *algorithme plus intelligent* n'a pas été facile.

Cependant, on a fait beaucoup de progrès sur le travail en équipe, ainsi que sur l'utilisation de Git.

Malheureusement, suite à un dysfonctionnement d'Arise, on a dû déménager le dépôt vers GitHub. Comme tout le monde n'a pas suivi l'affaire, il y a eu des commits dans la semaine sur l'ancien dépôt, ce qui a causé des besoins de fusionner des fichiers de manière pas toujours pratique.

A Annexe

A.1 Code

Le code source du projet est comme d'habitude dans le dossier `librairies/` et les solveurs se trouvent les fichiers : `librairies/0_de_un.c` (glouton avec ordonnancement et solveur mixte) et `librairies/resolution.c` (les autres).

A.2 Makefile

Pour jouer au jeu et avoir la solution proposée par un solveur comparée à la sienne en fin de partie, il faut faire `make honshu_B` puis `./honshu_B`.

Pour comparer les solveurs il faut choisir la partie à charger dans `librairies/compare.c` puis `make compare` et enfin `./compare`.

A.3 README

Pour compiler et exécuter le lot C :

```
make honshu_C && ./honshu_C
```

Pour vérifier les fuites de mémoire :

```
make vg_c
```