

**IFT436**

**Devoir n°5**

**Présenté à:**

**Richard St-Denis**

**Par:**

**Olivier Perrault**

**Matricule: 16212377**

**Département d’informatique**

**Révisé le 29 octobre 2018**

**1. Facteurs externe pouvant biaiser les résultats d’exécution.**

Afin de réduire le billet par rapport au temps d’exécution d’un algorithme en particulier à un niveau acceptable, plusieurs précautions sont prisées.

**1.1 Environnement d’execution**

Chacun des algorithmes sont converti en format binaire et interprétés par Python 3.7.1. Bien qu’il soit difficile de consacrer l’entièreté des ressources a l’interpréteur, afin de diminuer les perturbations par les autres processus de l’ordinateur, je m’assure de d’exécuter l’interpréteur en ligne de commande et de modifier sa priorité à *High* dans le gestionnaire de processus Windows. De plus, je m’assure de terminer tous les processus ayant une consommation élevé

**1.2 Programmation uniforme**

Chacun des algorithme sont programmé en utilisant la syntaxe Python 3.7.1. Chaque algorithme est invoqué de manière uniforme lors de la prise de mesure. En particulier on utilise une fermeture des paramètres *adj* et *order* sur la fonction *solve* qui est invoquée lors de l’appel *timeit.repeat.* Bien que cet indirection est un coût plus élevé que l’appel direct, cet procédure est utilisé pour chacun des algorithmes et donc ne biaise pas les résultats d’un algorithme en particulier. Finalement, chaque algorithme est programmé de façon itérative.

**1.3 La prise de mesure du temps de calcul a été réalisée correctement.**

Les mesures de temps de calculs sont effectuées à l’aide de la fonction *timeit.repeat* qui prend une fermeture en paramètre ainsi qu’un nombre de répétitions. Le nombre de répétitions permet de conserver uniquement le temps minimal de calcul parmi les répétitions ce qui reflète le mieux le temps de calcul optimal par le processeur, pour un algorithme donnée. De plus, la fonction *timeit.repeat* désactive le ramasse-miette python. La fréquence et la durer des activités du ramasse miette sont difficile à prédire. Donc du point de vue de la performance de l’analyse de performance ceci constitue principalement que du bruit. L’utilité de la mesure des activités du ramassée miette est moindre puisque nos algorithmes possèdent tout la même complexité mémoire.

Bien entendu, étant donnée la taille des échantillon requis par ce devoir, le nombre de répétions 10 ce qui ne permet pas d’obtenir des résultats assez fiables pour un jeu de donnée. Pour cette raison, je regarde les résultats moyen parmi plusieurs jeux de données afin de tirer une meilleur conclusion.

**1.4 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’à 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.

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

**3.1 Brute force**

**3.1.1 Complexité du temps de calcul**

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

Aux lignes 31-45, on observe le pire cas lorsqu’à chaque fois, on distingue le chemin candidat du plus petit chemin seulement grâce à la distance entre les deux dernières villes. Le coût est linéaire.

Il est possible de démontrer que la fonction *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 :

Puisqu’il existe exactement permutations entre de villes en omettant la ville de départ. Pour chacune de ces permutations nous sommes certains d’inclure le pire cas pour . Donc puisqu’aux lignes 54-56, borne on en déduit :

**3.1.2 Complexité mémoire**

En faisant abstraction de la matrice d’adjacence, on observe que puisque que seul le circuit candidat et le circuit calculé dans la fonction *shortest\_path* dépendent de *n,* il est facile de deviner la complexité mémoire de cet algorithme. Soit

**3.1.3 Mesure de performance**

**3.1.4 Mesure d’exactitude**

Étant donné la nature de cet algorithme nous savons que les résultats sont exacts. Ainsi je me sers de ces résultats afin de vérifier l’exactitude des heuristiques proposées dans le cadre de ce travail.

**3.2 Nearest neighbor**

**3.2.1 Complexité du temps de calcul**

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

En regardant les lignes 17-20, le contenu de la boucle *for* est effectué exactement fois puisque nous voulons traverser chacune des *n* villes et pour chacune de celles-ci nous vérifions parmi *n* villes soit que la ville soit déjà visitée ou nous vérifions que celle-ci a un cout minimal. Nous avons

et donc puisque cette boucle borne les autres opérations :

**3.2.2 Complexité mémoire**

En faisant abstraction de la matrice d’adjacence, pour cet algorithme nous seul le circuit calculée dépend de *n*. Donc :

**3.3 Nearest insertion**

**3.3.1 Complexité du temps de calcul**

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

Aux lignes 28-29, on observe que la boucle *for* effectue un nombre croissant d’itérations. Soit une itération de plus à chaque itération de la boucle *while* principale.

On observe cependant que cette boucle est effectuée un nombre décroissant de fois. Soit une fois de moins à chaque itération de la boucle *while* principale.

On observe donc qu’en relation avec le *while* principal:

Et étant donne ces simplification :

On déduit de cette expression que :

Aux lignes 38-44, on remarque que la boucle performe un nombre croissant d’itérations :

Donc :

Puisque les lignes 25-32 bornent les lignes 38-44, on note que

**3.3.2 Complexité mémoire**

En faisant abstraction de la matrice d’adjacence, pour cet algorithme nous seul le circuit calculée dépend de *n*. Donc :

**3.3.3 Tests de performance**

**3.4 Cheapest insertion**

**3.4.1 Complexité du temps de calcul**

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

Aux lignes 29-36, on observe que la boucle *for* effectue un nombre croissant d’itérations. Soit une itération de plus à chaque itération de la boucle *while* principale.

On observe comme pour *nearest\_insertion* que cette boucle est effectuée un nombre décroissant de fois. Soit une fois de moins à chaque itération de la boucle *while* principale. Nous avons donc :

Le cout de ces opérations borne le coût des autre opérations de l’algorithme. Donc :

Même si  et possède la même complexité, le coût est plus élevé lorsque on augmente *n* puisque la boucle centrale (:29-36) a un plus grand coût que celle de (25-32).

**3.4.2 Complexité mémoire**

En faisant abstraction de la matrice d’adjacence, pour cet algorithme nous seul le circuit calculée dépend de *n*. Donc :

**3.2.3 Tests de performance**

**3.5 Minimum spanning tree**

**3.5.1 Complexité du temps de calcul**

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

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

Avec *m* étant le nombre d’arrêtes et n le nombre de sommets du graphe. Cependant puisque dans ce problème le nombre maximal d’arrêtes notons que l’algorithme de prime coûte . En effet nous observons que les lignes 49-57 ont cette complexité. Donc :

Dans la fonction *dfs\_traversal*, aux lignes 23-29. 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 en temps logarithmique. Donc :

Puisque *calculate\_mst* borne *calculate\_hamiltonian\_path* :

**3.5.2 Complexité mémoire**

En faisant abstraction de la matrice d’adjacence, on observe que cet algorithme fait aussi usage d’un monceau contenant des objets de type *Node* lors de la construction de l’arbre sous-tendant de cout minimal. De plus on remarque l’usage d’une pile utilise dans le parcours *DFS*. Bien que ces représentations soient plus complexes, la taille maximale de chacune d’entre elle est borné par *n* qui est aussi la taille du circuit.

**3.5.3 Tests de performance**

**4 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).** |  |  |
|  |  |  |
|  |  |  |
|  |  |  |