Projet de Programation

Kamil Stos – Killian Pliszcak

Notation : **V** : nombre de sommets ; **E** : nombre d'arêtes ; **T** : nombre de trajets intéressants

Question 3 : Complexité de la fonction *get_path_with_power*

Complexité Temporelle : O(E)

On garde en mémoire les arêtes déjà visitées dans un dictionaire. Ainsi on ne fera au plus que E appels récursifs.
En effet, si par l'absurde emprunter une arête déjà visitée permettait de trouver un chemin admissible, alors lors d'un appel récursif précédent, on l'aurait déjà trouvé et la fonction se serait terminée.

- Vérifier qu'une valeur est une clé d'un dictionaire est en cout constant O(1)
- Ajouter une valeur à un dictionnaire est en cout constant O(1)
- Comparer deux valeurs auxquelles on a accès en coût constant O(1) est en coût constant

Le corps de la boucle "for n in neighbors" est donc constitué de l'appel récursif et d'opérations en cout constant O(1). Les opérations en coût constant O(1) vont être effectuées au plus O(E) fois au total. En effet, à chaque fois que l'on arrive sur un sommet, on effectue ces opérations en cout O(1) pour tous ces voisins, c'est-à-dire le nombre d'arêtes connectées à ce sommet.s

Complexité Spatiale : O(E+V)

On utilise un dictionnaire de taille en mémoire O(E). On effectue, grace au dictionnaire, au plus E appels récursifs. La longueur de la liste renvoyée est au plus V. La complexité spatiale est donc O(E+V)

Question 6 : Complexité de la fonction *min_power*

Complexité Temporelle : O(E log(E))

On itère sur les noeuds mais le nombre d'ajouts de power à la liste au total est 2×E car chaque arête est vue deux fois. La construction de cette liste prend donc O(E) opérations élémentaires.

Ensuite, transformer la liste en ensemble puis à nouveau en liste prend un nombre d'opérations de l'ordre de grandeur de la longueur de la liste, i.e. 2xE, cela prend donc O(E) opérations ici.

Trier la liste prend O(E log(E)) opérations élémentaires.

La recherche dichotomique prend un nombre d'itération de l'ordre de grandeur le log la longueur de la liste sur laquelle on l'effectue, ici O(log(E)) donc. Or dans chaque itération, on appelle la fonction *get_path_with_power* qui prend un nombre d'opérations élémentaires O(E). Le nombre d'opération élémentaires effectuées pour la recherche dichotomique est donc O(E log(E))

Au total, la complexité temporelle est donc : O(E + E log(E) + O(E) log(E)) = O(E log(E)). L'implémentation est sous-optimale puisque l'on sait que chaque arête est prise en compte deux fois, il suffirait de répertorier dans un dictionnaire les arêtes déjà croisée.

La vérification serait en temps constant O(1) donc cela ne changerait pas la complexité de l'algorithme mais cela éviterait d'avoir recours à la fonction *set* qui est couteuse et ainsi évitable. Nous avons pris le parti d'un code plus clair, puisque de toute manière cette version de min_power ne suffira pas pour les plus gros graphes et sera re-implémentée plus tard dans le projet.

On retrouve une complexité similaire à celle donnée par l'algorithme de Djikstra, mais celui ci prend une complexité spatiale O(V^2) tandis que cette version prend seulement :

Complexité Spatiale : O(E log(E))

La liste (que l'on trie en place dans l'implémentation), est de longueur E. Cependant chaque itération de *get_path_with_power* prend une place en mémoire de l'ordre de O(E). On fait un nombre O(log(E)) d'itérations de *get_path_with_power*. Le reste des variables prend une place constante en mémoire O(1).

Au total l'algorithme utilise une place en mémoire de l'ordre de O(E log(E))

(N.B. Il était possible d'éviter de dresser la liste *powers* et de directement procéder par dichotomie sur power jusqu'à obtenir une précision 10^{-16}. Nous avons préféré une solution théoriquement exacte puisque de toute manière cette implémentation ne sera pas assez efficace pour les gros graphes. Cette autre solution peut être plus rapide en pratique toutefois.)

```python
def min_power(self, src, dest):
        """
        Returns path from src to dest with minimal power and
        the associated power when there is one
        Returns None otherwise

        Parameters:
        -----------
        src: NodeType
            The source node
        dest: NodeType
            The destination node

        Output:
        --------
        tupple(list[Nodetype], float) | NoneType

        with
                path : list[Nodetype]
                    Sequence of nodes leading from src to dest through edges
                    whose power is less than that of the agent.

                power : float
                    Minimal power necessary to go from src to dest
        """
        #constructing list of distinct powers
        powers =[]
        for n in self.nodes:
            for e in self.graph[n]:
                powers.append(e[1])

        powers =list(set(powers))
        powers.sort() #in place
        # Dichotomic research
        i = 0
        j = len(powers) -1
        while i <j:
            if j ==i+1: # because of //2
                result_i =self.get_path_with_power(src, dest, powers[i])
                if result_i is None: #minimal power is at least powers[j]
                    i = j
                else:
                    return result_i
            #moving lower bound
            if self.get_path_with_power(src, dest, powers[(i+j)//2]) is None:
                i = (i+j)//2
            #moving upper bound
            else:
                j = (i+j)//2

        path =self.get_path_with_power(src, dest, powers[i])
        if not (path is None):
            return path, powers[i]
```

Question 11 : Montrer qu'un arbre possédant V sommets possède exactement V-1 arrêtes.

Un arbre est une structure récursive, il serait possible de procéder par induction structurelle. Cependant l'énoncé adopte le point de vue d'un graphe connexe acyclique, on prend donc la preuve selon ce point de vue.

Etape 1 : Pour un graphe connexe, $E \geq V - 1$

Pour V = 1, cette proposition est directement vraie.

Soit $n \geq 1$ et soit G un graphe connexe à $n+1$ noeuds. Supposons que tous les graphe connexe à n noeuds possèdent au moins $n-1$ arêtes.

S'il existe un noeud de degré 1 dans G, en le supprimant avec l'arête qui le lie au graphe on obtient un graphe connexe à n noeuds. Ce sous graphe possède alors par hypothèse au moins $n-1$ arêtes et donc G en possédait au moins n. Sinon tous les noeuds sont de degré au moins deux. Alors $2n = \sum_{n \in G} 2 \leq \sum_{n \in G} deg(n) = 2 * E$ et donc G possède bien au moins n arêtes.

Etape 2 : Pour un graphe acyclique donc $E \leq V - 1$

.

***Lemme* :** Si tous les noeuds d'un graphe sont de degré au moins 2, alors il existe un cycle

Dém : On note $v_1, ..., v_n$ les sommets d'un tel graphe. On construit un chemin récursivement de la manière suivante : $v_{i_0} = v_1$; $v_{i_{k+1}}$ est un voisin de v_{i_k} ; $v_{i_{k+1}} \in \{v_{i_0}, ..., v_{i_k}\}$ jusqu'à ce que ne soit plus possible. Comme le graphe est fini, ce processus s'arrête, on note N le nombre étapes de construction. v_{i_N} possède alors un voisin dans $\{v_{i_0}, ..., v_{i_{N-1}}\}$ sinon on pourrait continuer à construire la suite (on suppose $N \geq 2$ le cas non trivial). On a ainsi construit un cycle.

On prouve maintenant la propriété par récurrence.
Un graphe acyclique à 1 noeud a bien au plus 0 arêtes.

Soit G un graphe acyclique de taille $n+1 \geq 2$, on suppose que tout graphe acyclique de taille n possède au plus $n-1$ arêtes. Par le lemme, G possède un noeud de degré 0 ou 1. En le supprimant ainsi que l'éventuelle arête connectée à lui, on obtient un graphe acyclique de taille n. On applique l'hypothèse de récurrence à ce graphe qui possède alors au plus $n-1$ arête, on déduit ensuite que G possédait au plus n arêtes.

Conclusion :
Un arbre de taille V est un graphe connexe acyclique et possède donc exactement $V - 1$ arêtes.

Question 12 : Complexité de la fonction *kruskal*

Complexité Temporelle : O(V + E log(E))

Nous n'avons pas encore choisi d'orienter le graphe pour avoir un arbre avec une relation de parenté, la conversion sera de toute manière faisable en temps O(V) à l'issue de cette fonction (en parcourant le graphe en largeur). On construit d'abord une liste des arêtes sans répétition grace à un dictionnaire qui garde en mémoire les arêtes déjà rajoutées.
Chaque arête est vue deux fois, mais puisqu'elle est déjà dans le dictionaire la deuxième fois, elle n'est pas ajoutée à la liste.
Ce processus a un coût en opération élémentaire de l'ordre de $O(2 * E + E) = O(E)$.

Trier la liste sur power prend O(V + E log(E)) opérations élémentaires classiquement.

Ensuite on construit le graphe. On itère sur la liste des arêtes. La vérification :

```
if not (connected[node_b][0] ==connected[node_a][0]):
```

s'effectue en temps constant. Ce test suffit car les composantes connexes sont disjointes, et deux éléments dans la même composante renvoient vers un même pointeur de liste. Les listes sont de plus toujours non-vides.

La section de code

```
for node_c in connected[node_b]: # none of the nodes connected to b were connected to a
                    connected[node_a].append(node_c)
                    connected[node_c] =connected[node_a]
```

n'est pas constante pour chaque itération, mais puisque ce code ne s'effectue plus (grace au test préalable) quand il n'y a plus qu'une composante connexe, on déduit qu'au total l'execution de cette partie du code regroupée pour toutes les itérations sur *edges* ne va représenter que O(V) opérations élémentaires.

Complexité Spatiale : O(E+V)

On utilise successivement un dictionnaire de taille E, une liste de taille E, un tri en place (O(1)), tout ceci représente un espace mémoire de l'ordre de O(E). Enfin, on utilise un dictionaire *connected* de taille O(V) ; car il y a V clefs et la somme des longueur des listes disctinces contenues dedans est également constante égale à V (invariant de boucle immédiat).

```python
def kruskal(self):
        """
        Returns the minimal covering tree of the graph.
        Construction uses Kruskal's algorithm.

        Output
        ---------
        G : Graph
        """

        # constructing list of (power, edge) without repetition
        edges_seen ={} #to check repetition
        edges =[]
        for node_a in self.nodes:
            for edge in self.graph[node_a]:
                node_b, p, d =edge
                if not ((node_a, node_b) in edges_seen):
                    edges_seen[(node_a, node_b)] =True
                    edges_seen[(node_b, node_a)] =True # to not add it again when on node_b
                    edges.append((node_a, node_b, p, d))
        # sorting in place on powers
        edges.sort(key= lambda a :a[2])

        #constructing covering tree
        G = Graph(self.nodes)
        connected ={n: [n] for n in self.nodes} #to identify accessible nodes from n
        # using dictionary for O(1) check if two nodes are connected
        # loop invariant: connected[n] contains the connected component of node n
        for edge in edges:
            node_a, node_b, p, d =edge
            if not (connected[node_b][0] ==connected[node_a][0]):
                G.add_edge(node_a, node_b, p, d)
                for node_c in connected[node_b]: # none of the nodes connected to b were
                                                    connected to a
                    connected[node_a].append(node_c)
                    connected[node_c] =connected[node_a] # pointers
                    # changing connected[a] will update it automatically for all nodes in the
                                                        connected component
            if G.nb_edges() ==G.nb_nodes()-1:#optionnal: trees of size V have exactly V-1 edges
                break
        return G
```

Question 13 : Test de la fonction

Pour tester l'algorithme implémenter, on considère des graphs de petite taille que sont "network.00" et "network.01", mais on teste aussi "network.10". On remarque alors que l'ensemble de ces 3 tests s'effectuent en seulement 1.471s ce qui, considérant que le dernier graph comporte 200 000 noeuds, est appréciable et vient valider les résultats de la question précédente.

Question 15 : Complexité de la fonction *min_power_tree*

Complexité Temporelle : O(V)

On considère que le graph passé en argument a déjà été converti en arbre couvrant minimal par la fonction kruskal.
La fonction *min_power_tree* est ainsi simplifiée.
On parcourt en profondeur récursivement le graphe, on garde en mémoire le pouvoir minimal du chemin en le passant dans les arguments.
Grace au dictionnaire *seen*, on ne passe qu'une fois par chaque noeuds, et donc par chaque arête.
Cela fait une complexité O(V).

Problème : Ce n'est pas assez bien pour de grands graphes avec beaucoup de routes !
Par exemple, pour V = 100 000 et 100 000 routes, cela fait un ordre de grandeur de 10 milliards d'opération élémentaire.
Comme ce n'est qu'un ordre de grandeur et que cela peut être plusieurs fois cette quantité dans l'implémentation, en réalité on peut arriver facilement plusieurs dizaines de milliard d'opérations élémentaires.

Si l'on considère que Python peut effectuer environ 1 million d'opération élémentaires par seconde approximativement, alors l'executation prendrait une trentaine de minutes.
On retrouve ce résultat sur network.2 !

En réalité, la structure d'arbre permet de faire bien mieux.
Si l'arbre est bien équilibré, par un preprocessing des ancêtres, on peut arriver à une complexité O(log(V)), et alors l'execution de *min_power_tree_optimised* prendrait un temps $T \times log(V)$. Pour les graphes proposés dans input, cela donnerait une implémentation de l'ordre de la minute, comme le suggère la séance 3.

```python
def min_power_tree(self, src, dest):
        """
        Returns, if there is one, the only path from src to dest in a tree and the minimal
        power necessary to cross it.
        By construction of the minimal covering tree, the power is
        the smallest one needed to go from src to dest
        It is assumed the function is being applied on a minimal covering tree,
        e.g. the output of the kruskal function


        Parameters:
        -----------
        src: NodeType
            First end (node) of the edge
        dest: NodeType
            Second end (node) of the edge

        Output
        ----------
        tupple(list[NodeType], float) | NoneType
        path :
            Sequence of nodes leading from src to dest through edges
            whose power is less than that of the agent.

        power : float
            Minimal power necessary to go from src to dest
        """
        # Recursive Deep First Search
        # unlike with a graph, we can just check nodes

        seen ={} # node : True
        def rec_path(position, min_p): #keeping track of the minimal power on the path
            if position ==src:
                return [src], 0
            #checking neighbors
            for edge in self.graph[position]:
                node_b, p, _ =edge
                if node_b not in seen:
                    seen[node_b] =True
                    result =rec_path(node_b, min(p, min_p))
                    if result is not None:
                            path, p_min =result # p_min <= p necessary
                            path.append(position)
                            return path, p_min
        return rec_path(dest, 0)
```