INF8775 – Analyse et conception d'algorithmes

TP2 – Hiver 2023

Nom, prénom, matricule des membres	
Note finale / 13	

Informations techniques

- Répondez directement dans ce document. Veuillez ne pas inclure le texte en italique servant de directive.
- La correction se fait sur ce même rapport.

  Vous devez faire une remise électronique sur Moodle avant le

  14 mars à 23h59 pour le groupe B2

  21 mars à 23h59 pour le groupe B1

  en suivant les instructions suivantes :
- Vos fichiers doivent être remis dans une archive zip à la racine de laquelle on retrouve :
  - Ce rapport sous format ODT ou docx.
  - Un script nommé *tp.sh* servant à exécuter les différents algorithmes du TP. L'interface du script est décrite à la fin du rapport.
  - Le code source et les exécutables.
- Vous avez le choix du langage de programmation utilisé mais vous devrez utiliser les mêmes langage, compilateur et ordinateur pour toutes vos implantations. **Le code et les exécutables soumis devront être compatibles avec les ordinateurs de la salle L-4714.**
- 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

Ce travail pratique se répartit sur deux séances de laboratoire et porte sur l'analyse et la conception d'algorithmes développés suivant différents patrons de conception.

On vous demande d'y résoudre le problème du voyageur de commerce (*Traveling Salesman Problem*, TSP) en utilisant trois patrons de conception distincts :

1. Algorithme glouton : en démarrant d'une ville arbitraire et en faisant une tournée en sélectionnant à chaque fois le plus proche voisin ;
2. Algorithme de programmation dynamique : en résolvant le problème à travers la relation de récurrence suivante :

$$D[i,S] = \begin{cases} \min_{j \in S} d_{ij} + D[i, S \setminus \{j\}] & \text{si } S \neq \emptyset \\ d_{i0} & \text{sinon} \end{cases}$$

3. Algorithme approximatif (1-relatif) : en construisant un arbre sous-tendant minimum (Minimum Spanning Tree, MST) du graphe et en le parcourant en préordre à partir d'une ville arbitraire.

On s'intéresse dans ce TP au problème du TSP métrique : les villes appartiennent à un plan N^2 muni de la distance euclidienne **arrondie à l'entier le plus proche**.

Jeux de données

Pour tester les algorithmes, vous devez générer des jeux de données en utilisant le script fourni :

```
$ ./inst_gen.py [-h] -s NB_VILLES [-n NB_EXEMPLAIRES] [-x PRÉFIXE]
```

On suggère d'utiliser le script bash suivant pour automatiser la génération des exemplaires :

```
#!/bin/bash

# Pour glouton et approx
for n in {"1000","5000","10000","50000","100000"}; do
    ./inst_gen.py -s $n -n 5
done

# Pour tous les algorithmes
for n in {"5","10","15","20","25"}; do
    ./inst_gen.py -s $n -n 5 -x DP
done
```

La première ligne de chaque exemplaire contient le nombre de villes à visiter (qui est aussi la taille du problème *N*). Les *N* lignes subséquentes contiennent chacune les coordonnées d'une ville à la fois, séparées par des espaces.

Les coordonnées des villes sont chacune comprises entre 0 et 2000 et aucune paire de villes n'est telle que la distance entre elles est de 0.

Présentation des résultats

0	/ 4 pt

Tableau des résultats

N\ Algo	N run	Glouton (temps ms | distance)		Approximatif (temps ms | distance)	
1000	50	2.082	58’091	6.003	63’538
5000	50	50.39	126’477	130.9	140’762
10000	50	196.0	177’771	518.7	198’344
50000	50-25	4’958	392’415	14’767	441’777
100000	25-10	19’905	552’918	60’272	623’947

Une liste de 25 points remplit la RAM pour l’algorithme de programmation dynamique.

N \ Data	Prog Dyn - Temps	Prog Dyn - Distance	Glouton	Approximatif
5	0.01268	4’298	4’492	4’539
10	1.203	5’807	6’628	6’723
15	122.3	6’785	7’831	8’244
20	13’150	7’742	9’134	9’529
25	N/A	N/A	N/A	N/A

Analyse et discussion

0	/ 6 pt

Faites une analyse asymptotique du temps de calcul pour chaque algorithme.

Dans analyse.pdf.

Servez-vous de vos temps d'exécution pour confirmer et/ou préciser l'analyse asymptotique théorique de vos algorithmes avec la méthode hybride de votre choix.

On trouve bien une complexité polynomiale de degré 2 pour les algorithmes glouton et approximatifs. L'algorithme de programmation dynamique a une complexité exponentielle.

La complexité quadratique était attendue pour les deux premiers algorithmes.

- *Pour l'algorithme glouton on itère sur la liste des villes et pour chaque ville on itère sur les villes restantes, on a donc environ* $\sum_{i=1}^{n} i(i-1) \sim n^2$ *opérations.*
- *Pour l'algorithme approximatif. La complexité vient de l'utilisation de l'algorithme de Prim sur un graph complet. On a obligatoirement une complexité quadratique par ce simple fait. Le reste des opérations ont une complexité moindre (le parcours de l'arbre a une complexité linéaire) donc elles n'influencent pas sa complexité globale.*

L'algorithme de programmation dynamique a une complexité exponentielle à cause de sa vérification de toutes les possibilités. En effet, pour chaque nœeuf, avec un ensemble de taille k, on doit faire un minimum sur k *parmi* $n-2$ *éléments, et ayant* $\sum_{k=0}^{n-2} C_k^n = 2^{n-2}$ *on a bien une complexité exponentielle. En revanche, en pratique on trouve un exposant plus proche de 2.5, sûrement dû à une mauvaise optimisation de cet algorithme.*

Discutez des trois algorithmes en fonction de la qualité respective des solutions obtenues, de la consommation de ressources (temps de calcul, espace mémoire) et de la difficulté d'implantation.

L'algorithme glouton est en moyenne plus rapide et donne de meilleures solutions. Il n'est pas trop compliqué à implémenter, le seul point à surveiller est la consistance des indices si l'on retire des villes au fur et à mesure. Il a un temps de calcul raisonnable quadratique, il ne fournit cependant pas de solution optimale. En termes d'usage mémoire il est très modéré, il n'y a besoin que d'une liste d'indices. On a donc un usage d'espace mémoire linéaire.

L'algorithme approximatif est assez intuitif. Dans un langage haut niveau son implémentation peut être assez simple. Il se décompose facilement en sous-étapes. Cependant, malgré sa complexité polynomiale comme l'algorithme glouton, il est en moyenne plus lent et fournit de moins bonnes solutions. Il n'est donc pas à privilégier. Pour l'usage mémoire on stocke d'abord les arêtes qui occupent un espace linéaire (on a $n-1$ *arêtes). Le stockage de l'arbre peut également être fait de manière linéaire. Et le parcours DFS utiliser également un stockage borné par la profondeur de l'arbre, ici borné par* n *également.*

L'algorithme de programmation dynamique est très sympathique car aussi assez intuitif. On notera qu'il est assez peu souhaitable de l'implémenter dans un langage plus bas niveau comme C ou C++ surtout lorsque l'on souhaite limiter l'espace mémoire, une implémentation dans un langage

comme python serait préférable pour cet algorithme. Il a une consommation de mémoire et temporelle exponentielle donc son seul avantage réside dans l'exactitude de sa réponse.

Indiquez sous quelles conditions vous utiliseriez chaque algorithme.

Pour les exemplaires larges l'algorithme glouton est à privilégier absolument, il est rapide et assez efficace en mémoire donc un candidat de choix. On peut essayer de l'améliorer en optimisant le choix de la première ville pour de meilleures performances.

Pour des exemplaires avec un faible nombre de points on privilégiera l'algorithme de programmation dynamique, d'autant plus si l'on souhaite obtenir une solution plus exacte.

On vous fournit 5 exemplaires difficiles à résoudre jusqu'à optimalité[1] (fichiers avec préfixe hard). Tentez de les résoudre à l'aide de vos algorithmes glouton et approximatif et discuter des écarts obtenus.

Fichier	N villes	Solution optimale	Glouton	Approximatif
hard_N52	52	551609	640227	852753
hard_N91	91	1228726	1342503	1569069
hard_N130	130	1928734	2080251	2397227
hard_N169	169	2600546	2808987	3850171
hard_N199	199	3139778	3327628	3909166

Comme toujours, l'algorithme glouton trouve une meilleure solution que l'algorithme approximatif. Mais on peut voir que les deux sont très loin de l'optimalité.
L'algorithme glouton arrive à rester proche des solutions optimales. Alors que pour certains problèmes, l'algorithme approximatif est significativement plus mauvais (par exemple l'exemplaire N169).

[1] Tirés de http://www.or.uni-bonn.de/~hougardy/HardTSPInstances.html, les coordonnées peuvent dépasser 2000.

Autres critères de correction

Respect de l'interface `tp.sh`

	/ 1 pt

Utilisation :

```
$ ./tp.sh -a {glouton, progdyn, approx} -e CHEMIN_EXEMPLAIRE [-p] [-t]
```

Arguments optionnels :

> `-p` affiche dans l'ordre, sur chaque ligne, les indices des villes à visiter en commençant par 0 et en finissant par 0, sans texte superflu. Rapportez le chemin tel que la deuxième ville visitée ait un indice inférieur à celui de l'avant dernière ville affichée (voir exemple).
> `-t` affiche le temps d'exécution en millisecondes, sans unité ni texte superflu.

Important : l'option `-e` doit pouvoir accepter des chemins absolus.

```
user@host folder $ ./tp.sh -p -e N5_0 -a progdyn
0
1
4
3
2
0
```

Qualité du code

	/ 1 pt

Présentation générale

	/ 1 pt

- Concision
- Qualité du français

Pénalité retard

0

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