INF8775 – Analyse et conception d'algorithmes

TP3 – Hiver 2019

Nom, prénom, matricule des membres	Luc Courbariaux, 1670433
Note finale / 14	0

Informations techniques

- Répondez directement dans ce document ODT avec LibreOffice. Utilisez LibreOffice et non Word, sinon ce document sera corrompu.
- La correction se fait à même le rapport.
- Avant le 16 avril à 23h59 pour les deux groupes (B1 et B2), vous devez faire une remise électronique en suivant les instructions suivantes :
  - Le dossier remis doit se nommer matricule1_matricule2_tp3 et doit être compressé sous format zip.
  - À la racine de ce dernier, on doit retrouver :
    - Ce rapport sous format ODT.
    - Un script nommé *tp.sh* servant à exécuter votre algorithme. L'interface du script est décrite à la fin du rapport.
    - Un fichier texte nommé *emails.txt* contenant le courriel de chaque membre de l'équipe
    - Le code source et les exécutables
- Vous avez le choix du langage de programmation utilisé. Le code et les exécutables soumis devront être compatible avec les ordinateurs de la salle L-4714. Conseil pour le TP3 : Compilez avec -O3 si vous utilisez C++.
- Si vous utilisez des extraits de codes (programmes) trouvés sur Internet, vous devez en mentionner la source, sinon vous serez sanctionnés pour plagiat.

Mise en situation

Le dernier travail pratique se fera dans le cadre du concours du meilleur algorithme pour la session d'hiver 2019. Le travail demandé consiste à concevoir et implanter un algorithme de votre cru pour résoudre un problème combinatoire. Le classement des équipes déterminera votre note pour la qualité de l'algorithme. Votre algorithme sera exécuté sur 3 exemplaires de notre choix pendant 3 minutes.

Dans un élan de charité, vous décidez d'offrir à la fondation Child's Play les Lego que vous aviez lorsque vous étiez enfant. Vous les retrouvez pêle-mêle dans une grosse boîte chez vos parents. Vous réalisez que vous n'avez plus les plans des modèles, et que même si vous les aviez il vous manquerait sûrement plusieurs pièces perdues au fil des années pour avoir des modèles complets.

Heureusement, la compagnie Lego garde un catalogue des plans de tous ses modèles, et offre un service en ligne pour commander des pièces. Vous devez trouver un ensemble de modèles utilisant la totalité de vos pièces, ce qui nécessitera possiblement de commander certaines pièces s'il en manque. Malgré toutes vos bonnes intentions, vous êtes quand même un pauvre étudiant et vous devez donc minimiser le coût d'achat des pièces manquantes. Vous pouvez construire plusieurs modèles identiques.

Plus formellement :

Soit $P = \{p_1, p_2, ..., p_g\}$ le prix des différents types de pièces de Lego, et $M = \{M_1, M_2, ..., M_d\}$ les différents types de modèles (chaque modèle est un ensemble de pièces). Les pièces que vous avez en votre possession sont un vecteur de longueur g. La réponse est que vous devez fournir est un vecteur de longueur d représentant les modèles que vous voulez former et leur nombre. Un vecteur de réponse valide doit contenir des modèles qui utilisent l'ensemble de vos pièces, et si des pièces additionnelles sont nécessaires la somme de leur coût doit être minimal.

Le rapport pour ce dernier travail pratique est assez succinct. Vous êtes encouragés à terminer ce travail assez tôt afin de ne pas compromettre la préparation à vos examens finaux.

Exemple

Supposons que nous avons trois types de pièces ($g = 3$) et deux sortes de modèles ($d = 2$) :
Prix des pièces : [6, 3, 4]
Pièces en votre possession : [10, 11, 8]
Modèles :
M_1 : [5, 6, 7]
M_2 : [2, 0, 1]
Vous avez en votre possession 10 pièces du premier type, 11 pièces du second type, et 8 pièces du troisième type. Le modèle M_1, par exemple, utilise 5 pièces du premier type, 6 pièces du second type, et 7 pièces du troisième type. Vous décidez de construire deux modèles M_1 et aucun modèle M_2. Votre vecteur de réponse est donc [2, 0]. Votre solution requiert donc :

- (2 * 5) + (0 * 2) = 10 pièces du premier type
- (2 * 6) + (0 * 0) = 12 pièces du second type
- (2 * 7) + (0 * 1) = 14 pièces du troisième type

Toutes les pièces qui sont en votre possession ([10, 11, 8]) trouvent leur place parmi les pièces nécessaires à la solution ([10, 12, 14]), donc votre solution est valide. Il vous faut cependant acheter certaines pièces manquantes pour compléter les deux modèles :

- Il faut acheter 10 – 10 = 0 pièces du premier type, à un coût de 6 par pièce (0 * 6 = 0)
- Il faut acheter 12 – 11 = 1 pièce du second type, à un coût de 3 par pièce (1 * 3 = 3)
- Il faut acheter 14 – 8 = 6 pièces du troisième type, à un coût de 4 par pièce (6 * 4 = 24)

La valeur de la solution [2, 0] est donc 0 + 3 + 24 = 27

Jeu de données

Pour tester votre algorithme, vous disposez d'un jeu de données de 8 exemplaires.
La première ligne contient le nombre de différents types de pièces *g*.
La ligne suivante contient le nombre de pièces de chaque type que vous avez en votre possession.
La prochaine ligne indique le prix associé à l'achat de chaque type de pièce.
La ligne suivante indique le nombre *d* de différents modèles,
et les *d* lignes suivantes indiquent de quelles pièces sont formés ces *d* modèles.

Description de votre algorithme

0	/ 1 pt

Décrivez brièvement votre algorithme en quelques phrases.

C'est un algorithme qui trouve une première solution approximative sous forme d'un vecteur de nombres réels non négatifs (via l'algorithme NNLS de Lawon-Hanson) puis l'affine par recherche aléatoire.

Pour expliquer un peu plus :
Le problème fourni peut être vu de cette manière : On a les vecteurs *pièces* et *coûts* de taille m, une matrice *modèles* de taille (m, n), tous formés d'entiers naturels (N), et on veut trouver le vecteur *solution* de taille n d'entiers naturels tel que la somme du vecteur (*modèles* x *solution* – *pièces*) x *coûts* soit minimale.
L'algorithme NNLS quant à lui prend une matrice *C* et un vecteur *vec* de nombres réels et cherche un vecteur *solution* de nombres réels non négatifs tel que la norme du vecteur C x *solution* – *vec* soit minimale.
L'algorithme commence par minimiser *modèles* x *solution* – *pièces* avec NNLS puis ajuster cela avec une recherche probabiliste (qui se base sur le reste des nombres réels trouvés par NNLS).

Présentation de votre algorithme

0	/ 2 pt

Sous forme de pseudo-code et incluant une analyse de complexité théorique des principales fonctions. Si vous préférez écrire vos équations en Latex, vous pouvez ajouter un pdf à la remise avec la réponse à cette question et le mentionner ici.

Le pseudo-algorithme de NNLS :

- Inputs:
  - a real-valued matrix A of dimension $m \times n$
  - a real-valued vector $\mathbf{y}$ of dimension m
  - a real value ε, the tolerance for the stopping criterion
- Initialize:
  - Set $P = \emptyset$
  - Set $R = \{1, \ldots, n\}$
  - Set $\mathbf{x}$ to an all-zero vector of dimension n
  - Set $\mathbf{w} = A^{\mathrm{T}}(\mathbf{y} - A\mathbf{x})$
- Main loop: while $R \neq \emptyset$ and $\max(\mathbf{w}) > \varepsilon$,
  - Let j in R be the index of $\max(\mathbf{w})$ in $\mathbf{w}$
  - Add j to P
  - Remove j from R
  - Let A^P be A restricted to the variables included in P
  - Let s be vector of same length as $\mathbf{x}$. Let s^P denote the sub-vector with indexes from P, and let s^R denote the sub-vector with indexes from R.
  - Set $s^P = ((A^P)^{\mathrm{T}} A^P)^{-1} (A^P)^{\mathrm{T}}\mathbf{y}$
  - Set s^R to zero
  - While $\min(s^P) \leq 0$:
    - Let $\alpha = \min(\frac{x_i}{x_i - s_i})$ for i in P where $s_i \leq 0$
    - Set $\mathbf{x}$ to $\mathbf{x} + \alpha(s - x)$
    - Move to R all indices j in P such that $x_j = 0$
    - Set $s^P = ((A^P)^{\mathrm{T}} A^P)^{-1} (A^P)^{\mathrm{T}}\mathbf{y}$
    - Set s^R to zero
  - Set $\mathbf{x}$ to s
  - Set $\mathbf{w}$ to $A^{\mathrm{T}}(\mathbf{y} - A\mathbf{x})$

source : https://en.wikipedia.org/wiki/Non-negative_least_squares#Algorithms

Cet algorithme est formé de deux boucles internes qui s'exécutent tant qu'un critère d'erreur est supérieur à une certaine valeur ; en pratique un nombre maximum d'itérations totales de la boucle la plus interne est utilisé (ici égal à 3 x *n*, où *n* est l'une des dimensions de *modèles*) ; son opération la plus complexe asymptotiquement est un calcul de l'inversion d'une matrice interne à l'algorithme de même dimension que *modèles* selon la méthode de Moore-Penrose (fonction *numpy.linalg.pinv*) dont la complexité est $m\ n^2$. Donc, sa complexité asymptotique en pire cas est en $m\ n^3$.

Pour la seconde partie de recherche aléatoire, le code est :

```
# appel de NNLS pour la solution initiale
float_solution = lsqnonneg( mat.T, pieces)[0]
# comme NNLS return des valeurs non-entières, la solution entière minimale (pas nécessairement légale) est calculée
min_solution = np.floor(float_solution)

# le score de la meilleure solution (coût total des pièces supplémentaires, doit être minimisé)
best_score = 99999999999999999.
# la meilleure solution trouvée
best_solution = min_solution


# un grand nombre d'itérations
ITERATIONS = 200000

# dans le code remis, sera remplacé par un while pour profiter au maximum des 3 minutes
for i in range(ITERATIONS):
        # on cherche d'abord autour de la solution données par NNLS, puis autour de la meilleure solution
        if i< ITERATIONS*3/4:
                evaluated = min_solution
        else:
                evaluated = best_solution

        # les probabilités d'augmenter le nombre d'un modèle est basé sur la solution de NNLS
        odds = float_solution - evaluated
        odds[odds<0.] = 0.
        odds = odds%1.

        # génère un changement aléatoire (entre -2 and +2)
        random_guess = []
        for x in odds:
                seed = random.random()
                if x < 0.1:
                        ret = -2 if seed<0.03 else -1. if seed < 0.2     else 0. if seed > 0.4 else 1. if seed < 0.96 else 2.
                else:
                        ret = -2 if seed<0.03 else -1. if seed < 0.3*x else 0. if seed > x    else 1. if seed < 0.96 else 2.
                random_guess.append(ret)
        random_guess = np.asarray(random_guess)
```

```
# la solution proposée
solution = np.abs(evaluated + random_guess)

# calcule les pièces qui restent de la solution proposée après qu'on ait enlevées celles originelles
remaining_pieces = solution * mat - pieces
remaining_pieces = np.squeeze(np.asarray(remaining_pieces))

# vérifie si la solution est légale (toutes les pièces originales sont utilisées)
illegal = np.any(remaining_pieces < 0)
score = np.sum(remaining_pieces * dataset.costs)

# si la solution est légale et meilleure, elle est choisie pour l'instant
if not illegal and  score<best_score:
                    best_score = score
                    best_solution = solution
                    print(*map(int, best_solution))
```

Sa complexité est celle de son instruction la plus complexe le calcul de *remaining_pieces = solution * mat* qui est de complexité $m\ n^2$.

La complexité de l'algorithme entier est donc $m\ n^3$.

Justification de l'originalité de votre algorithme

0	/ 3 pt

La conception de votre algorithme sera jugée avec les critères suivants :

- *Lien avec le contenu du cours*
- *Originalité*
- *Initiative*

Le problème donné avait une grande complexité : si on considère le nombre de fois qu'un modèle peut être choisi comme étant *k,* le nombre de possibilités à investiguer est k^n : soit plus de 10^{22} possibilités si $k = 3$ et $n = 50$. Puisque chaque solution doit être évaluée via un calcul avec la matrice *modèles*, cela donnait une complexité dans $m\, n^2\, k^n$. Une recherche brute n'était donc pas raisonnablement possible.

Une recherche aléatoire brute était faisable, mais donnait des résultats loin d'être aussi optimaux que possible. Le fait que la situation soit proche d'un problème d'algèbre linéaire m'a fait penser à caculer l'inverse de la matrice *modèles* pour trouver la solution à l'équation *modèles* x *solution = pièces.*
Le problème était que les solutions comprenaient des valeurs négatives et non entières. Cependant cette première idée m'a permis de trouver l'algorithme NNLS : bien que les solutions trouvée par ce dernier était non-entières, il s'agissait d'un bon début pour un algorithme d'amélioration locale.

Mais même avec cette première solution, le champs des possibilités était grand : le fait d'utiliser une recherche probabiliste autour de la solution permet de palier à cela. De plus, si on prend en compte les modalités de l'épreuve (3 minutes d'exécution continue), cela avait aussi l'avantage de tirer au maximum parti du temps donné.

Votre algorithme est-il assuré de trouver une solution optimale?

0	/ 1 pt

Aucune justification requise. Un simple "oui" ou "non" suffit.

Non

Autres critères de correction

Respect de l'interface tp.sh

0	/ 1 pt

Utilisation :

./tp.sh -e [path_vers_exemplaire]

Chaque fois qu'une meilleure solution est trouvée, votre programme affiche le vecteur de réponse (la liste des modèles de la solution, et non la valeur de la solution).

Important : l'option -e doit accepter des fichiers avec des paths absolus.

Le script tp.sh ne vous est pas fourni, mais vous pouvez facilement adapter celui du TP2.

Voici un exemple d'affichage pour un exemplaire ayant 11 modèles (dans cet exemple, après avoir trouvé une solution initiale, le programme en a par la suite trouvé une meilleure) :

```
5 4 0 11 0 2 1 4 0 0 13
0 4 1 9 2 0 7 9 15 3 0
```

Un script de vérification de solution vous est fourni (sol_check.py, les instructions d'utilisation sont dans le code source). Ce script vous indiquera si l'affichage est correct et si votre solution est valide. C'est ce script qui sera utilisé pour la correction, donc assurez-vous qu'il reconnaisse vos solutions.

Qualité de l'algorithme

0	/ 4 pt

Qualité du code

0	/ 1 pt

Présentation générale (concision, qualité du français, etc.)

0	/ 1 pt

Pénalité pour retard ou autre

0

- -1 pt / journée de retard, arrondi vers le haut. Les TPs ne sont plus acceptés après 3 jours.