IG 4 - Algorithmique avancée

TP problème de la grille : confrontation de différentes approches algorithmiques

---

Avant de commencer

Télécharger les fichiers fournis dans "squelette_code_TPgrille.zip". Ces fichiers contiennent un squelette de code (avec des trous) qui compile, mais qui ne fait rien pour l'instant. Ne regardez pas le code pour l'instant, le code vous sera présenté à travers les questions. Commencez donc par la lecture des sections ci-dessous qui présentent le sujet globalement. Un ordre vous sera suggéré pour compléter les morceaux de code.
Ce TP est à faire par groupe de 2 (si quelqu'un se retrouve seul il peut trouver un binôme dans l'autre groupe, mais privilégiez les binômes au sein du même groupe), et sera corrigé par des tests automatiques. Certains de ces tests (les tests "publics") sont déjà présents dans le fichier TestsAutomatiques.java, et les autres sont gardés secrets. Votre note sera égale à la somme des points des tests passés ramené sur 20, ou 0 si votre code ne compile pas. Nous vous encourageons à tester votre code sur ces tests publics dès que cela est possible, et à aussi écrire d'autres tests.
Veillez aux choses suivantes pour éviter le 0 stupide :

- ne changez pas les noms des classes, noms/droits des attributs, et signatures des méthodes fournies dans le squelette de code (vous pouvez par contre ajouter vos méthodes auxiliaires)
- n'ajoutez pas de package au début des fichiers fournis (attention si vous utilisez un IDE qui en ajoute automatiquement), n'ajoutez pas non plus de nouveau fichier
- créez un dossier appelé TP-NOM1-NOM2, et placez-y tout le squelette de code fourni ainsi que le dossier images. Si par exemple Frederic Chopin et Sergeï Rachmaninov font le TP ensemble, ils devraient avoir un dossier "TP-CHOPIN-RACHMANINOV/" contenant les fichiers "Instance.java, Algos.java, etc"
- juste avant de soumettre votre travail : vérifiez, dans votre répertoire TP-NOM1-NOM2, que l'on peut bien faire sans erreur (car c'est ce qui sera fait par le script) :
  - rm *.class
  - javac TestsAutomatiques.java
  - java TestsAutomatiques
- le travail doit être remis sur MOODLE dans Rendu_TP_Grille, sous la forme **d'un fichier .zip contenant votre dossier principal du TP (par exemple "TP-CHOPIN-RACHMANINOV.zip"**. Veillez à ce qu'un seul membre du groupe seulement dépose le fichier.
- la date limite de rendu est le **dimanche 2 avril à 23h59**, mais n'hésitez pas à déposer une première version avant! (vous aurez le droit de re-déposer plusieurs fois).
- je ferai tourner le script une première fois le **lundi 27 mars à 12h**, si vous avez remis (même une version préliminaire!) votre travail à ce moment là, je pourrai vous prévenir si il y a un problème de compilation (sans pénalité à ce moment là).
- jouez le jeu : vous pouvez parler entre groupes, mais ne copiez collez pas de code (pour information, des scripts de détection de plagiat seront lancés)

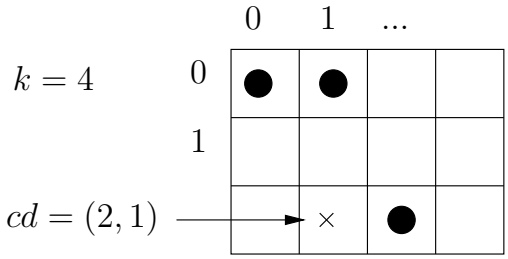
Le non respect de ces consignes entraînera un retrait de 5 points (le barème est sur 20).

1 Le problème de la grille

On considère le problème de la Collecte de Pièces dans une Grille (CPG) vu en TD :

- instance $I = (p, c_d, k)$:
  - un plateau rectangulaire p où chaque case contient 0 ou 1 pièce
  - une coordonnée de départ dans le plateau c_d
  - un nombre de pas k autorisés
- sortie : une solution s est une séquence $s = (c_0, \ldots, c_x)$ de $x \leq k + 1$ coordonnées telle que
  - $c_0 = c_d$ (on doit partir de c_d)
  - c_{i+1} est voisine de c_i (mouvement h, b, g ou d à chaque étape)
- but : maximiser $f(s)$: nombre de pièces ramassées en suivant s

Par exemple, pour l'instance I suivante :



Pour la solution $s = ((2,1),(2,2),(2,1),(1,1),(0,1))$, on a $f(s) = 2$ et $opt(I) = 2$.
La classe `Instance` représente une instance de ce problème. On fixera l'orientation du plateau en considérant que la case plateau[0][0] est en haut à gauche, et qu'on oriente ainsi : plateau[ligne][col]. On fixe également une numérotation des pièces du plateau : on numérote les pièces de haut en bas, puis de gauche à droite, par exemple sur l'instance de l'exemple, la pièce 0 est en $(0,0)$, la pièce 1 en $(0,1)$, et la pièce 2 en $(2,2)$.
Le but du TP est d'implémenter différents types d'algorithmes pour résoudre ce problème : glouton, FPT, recherche locale. Avant cela, vous allez commencer par écrire les méthodes de base de la section suivante.

2 Méthodes de base

1. Executez le main de la class `App` et prenez le temps de comprendre le code et le résultat de ce main.

2. Observez le code déjà fourni dans la classe `Instance`, puis complétez le code des méthodes (pensez à lancer les tests)

- public boolean estValide(Solution s)
- public int evaluerSolution(Solution s)

3 Algorithme glouton

Nous allons ici implémenter l'algorithme glouton qui, à chaque étape, se dirige vers la pièce restante la plus proche.

3. Dans la classe `Instance`, complétez le code des méthodes

- public ArrayList<Integer> greedyPermut()
- public Solution calculerSol(ArrayList<Integer> permut)

Vous pouvez maintenant tester (dans les TestsAutomatique et le main) la méthode
`Solution greedySolver(Instance i)` de la classe `Algos`. Constatez que sur l'instance in1 du main que le glouton n'est pas optimal!

4 Algorithmes FPT

Nous allons ici implémenter l'algorithme FPT en 4^k qui branche dans les 4 directions, ainsi que des améliorations de cet algorithme. Etant donné qu'un algorithme FPT résout un problème de décision, on utilisera dans cette partie des `InstanceDec` qui contiennent une instance i et un seuil c, et pour lesquelles on se demande si il est possible de ramasser dans i au moins c pièces.

4. Dans la classe `Algos`, complétez le code de la méthode

- public static Solution algoFPT1(InstanceDec id)

5. Expérimentez l'algorithme précédent :

- Utilisez l'instance in6 dans le main, et modifier sa valeur k pour trouver le petit k tel que l'on puisse ramasser 5 pièces.
- Remettez la valeur $k = 60$ pour in6, et calculez $c = greedy$ comme la valeur trouvée par le glouton. Observez que `algoFPT1(new InstanceDec(in6,c))` est .. très long !
- Changez le k de l'instance in6 pour le remplacer par $1, 2, ..$ (et garder $c = greedy + 1$ pour essayer de battre le glouton). A partir de quelle valeur de k l'algorithme prend plus d'une minute ?
- Utilisez `algoFPT1` sur l'instance in1 pour trouver une solution meilleure que celle du glouton.

Conclusion : écrit ainsi, l'algorithme FPT n'est utile que quand son paramètre k est petit, mais dans ce cas il garantit de trouver une solution optimale (en l'exécutant pour différentes valeurs de c comme vu en cours), contrairement au glouton.

Nous allons améliorer `algoFPT1` de deux façons.

4.1 Amélioration 1 : utilisation d'une borne supérieure.

6. Dans la classe `Instance`, complétez le code de la méthode

- public int borneSup()

Ensuite, modifiez le code de `algoFPT1` pour ajouter un cas base utilisant cette borne supérieure.

7. A nouveau, changez le k de l'instance in6 pour le remplacer par $1, 2, ..$ (et garder $c = greedy + 1$ pour essayer de battre le glouton). A partir de quelle valeur de k l'algorithme prend plus d'une minute ?

4.2 Amélioration 2 : transformation en programmation dynamique.

Si l'on imagine l'arbre de calcul de `algoFPT1` dans lequel chaque noeud correspond à une instance, on peut imaginer que de nombreux noeuds de l'arbre sont égaux (au sens : correspondent à la même instance), et donc que de nombreuses instances sont résolues inutilement plusieurs fois. Nous allons pallier à ce problème en transformant `algoFPT1` en programmation dynamique. En effet, étant donné qu'on ne collecte qu'au plus c pieces, et que seules les pièces dans un carré de taille $\mathcal{O}(k^2)$ cases centrées sur point de départ peuvent être concernées, le nombre de sous ensembles différents de pièces collectées (et donc de plateaux différents) est en $\mathcal{O}(k^{2c})$. Etant donné qu'une instance de décision comprend un plateau, un entier k, et c, le nombre possible d'instances de décision différentes est en $\mathcal{O}(k^{2c}kc)$. On peut donc espérer gagner en complexité par rapport à `algoFPT1` (qui est en $\mathcal{O}(4^4 poly(n))$).lorsque c est petit.

8. Dans la classe `Algos`, complétez le code des méthodes

- public static Solution algoFPT1DP(InstanceDec id, HashMap<InstanceDec,Solution> table)
- public static Solution algoFPT1DPClient(InstanceDec id)

9. A nouveau, changez le k de l'instance in6 pour le remplacer par $1, 2, ..$ (et garder $c = greedy + 1$ pour essayer de battre le glouton). A partir de quelle valeur de k l'algorithme prend plus d'une minute ?

5 Recherche Locale

10. Compléter le code de la classe `HillClimbing`.

11. Compléter le code de la classe `ElemPermutHC`.

12. Expérimentez la recherche locale :

- dé-commentez et testez la section "comparaison des algorithmes" du main qui lance le glouton et la recherche locale sur un grand nombre d'instance, et compare la somme des résultats obtenus.
- modifiez ce main pour compter sur combien d'instances la recherche locale est meilleure que le glouton.
- recommencez les comparaisons précédentes en faisant varier les paramètres de la recherche locale : nombre de répétition, distance (ElemPermutHC.setDist()) à laquelle on génère les voisinages