Directed Feedback Vertex Set

Projet Programmation

L3 MIAGE 2022-2023

PEREIRA Guillaume

Pseudo Optil.io : guillaumehpereira_l3_miage

Introduction :

Le projet d'algorithme dans les graphes, que nous avions à réaliser, visait à résoudre un problème particulier dans les graphes : comment rendre un graphe acyclique en supprimant un minimum de sommets. Pour ce faire, j'ai développé deux algorithmes en Python qui prennent en entrée un graphe non-acyclique sous forme de dictionnaire ayant pour clé le numéro des sommets et pour valeur associée, la liste des sommets vers lequel il pointe et renvoie l'ensemble de sommets minimaux à supprimer pour le rendre acyclique. Il conviendra d'aborder dans un premier temps la méthode utilisant l'algorithme de Kosaraju avant de s'intéresser à la seconde méthode utilisant les arcs arrière. Enfin, il sera évoqué les différents procédés utilisés afin d'améliorer le résultat.

I – L'algorithme de Kosaraju

La première méthode, me permettant d'obtenir un graphe acyclique à laquelle j'ai pensé, a été d'utiliser l'algorithme de Kosaraju afin de déterminer les composantes fortement connexes d'un graphe. Par définition, il n'existe pas de cycle entre des sommets n'appartenant pas à la même composante fortement connexe. Ainsi, si je ne conserve uniquement qu'un seul sommet dans chacune d'entre d'elles, cela me permettra d'obtenir un graphe acyclique. L'algorithme de Kosaraju consiste à effectuer une recherche en profondeur sur le graphe et d'en établir un ordre suffixe. Puis d'effectuer une nouvelle recherche en profondeur suivant l'ordre suffixe inverse sur le graphe transposé. Chacune des arborescences obtenues constitue une composante fortement connexe. J'ai donc implémenté trois fonctions : *dfs_iterative*, *transpose_graph* et *find_scc* La complexité de chacune de ces fonctions est de O(V + E) avec V le nombre de sommets (vertices) et E le nombre d'arêtes (edges). Cependant, il était évident que cette méthode ne pouvait être optimale car elle retirait forcément des sommets qui n'intervenaient pas dans l'apparitions de cycles, ce qui m'a amené à réfléchir à une autre approche afin de résoudre ce problème.

II – La méthode des arcs arrière

En recherchant dans le cours, j'ai relevé une propriété indiquant qu'un graphe acyclique ne comportait pas d'arcs arrière. J'ai donc élaboré dans un premier temps un algorithme de recherche en profondeur itératif afin de relever tous les arcs arrières de complexité O(V + E). Puis afin de retirer le moins de sommets avec cette méthode, j'ai donc décidé d'écrire une fonction *most_common_vertex*, relevant le sommet apparaissant dans le plus d'arcs arrière de complexité O(V + E) et de supprimer ce dernier du graphe à l'aide de la fonction *LVR* (List Vertices Removed) de complexité O(B * (V+E)) dans le pire des cas avec B le nombre d'arcs arrière (back_edges), ce qui n'est pas optimal pour les très grands graphes. Cependant, cette méthode s'est beaucoup plus efficace sur les graphes de taille raisonnable. C'est pourquoi, après plusieurs tests, j'ai décidé de poser une condition à l'utilisation de cet algorithme au niveau du nombre d'arêtes présentes dans le graphe : si ce dernier en contient plus de 150 000 (information donnée par le deuxième nombre de la première ligne de chacune des instances de graphe), j'utilise l'algorithme de Kosaraju, dans le cas contraire celui faisant appel aux arcs arrière.

III – Optimisation des résultats

Dans un dernier temps, j'ai voulu réduire le nombre sommets supprimés en particulier dans la méthode des arcs arrière en réduisant le nombre de sommets dans le graphe. En effet, les sommets n'ayant 0 voisins entrants ou sortants ne pouvant donc pas intervenir dans la création d'un cycle peuvent donc être supprimés du graphe sans changer la liste des sommets à supprimer. Cependant, la suppression totale d'un sommet dans un graphe peut s'avérer d'une complexité très élevée si on souhaite faire disparaître le sommet supprimé de chacune des listes de voisins sortants auquel il appartient. C'est pourquoi, j'ai décidé de supprimer uniquement la clé dans le dictionnaire représentant le graphe. Cependant, lorsqu'un de mes algorithmes parcourent la liste des voisins sortants d'un sommet, j'ai rajouté une condition demandant de vérifier si le voisin appartient encore au graphe, ce qui est réalisé dans une complexité O(1) en Python, ce qui est donc plus avantageux dans notre cas. Par ailleurs, les sommets pointants sur eux-mêmes (s'il y en a) peuvent directement être ajouter dans la liste des sommets à supprimer et supprimer du graphe puisque leur simple existence provoque l'existence d'un cycle. Les algorithmes permettant la détection des sommets de tous ces types de sommets est également de complexité O(V + E) et leur suppression est O(1) puisque je supprime uniquement la clé dans le dictionnaire. Enfin, j'ai décidé dans un dernier temps de supprimer de mon graphe un à un les sommets que j'avais relevé dans une boucle *while(contains_cycles)* qui n'est autre qu'une recherche en profondeur retournant True si le graphe contient un cycle et False sinon. Cette méthode permet de supprimer un nombre inférieur ou égale, à la liste précédemment établie de sommets à supprimer et d'encore une fois réduire notre résultat.

Conclusion :

Ainsi, à l'aide de ces deux méthodes et à l'élaboration de quelques optimisations, j'ai pu obtenir un résultat valide, une liste de sommets à supprimer non triviale permettant de rendre un graphe acyclique pour donner suite à leur suppression fonctionnant sur toutes les instances publiques données dans le cadre du PACES 2023. Mes résultats sont cependant encore loin des réponses optimales probablement en raison des méthodes que j'ai utilisées, du peu de recherche locale réalisée (uniquement une insertion d'une partie de la liste de sommets à supprimer) mais aussi du langage de programmation Python qualifié de langage haut niveau, ne permettant d'optimiser pleinement le code. J'ai trouvé ce projet très enrichissant dans la mesure où il n'y avait pas d'instructions et il était donc nécéssaire de réfléchir à une solution par soi-même, de la tester, d'en comprendre les erreurs et de réitérer ainsi de suite afin d'obtenir un résultat satisfaisant.