Rapport - Projet Algorithmique et Structure de données
Michel Yoeung et Billy Ndihokubwayo (Groupe 2)
Ensimag 1A - 2018

I) Estimation a priori des coûts asymptotiques des fonctions codés

1) Itérateur avec hachage

On cherche d'abord à calculer approximativement le coût asymptotique de notre itérateur avec hachage. On considère ici n le nombre de points. En partant d'une précision (arbitraire) de 100.0, il faudra toujours un nombre constant (indépendant de n) d'itération pour qu'il n'y ait plus de collision
Pour tester s'il y a une collision dans chacune des 4 tables de hachage, notre fonction le fait en O(n) en itérant sur toutes les clés (et en regardant la longueur de chaque list qui est en O(1) en Python). Dans chacune des itérations (dont le nombre est indépendant de n), on va utiliser la fonction hash() qui consiste à hacher tous les points dans les 4 tables de hachage de la structure Hash. Dans cette fonction, on itère sur tous les points 4 fois, et, comme on suppose que chaque case de la table de hachage va contenir un "petit" nombre de points donc indépendants de n (car on va hacher de telle sorte qu'en augmentant la précision, on finira avec un point par case). Donc le coût de la fonction hash() est en O(n). Donc la construction de la liste des jeux de tables se fait en O(n) car on va utiliser la fonction hash() un nombre de fois constant (indépendant de n). Ensuite, l'inversion de la liste des jeux de tables se fait en O(1) (indépendant de n toujours).
On itère à présent pour proposer les segments : on commence par itérer un nombre constant de fois. A chaque itération, on va itérer sur toutes les cases des tables de hachage qui se fait en O(n) car on commence par la fin où le nombre de cases peut être approximé asymptotiquement par le nombre de points. Dans la dernière boucle d'itération où on va proposer toutes les combinaisons de segment, avec l'argument précédent, chaque case des tables de hachage vont comporter un nombre de points non dépendant de n car "très petit" donc pour chaque case, fabriquer toutes les combinaisons a un coût asymptotique en O(1).
Finalement, notre itérateur possède une complexité asymptotique en O(n) dans le cas général (moyen) et varie jusqu'à $O(n^2)$ dans le pire cas.
Pour le comparer avec l'itérateur quadratique, ce dernier est en $O(n^2 \log(n))$ car on fabrique environ n^2 combinaisons et le tri de ces n^2 combinaisons avec la fonction sorted() de Python (qui se base sur le quicksort ou le tri fusion) se fait donc en $O(n^2 \log(n))$.

2) Fonction reconnect()

Ici, on considère que n est le nombre d'arêtes du graphe. Mais comme dans le cas général, dans nos graphes, alors on peut faire l'hypothèse que chaque sommet contient un nombre constant (indépendant de n) d'arêtes car le graphe est généralement creux, alors on peut approximer le nombre d'arêtes du graphe par le nombre de sommets du graphe dans le calcul de nos coûts asymptotiques.
On commence par regrouper avec la structure Union-find les composantes connexes du graphe :
On itère sur tous les sommets du graphe donc en O(n) puis on ajoute chaque sommet dans la structure Union-find qui se fait en O(1). On reboucle sur tous les sommets déjà ajoutés donc le parcours devient du $O(n^2)$ et on utilise la fonction linked() qui permet de vérifier si deux sommets sont reliés ou non dans le graphe donc en coût constant O(1) car on itère sur un nombre d'arêtes indépendant de n à chaque sommet (suivant la première hypothèse). On utilise ensuite la méthode union() de la structure qui a une complexité en O(n). Cette méthode est appelée dans le cas où la fonction linked() renvoie True ce qui est le cas dans la majeure partie des cas.

On peut donc dire que juste avec la construction de la structure pour représenter les composantes connexes, on obtient une complexité en $O(n^3)$.
Concernant le parcours du graphe, on commence par itérer avec l'itérateur choisit (avec hash en O(n) approximativement ou avec le quadratique en $O(n^2 log(n))$). Pour chaque segment proposé par l'itérateur, on applique find() de la structure Union-find qui est en O(n) pour notre condition de liaison et si cette condition est satisfaite (très rare car en fonction du nombre de composantes connexes qui est dans la plupart des cas très faible par rapport à n) alors on va utiliser union() de la structure en O(n).
Finalement, avec l'itérateur quadratique on obtient une fonction reconnect() en $O(n^3 log(n))$ et avec l'itérateur hash, la fonction a un coût asymptotique en $O(n^3)$ quand même à cause de notre implémentation de la représentation des composantes connexes dans la première partie de la fonction (qui peut être donc optimisée)..

3) Fonction even_degrees()

Ici, on considère de même que n est le nombre d'arêtes et on garde la même hypothèse que précédemment (comme quoi ce nombre peut être approximé par le nombre de sommets dans le calcul des coûts asymptotiques).
On commence par parcourir tous les sommets en O(n) pour compter le nombre de sommets impairs (la vérification se fait en O(1)).
Ensuite on itère avec l'itérateur choisit comme précédemment. Pour chaque segment proposé par l'itérateur, si les deux sommets du segment sont impairs (le test se fait en O(1)), alors on ajoute l'arête (en O(1) dans le cas général avec la méthode list.append() de Python).
Donc la complexité de cette fonction dépend entièrement de celle de l'itérateur mais il faut aussi noter que l'on ne va pas itérer jusqu'au bout (et loin de là) pour faire en sorte que tous les sommets soient de degré pair.
Finalement, avec l'itérateur quadratique, la fonction even_degrees() est en $O(n^2 log(n))$ et avec l'itérateur hash, la complexité est en O(n) jusqu'à $O(n^2)$ dans le pire cas

4) Fonction eulerian_cycle()

La fonction eulerian_cycle() possède un bug d'itération et le coût asymptotique est de toute façon bien trop grand dans l'état actuel de l'implémentation (alors que le sujet attendait un coût assez faible). Nous n'avons donc pas réalisé de mesure de performance sur cette fonction dans la deuxième partie.

II) Mesures expérimentales des performances des fonctions codés

Toutes les mesures qui vont suivre ont été réalisé avec la commande shell "time" et la durée retenue est le temps réel (ligne "real").

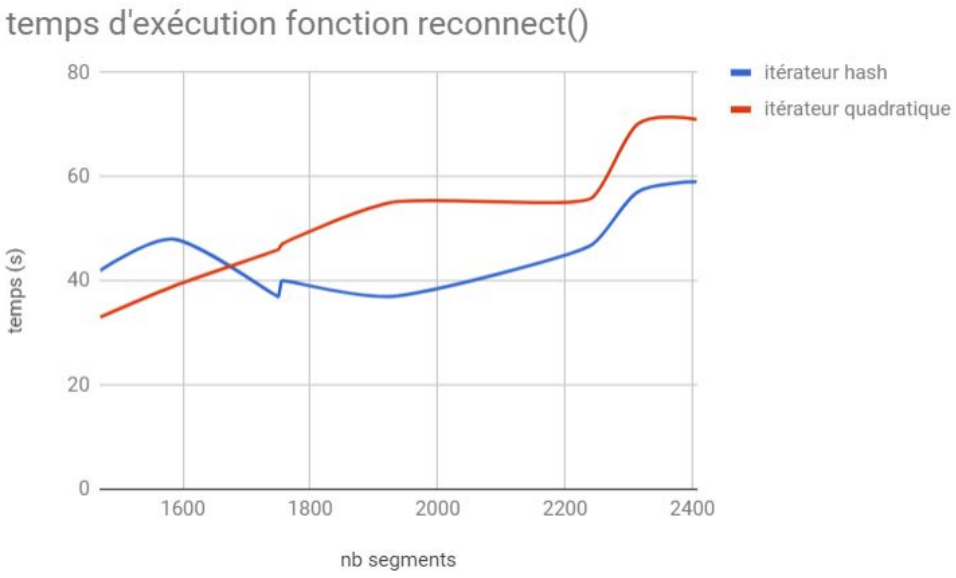
1) Fonction reconnect()

- En faisant varier le nombre de composantes connexes et en changeant l'itérateur

		ITERATEUR HASH	ITERATEUR QUADRATIQUE
nb composantes	nb segments	temps (s)	temps (s)
1	5394	104	100
4	5362	375	363
100	1825	38	60

En regardant les deux première lignes, si on ne fait pas varier (ou très peu) le nombre de segments, et en faisant surtout varier le nombre de composantes connexes; on voit que le temps d'exécution de la fonction reconnect() augmente, que ce soit avec l'itérateur hash, ou avec l'itérateur quadratique. Lorsqu'il n'y a qu'un seule composante connexe, on ne va pas itérer avec l'itérateur mais juste créer la structure Union-find mais la condition "linked()" va être satisfaite dans tous les cas donc on va appliquer la méthode union() en O(n) à chaque itération à coup sûr. Quand le nombre de composantes vaut 4, on ne va pas appliquer la méthode union() à chaque fois mais on va utiliser cette fois l'itérateur choisi pour parcourir ce qui explique le temps d'exécution plus élevé d'un certain facteur.
La troisième ligne servait à tester si le nombre de composantes avait un réel impact sur le temps d'exécution. D'après les résultat, on peut remarquer que non (même si le nombre de segments est bien inférieur). En effet, le nombre de composantes connexes va juste rallonger de très peu le temps passé dans l'itération avec l'itérateur car bien que la boucle est censée durer plus longtemps pour relier toutes les composantes, étant donné que la probabilité de liaison possible est très élevée au départ vu qu'il y a beaucoup de composantes connexes (et diminue ensuite), alors ce temps ajouté n'est pas très significatif. De plus, la condition "linked()" ne va pas toujours être satisfaite ce qui évitera de passer par union() en O(n) dans beaucoup de cas, donc ça réduit également le temps d'exécution.
Il y a donc une sorte de balance qui se créée et le nombre de composantes connexes n'a pas finalement trop d'impact sur le temps d'exécution de cette fonction.
Cependant, on constate une différence remarquable lorsqu'on utilise l'itérateur hash ou quadratique pour un même nombre de segments. On va mieux expliquer cela dans la partie suivante.

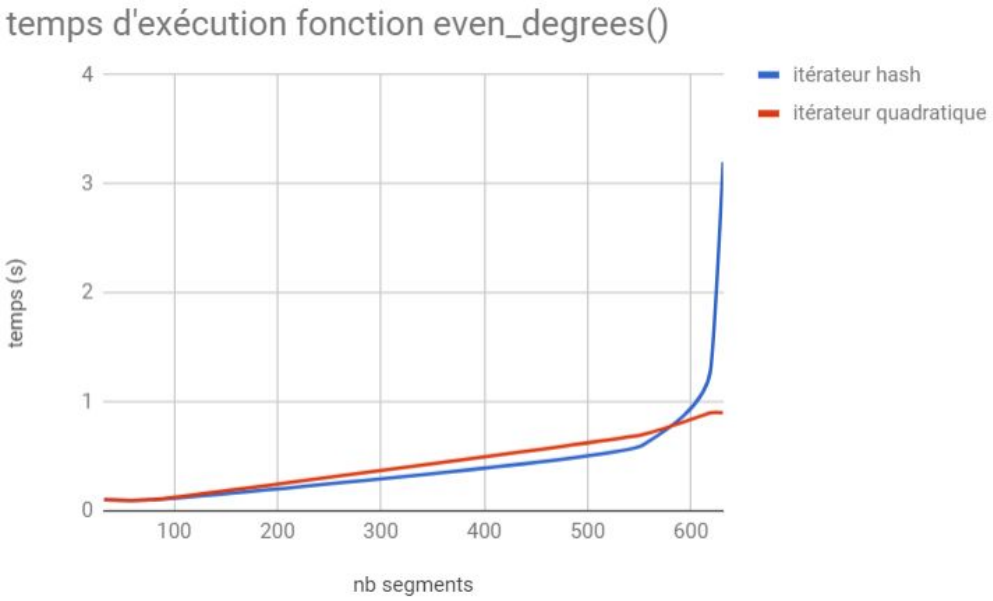
- **En faisant varier le nombre de segments et en changeant l'itérateur**



En faisant varier le nombre de segments, on constate que le temps d'exécution avec l'itérateur hash est plus faible qu'avec l'itérateur asymptotique. En effet, lorsqu'on analyse l'itérateur hash, ce dernier ne propose pas les segments dans un ordre parfait (le hachage est moins précis qu'un tri de toutes

les combinaisons possibles dans l'itérateur quadratique). Donc, la probabilité que la condition de liaison dans chaque itération est plus élevée avec l'itérateur hash car deux points de distance plus élevé a par exemple plus de chance d'être dans deux composantes connexes différentes dans le cas général (du moins dans nos graphes de test). Donc le nombre d'itérations avec l'itérateur sera inférieur avec l'itérateur hash vu que les composantes seront reliés plus rapidement et on quittera la boucle plus rapidement qu'avec l'itérateur quadratique.
On constate également que pour l'itérateur quadratique, il est cohérent que le temps d'exécution est croissant suivant le nombre de segments. Cependant, lorsqu'on regarde l'itérateur hash, on voit qu'au départ, le temps d'exécution diminue ce qui peut paraître incohérent. Cela pourrait peut-être s'expliquer par le fait que l'itérateur hash fournit un ordre non parfait et parfois "bruité" (dans les premiers segments proposés par l'itérateur, on a remarqué qu'il y a parfois quelques segments de longueurs étrangement bien plus élevés que la tendance).

2) Fonction even_degrees()



En faisant varier le nombre de segments, on constate que le temps d'exécution de la fonction even_degrees() est moins élevé lorsqu'on utilise l'itérateur hash pour un faible nombre de segments (entre 30 et 500). Cela pourrait paraître surprenant car on pourrait penser que, comme le nombre de segments est faible, le nombre de sommets est faible (dans le cas général dans nos fichiers de tests en tout cas), donc le fait de hacher tous les points plusieurs fois à chaque fois pourrait créer un surcoût non négligeable dans le cas d'un faible nombre de points. On pourrait donc penser dans ce cas que l'itérateur quadratique pourrait être plus performant dans ce cas (faible nombre de points). Mais dans la fonction even_degrees(), le coût dépend entièrement de l'itérateur (d'après la partie précédente, on a vu que c'était le coût asymptotique dominant), mais qu'il fallait aussi prendre en compte le fait qu'on ne va pas itérer jusqu'à la fin de l'itérateur. Il suffit juste que l'itérateur propose les segments proches (justement ce qui se passe dans les premières itérations) pour que l'ajout d'une nouvelle arête se fasse rapidement (dans le cas où la majeure partie des sommets impairs sont assez proches comme on peut le constater dans beaucoup de nos fichiers de tests).
Lorsque le nombre de segments dépasse 600, le temps d'exécution avec l'itérateur hash explose par rapport à celui avec l'itérateur quadratique, ce qui correspond bien à ce qu'on attendait puisque si on utilise l'itérateur quadratique, ce dernier va nous proposer les segments dans un ordre parfait contrairement à l'itérateur hash qui va non seulement devoir traiter les doublons, mais également proposer les segments dans un ordre non strict (car hachage).