Projet informatique: Honshu (Lot C)

Romain PEREIRA
Douha OURIMI
Afizullah RAHMANY
Guangyue CHEN

29/04/2018

Sommaire

Préambule

Ce projet est réalisé dans le cadre de nos études à l'ENSIIE. L'objectif est de prendre en main des outils de 'programmation agile', en developpant un jeu de carte : le Honshu.

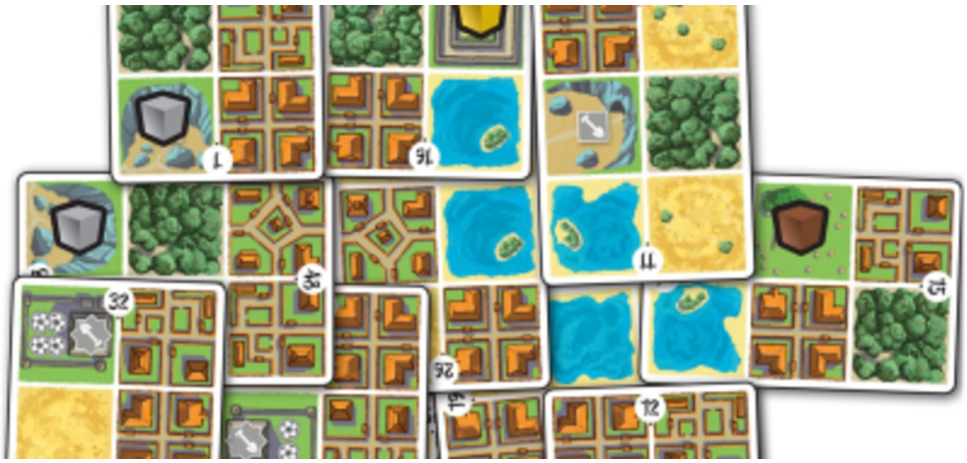

Figure 1: *Plateau de jeu*

1 Introduction

Pour ce lot, il nous est demandé d'effectuer le calcul des points, et de créer des solveurs (algorithmes proposant une solution de résolution de la grille). Le travail a été séparé de la manière suivante:

Douha : calcul du score

Afizullah : création de tests unitaires

Romain et *Guangyue* : création des solveurs (5)

2 Calcul du score

La fonction de score a été programmé en respectant les règles du sujets. (voir 'grille.c')

3 Tests unitaires

Des tests unitaires ont été implémenté afin de s'assurer que la fonction du calcul de score fonctionne. En effet, cette fonction est primordiale pour l'implémentation d'un solveur, c'est pourquoi du temps a été consacré à l'élaboration de tests unitaires.

4 Interface 'solveur.h'

Afin d'avoir une modularité dans notre travail, nous avons normé nos fonctions de solveurs. Un solveur prends une grille (tuiles insérés + tuiles en main), et renvoie la liste des insertions à effectuer pour obtenir le score trouvé.

```
typedef struct  s_insertion {
        /** index de la tuile dans 'grille>tuiles' */
        unsigned int tuileID;
        /** rotation de la tuile */
        BYTE rotationID;
        /** position ou la tuile a ete insere */
        INDEX x, y;
        /** score apr s cette insertion */
        unsigned int score;
}               t_insertion;

/** structure qui renvoie le resultat du solveur */
typedef struct  s_resultat {
        /** l'ordre dans lequel les tuiles doivent  tre  inseres */
        t_insertion     * insertion;
        /** taille du tableau 'insertion' */
        unsigned int    nb_tuiles;
}               t_resultat;

/** typedef d'une fonction de resolution */
typedef t_resultat * (* t_solveur) (t_grille * grille) ;
```

5 Solveur

5.1 Mise en contexte

Afin de mieux envisager les algorithmes suivants, on se propose de travailler dans le cadre suivant.

Définition On note $G = (M, I)$ une grille, avec:

- M l'ensemble des tuiles en main, $M = \{T_1, ..., T_m\}$, $m = Card(M)$
- I l'ensemble des tuiles insérées sur la grille, $I = \{(T_p, R_p, x_p, y_p, i_p)/p \in \{1, ..., k\}\}$, avec
  - . $k = Card(I)$
  - . T_p une tuile
  - . R_p la rotation de la tuile
  - . $x_p \in \{1, ..., n\}$ la position x d'insertion de la tuile dans la grille.
  - . $y_p \in \{1, ..., n\}$ la position y d'insertion de la tuile dans la grille.
  - . $i_p \in \{1, ..., k\}$ l'indice d'insertion de la tuile sur la grille. (l'ordre dans lequelle elle a été inséré, incrémenteur)

On suppose que les n-uplets de I respectent les règles d'insertion.

Définition Une grille $G = (M, I)$ est **finale** $\Leftrightarrow Card(M) = 0$. ($\Leftrightarrow$ 'toutes les tuiles sont insérées')

Définition Soit $G_1 = (M, I)$ et $G_2 = (N, J)$ 2 grilles. On dit que G_1 engendre G_2 si:

- $N \subset M$
- $I \subset J$
- $\forall (T, R, x, y, i) \in J \backslash I, \quad T \in M \quad et \quad T \notin N$

Autrement dit, G_1 engendre G_2 si l'on peut obtenir G_2 en insérant des tuiles de la main de G_1.

5.2 Approche par force brute (parcours en profondeur)

Dans un 1er temps, on se propose d'implémenter un algorithme qui renvoie la solution optimale, en effectuant un parcours en profondeur sur toutes les combinaisons possibles de grille finale engendrable par une grille donnée.

Le problème est le suivant: on suppose que l'on possède une grille de coté n, avec m tuiles en main, et k tuiles déjà insérées sur la grille. On souhaite tester le score de toutes les configurations possibles de la grille, après insertion des m tuiles en main sur la grille.

5.2.1 Etude et expérimentation

Experimentalement, (car cela dépends des cases), on remarque la propriété suivante:

Propriété faible d'insertion Soit $G = (\{T\}, I)$ une grille (avec une seule tuile T en main). G peut engendrer $\boxed{c * Card(I)}$ grilles finales 2 à 2 distinctes, avec $c \in [10; 14]$ une constante d'insertion.
N.B: (voir './complexity.csv'). c semble converger vers 12 en faisant la moyenne empirique sur des grands échantillons de parties aléatoires, mais on se limite à un énoncé faible car la valeur concrete de la constante n'a d'intêret que pour l'application numérique finale.

But Montrons par récurrence que l'hypothèse $H(m)$ suivante est vrai pour tout $m \in \mathbb{N}^*$.

$H(m)$: pour toute grille $G = (M, I)$, tel que $Card(M) = m \quad et \quad Card(I) = k$, G peut engendrer (en moyenne) jusqu'à $\phi(k, m)$ grilles finales 2 à 2 distinctes avec:

$$\boxed{\phi(k,m) = c^m * \frac{(k+m-1)!}{(k-1)!} * m!}$$

Initialisation Pour $m = 1$, d'après la propriété faible d'insertion, on a $\boxed{c * k}$ façon possibles d'insérer la tuile, et:

$$\begin{aligned} \phi(k,m) &= c^m * \frac{(k+m-1)!}{(k-1)!} * m! \\ &= c^1 * \frac{(k+1-1)!}{(k-1)!} * 1! \\ &= c * \frac{k!}{(k-1)!} * 1 \\ &= c * k \end{aligned} \tag{1}$$

Donc $H(G)$ est vérifié pour toute grille $G = (M, I)$ tel que $Card(M) = m = 1$.

Récurrence On suppose $H(G)$ vraie pour toute grille $G = (M, I)$ tel que $Card(M) \leq m$.
Montrons que $H(m + 1)$ est vraie.
Soit $G = (M, I)$ une grille tel que

$$Card(M) = m + 1 \quad et \quad Card(I) = k \ quelconque$$

D'après la propriété faible d'insertion, G peut engendrer jusqu'à $c * k * (m + 1)$ grille $G' = (I', M')$ tel que $Card(I') = k+1$ et $Card(M') = m$. Par hypothèse de récurrence, chaque G' peut engendrer jusqu'à $\phi(k + 1, m)$ grilles. Le nombre de grilles finales engendrables par G est donc de:

$$\begin{aligned} c * k * (m+1) * \phi(k,m) &= c * k * (m+1) * c^m * \frac{(k+m-1)!}{(k-1)!} * m! \\ &= c^{m+1} * k * \frac{(k+m-1)!}{(k-1)!} * (m+1)! \\ &= c^{m+1} * \frac{(k+m)!}{(k-1)!} * (m+1)! \\ &= \boxed{\phi(k, m+1)} \end{aligned} \tag{2}$$

La récurrence est vérifiée.

Conclusion Une grille $G = (M, I)$ peut engendrer en moyenne jusqu'à $\phi(k, m)$ grilles finales.

Complexité du parcours en profondeur Le calcul du score est un algorithme en $O(n^2)$. Trouver les insertions a effectué sur une grille G, afin d'obtenir un score optimal à l'aide d'un parcours en profondeur, est donc effectué avec une complexité

$$\boxed{C(k, m, n) = \phi(k, m) * O(n^2)}$$

Application numérique On se place dans le cadre du sujet: on possède une grille

$$G = (M, I) \; tel \; que \quad Card(M) = m \quad et \quad Card(I) = k = 1$$

$$\begin{aligned} \phi(k,m) &= c^m * \frac{(k+m-1)!}{(k-1)!} * m! \\ &= c^m * \frac{(1+m-1)!}{(1-1)!} * m! \\ &= c^m * (m!)^2 \end{aligned} \tag{3}$$

On choisit $c = 12$.
En fonction de m, on obtient donc un nombre de grille final engendrable de:

- $m = 1 \Rightarrow \phi(k,m) = 12$
- $m = 2 \Rightarrow \phi(k,m) = 576$
- $m = 3 \Rightarrow \phi(k,m) = 62\ 208$
- $m = 4 \Rightarrow \phi(k,m) = 11\ 943\ 936$

Soit pour une grille de coté $n = 32$, un nombre d'opération élémentaire de l'ordre de:

- $m = 1 \Rightarrow C(k,m,n) = 12288$
- $m = 2 \Rightarrow C(k,m,n) = 589\ 824$
- $m = 3 \Rightarrow C(k,m,n) = 63\ 700\ 992$
- $m = 4 \Rightarrow C(k,m,n) = 12\ 230\ 590\ 464$

On remarque que très rapidement, lorsque m augmente, cette approche (par force brute $\Leftrightarrow$ parcours en profondeur) est beaucoup trop lourde temporellement. Cependant, elle nous permettra d'assurer l'optimalité du résultat pour des petites grilles, afin de comparer ses résultats avec les algorithmes futurs.

Remarque Dans l'implémentation final, l'algorithme de score a été optimisé: on ne parcourt plus les n^2 cases de la grille, mais on garde en mémoire la position la plus 'en haut à gauche', et la plus 'en bas à droite' d'une case non vide, et on ne vérifie que le rectangle formé par ces 2 points.

5.2.2 Algorithme

Algorithm 1: Renvoie le score optimal atteignable pour une grille donnée

```
1: function PARCOURS_EN_PROFONDEUR(G = (M, I))
2:     function PARCOURIR(G = (M, I), best)
3:         for T ∈ M do
4:             for R ∈ {NORD, SUD, EST, OUEST} do
5:                 for x ∈ {1, ..., n} do
6:                     for y ∈ {1, ..., n} do
7:                         if (T, R, x, y) peut être inséré dans I then
8:                             (Insérer) : supprimer T de M et ajouter (T, R, x, y) à I
9:                             (Vérifier optimalité)
10:                            if Card(M) = 0 then
11:                                best := max(best, score(G))
12:                            else
13:                                best := parcourir(G, best)
14:                            end if
15:                            (Dé-insérer) : ajouter T à M et supprimer (T, R, x, y) de I
16:                        end if
17:                    end for
18:                end for
19:            end for
20:        end for
21:        return best
22:    end function
23:    return parcourir(G, 0)
24: end function
```

5.3 Approche par algorithme glouton

Dans une seconde approche, on se propose de créer un algorithme glouton. Bien que l'on perdra l'optimalité globale, on garde un résultat localement optimal, pour une complexité moins lourde qu'avec le parcours en profondeur.

5.3.1 Algorithme

L'algorithme glouton fonctionne sur le principe suivant: pour une grille donnée, insérer toutes les tuiles une à une, en insérant à chaque tour le n-uplet (tuile, rotation, x, y) maximisant le score.

Algorithm 2: Renvoie un score localement optimal pour une grille donnée

```
1: function GLOUTON(G = (M, I))
2:     while M ≠ ∅ do
3:         bestLocalScore := 0
4:         bestTuile := (T = 0, R = 0, x = 0, y = 0)
5:         for T ∈ M do
6:             for R ∈ {NORD, SUD, EST, OUEST} do
7:                 for x ∈ {1, ..., n} do
8:                     for y ∈ {1, ..., n} do
9:                         if (T, R, x, y) peut être inséré dans I then
10:                            (Insérer) : supprimer T de M et ajouter (T, R, x, y) à I
11:                            (Vérifier optimalité local)
12:                            if bestLocalScore < score(G) then
13:                                bestLocalScore := score(G)
14:                                bestTuile := (T, R, x, y)
15:                            end if
16:                            (Dé-insérer) : ajouter T à M et supprimer (T, R, x, y) de I
17:                        end if
18:                    end for
19:                end for
20:            end for
21:        end for
22:        (Insérer optimum local) : supprimer T de M et ajouter (T, R, x, y) à I
23:    end while
24: end function
```

Cet algorithme a une complexité:

$$\begin{aligned} C(n,m) &= \sum_{k=1}^{m} 4 * n^2 * c * k \\ &= 4cn^2 \sum_{k=1}^{m} k \\ &= 4cn^2 \frac{m(m+1)}{2} \\ &\boxed{\simeq 2cn^2m^2} \end{aligned} \tag{4}$$

5.3.2 Experimentation

Les fichiers '.csv' correspondant aux données d'études sont disponibles dans ce même dossier.

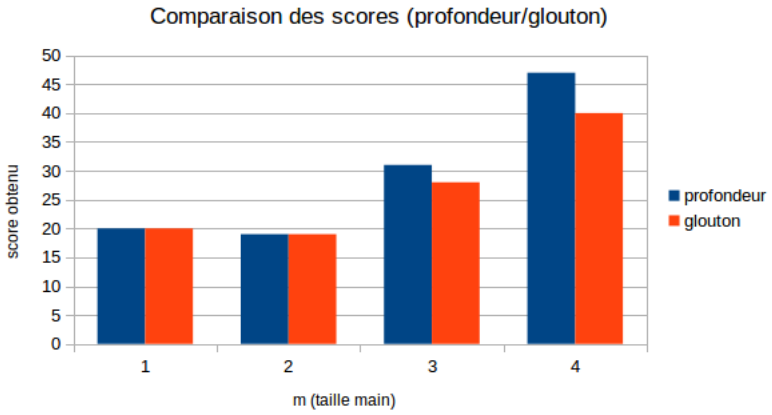

Figure 2: *Comparaison des scores obtenus entre les 2 algorithmes*

N.B: L'algorithme de parcours en profondeur étant trop lents, la comparaison n'a pas pu être effectué pour des valeurs plus grandes de m.

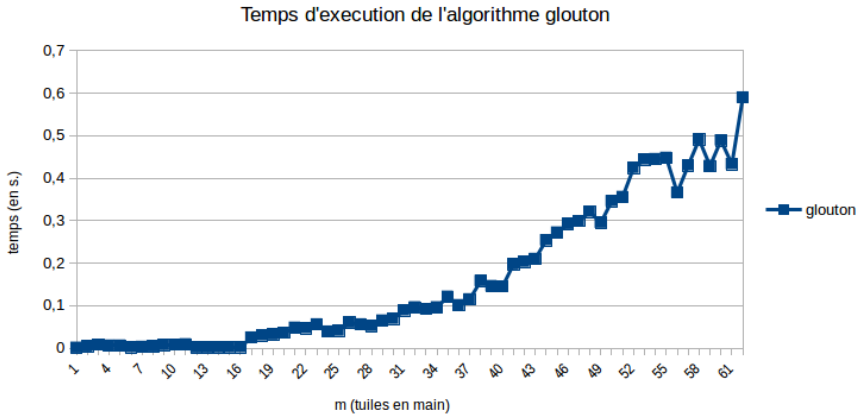

Figure 3: *Temps d'execution de la résolution par algorithme glouton, en fonction du nombre de tuiles en main: complexité quadratique en m*

Remarque Une comparaison temporelle des 2 algorithmes n'est pas pertinente, car l'algorithme en profondeur explose temporellement. Voici quelques valeurs qui ont pu être obtenus:

- $m = 1$; profondeur: $0.001s.$; glouton: $0.001s.$
- $m = 2$; profondeur: $0.007s.$; glouton: $0.004s.$
- $m = 3$; profondeur: $1.314s.$; glouton: $0.008s.$
- $m = 4$; profondeur: $131.6s.$; glouton: $0.006s.$

5.4 Conclusion

Ce document rapporte donc une étude et une comparaison de 2 algorithmes standarts de résolution: une approche par parcours en profondeur, et une approche par algorithme glouton.

PS.1 Une piste de résolution, qui n'a pas été exploité par manque de temps, est un mixage des 2 méthodes. Dans l'algorithme glouton, l'optimalité local est assuré sur chaques insertions. Une autre approche (plus lente) aurait consister à assurer l'optimalité sur x insertions. On insère les tuiles par paquet de x tuiles, et de manière optimal en faisant un parcours en profondeur 'local' sur x insertions.

Remarque Pour $x = 1$, on retrouve l'algorithme glouton, pour $x = m$ on retrouve l'algorithme de parcours en largeur.

Malheureusement, cette approche n'a pas été implementé et étudié par manque de temps. De plus, au regard des performances du parcours en largeur, seule les valeurs $x = 2$ ou $x = 3$ aurait été possibles.

PS.2 Une autre approche a été implémenté mais non-étudié / approfondie par manque de temps. Cette approche se base sur un parcours en profondeur, mais avec une fonction *filtre*. Ce *filtre* a pour but de limiter la profondeur du parcours: lorsque l'on remarque qu'une insertion ne satisfait pas les contraintes du filtre, le parcours s'arrête pour cette branche. Pour un filtre judicieusement choisit, on peut alors obtenir de meilleurs scores qu'avec l'algorithme glouton, pour un temps de recherche raisonnable. (plus lent que l'algorithme glouton, mais plus rapide que l'approche sans filtre)

6 Conclusion

Pour ce lot, le travail a été séparé de manière efficace, et la plupart du code a ainsi pu être programmé pendant la séance encadré. La résolution de ce type de problème (NP-difficle) a déjà été étudié en IPF (dans le projet SUBSET_SUM) pour Romain, Douha et Guangyue. Les différentes approches s'en sont inspirés (naïve, glouton, naïve avec filtre...). Ce sont des méthodes de résolutions classiques de problèmes NP-difficiles.