

**IFT436**

**Devoir n°5**

**Présenté à:**

**Richard St-Denis**

**Par:**

**Olivier Perrault**

**Matricule: 16212377**

**Département d’informatique**

**Révisé le 29 octobre 2018**

**0 L'environnement d'exécution ne biaise pas les temps d'exécution.**

**1 La génération des données aléatoires n'est pas biaisée**

Par définition, un ensemble de données de taille n est aléatoires si la probabilité de choisir un élément est le même que de choisir n’importe quel autre élément.

Dans le fichier *tsp.py*, la fonction *test\_randint* vérifie que la fonction *random.randint*. de la librairie standard répond au critère ci-dessus avec une marge d’erreur suffisamment basse. Cela consiste à générer n ensembles de données de taille n comprenant les valeurs 0 jusqu’a n-1. Pour chaque valeur on effectue une moyenne. Finalement on évalue la moyenne des moyenne obtenue pour obtenir un nombre préférablement très proche de 1.0.

En regardant les fichiers *randint0, randint1, randint2,* observons les sorties respective de la procédure *test\_randint :*

*Upon generating 100 datasets of size 100, random.randint outputs each entry between 0 and 99, 0.9899999999999999 times on average.*

*Upon generating 500 datasets of size 500, random.randint outputs each entry between 0 and 499, 0.9979999999999994 times on average.*

*Upon generating 1000 datasets of size 1000, random.randint outputs each entry between 0 and 999, 0.9989999999999991 times on average.*

Il est possible générer de nouveaux ensembles de donnes et de répéter ces étapes et nous observons des résultats similaires. Nous observons un marge d’erreur suffisamment faible, donc nous pouvons en conclure que la fonction *random.randint* n’est pas biaisée de manière significative.

**2 Études d’algorithmes et heuristique problème du voyageur de commerce**

**2.1 Brute force**

**2.1.1 Complexité du temps de calcul**

En se référant au fichier *brute\_force.py*

Lignes 24-27

On observe le pire cas lorsqu’à chaque fois, on distingue le chemin candidat du plus petit chemin grâce à la distance entre les deux dernières villes. Le coût est linéaire.

Lignes 5-20

Il est possible de démontrer que la fonction *brute\_force.next\_permutation* se fait en temps contant amorti. L’implantation utilisée [0] de l’algorithme de Narayana Pandita utilise environ 3 comparaisons et 1.5 swaps par permutation, lorsque ont amorti sur l’ensemble des séquences [1]. Donc on en déduit que :

Lignes 39-49

Il existe permutations entre de villes en omettant la ville de départ. Donc étant donné les résultats ci-dessus :

**2.1.2 Complexité mémoire**

**2.1.3 Tests de performance**

**2.2 Nearest neighbor**

**2.2.1 Complexité du temps de calcul**

En se référant au fichier *nearest\_neighbor.py*

Lignes 11-15

La boucle *for* effectue n itérations avec n étant le nombre de villes. On observe que la fonction *util.bit\_check* s’effectue en temps constant puisque cela consiste uniquement d’une opération binaire. Donc :

Lignes 9-21

La boucle *while* est exécutes exactement n fois puisque chaque itération *smallest* représente un diffèrent sommet jusqu’à ce qu’il n’y ait plus d’options. Donc :

**2.3 Nearest insertion**

**2.3.1 Complexité du temps de calcul**

En se référant au fichier *nearest\_insertion.py,*

Lignes 20-23

On observe que la boucle *for* central effectue un nombre croissant d’itérations.

Lignes 14-26

Étant donné que la deuxièmes boucle *for,* effectue au total n itérations.

Lignes 11-32

La boucle *while* est exécutes exactement n fois puisque chaque itération *smallest* représente un diffèrent sommet jusqu’à ce qu’il n’y ait plus d’options. Donc :

**2.2.2 Complexité mémoire**

**2.2.3 Tests de performance**

**2.4 Cheapest insertion**

**2.4.1 Complexité du temps de calcul**

En se référant au fichier *cheapest\_insertion.py,*

Lignes 11-17

On observe que la boucle *for* central effectue n d’itérations.

Lignes 9-23

La boucle *while* est exécutes exactement n fois puisque chaque itération *smallest* représente un diffèrent sommet jusqu’à ce qu’il n’y ait plus d’options. Donc :

**2.2.2 Complexité mémoire**

**2.2.3 Tests de performance**

**2.5 Minimum spanning tree**

**2.5.1 Complexité du temps de calcul**

L’algorithme consiste a dans un premier temps construire l’arbre sous-tendant de coût minimal en utilisant l’algorithme de Prim. Par la suite il est question de faire un parcours en profondeur à travers l’arbre résultant afin d’obtenir un chemin. Finalement, il reste à connecter la destination finale du parcours en profondeur a la racine de l’arbre.

Nous avons démontré en classe qu’il est possible d’effectuer l’algorithme avec complexité

En se référant au fichier *minimum\_spanning\_tree.py*

Lignes 75-78

On observe que le nombre d’itérations effectuer par la boucle diminue à chaque itération de la boucle extérieur. De plus l’appel à *heapq.\_siftdown* est exécuté avec complexité :

Lignes 69-78

La boucle *while* est exécutes exactement n fois puisque chaque itération on extrait le nœud avec la plus petite clé du *heap* jusqu’à ce qu’il n’y ait plus d’options. De plus, on observe que *heapq.heappush* est exécutée avec temps :

Donc étant donné les résultats ci-dessus:

Le parcours en profondeur est effectué dans les méthodes m*inimum\_spanning\_tree.MST.to\_hamiltonian\_path* et *minimum\_spanning\_tree.MST.\_dfs\_visit*. On note que parmi les appelle récursif, *\_dfs\_visit* est appelé n fois, soit une fois pour chacun des nœuds de l’arbre sous-tendant. De plus pour chacun de ces appels la boucle des lignes 38-40 consiste n itérations. Donc etant donnes ces resultats et ceux decrits ci-dessus.

**2.2.2 Complexité mémoire**

**2.2.3 Tests de performance**

**3 Conclusions**

5. **Références**

[0]

Project Nayuki. (2018, June 20). Next lexicographical permutation algorithm. Retrieved October 29, 2018, from <https://www.nayuki.io/page/next-lexicographical-permutation-algorithm>

[1]

std::next\_permutation. (2018, June 15). Retrieved October 29, 2018, from <https://en.cppreference.com/w/cpp/algorithm/next_permutation>

|  |  |  |
| --- | --- | --- |
|  |  |  |
| **Votre étude comporte au moins trois algorithmes.** |  |  |
| **Les graphes sont représentés par une**  **matrice d'adjacence.** |  |  |
| **Les graphes sont représentés par des listes d'adjacence.** |  |  |
| **La programmation des algorithmes est uniforme.** |  |  |
| **Les structures de données sont uniformes d'un algorithme à l'autre.** |  |  |
| **Les complexités de calcul des algorithmes sont présentes.** |  |  |
| **Les complexités en espace des algorithmes sont présentes.** |  |  |
| **Les stratégies de conception des algorithmes sont différentes.** |  |  |
| **Au moins 50 échantillons de données ont été générés** |  |  |
| **La taille des échantillons de données varie progressivement de 10 à 10 000** |  |  |
| **Des échantillons de données considèrent les meilleurs cas (si applicable).** |  |  |
| **Des échantillons de données considèrent**  **les pires cas (si applicable).** |  |  |
| **Chaque échantillon a été soumis à chacun**  **des algorithmes.** |  |  |
| **La prise de mesure du temps de calcul a été réalisée correctement.** |  |  |
| **La génération des données aléatoires n'est pas biaisée.** |  |  |
| **La comparaison des temps de calcul des algorithmes a été mise en évidence sous la**  **forme d'un ou plusieurs graphiques.** |  |  |
| **La comparaison des temps de calcul des algorithmes a été mise en évidence avec les résultats théoriques sous la forme de d'un ou plusieurs graphiques.** |  |  |
| **L'environnement d'exécution ne biaise pas**  **les temps d'exécution.** |  |  |
| **Les références ont des travaux empruntés sur**  **le Web ou ailleurs sont présentes dans le**  **rapport.** |  |  |
| **Le rapport comporte au moins 10 pages**  **(sans ce tableau).** |  |  |
|  |  |  |
|  |  |  |
|  |  |  |