TD Algo2 – session 7 – Algorithmes élémentaires pour les graphes

21 mars 2025

Objectifs d'apprentissage :
— analyser l'impact des représentations des graphes sur des opérations algorithmiques simples ;
— manipuler les cas particuliers et les propriétés des algorithmes de parcours ;
— concevoir des algorithmes s'appuyant sur ces représentations et parcours ;

Les 5 premiers exercices sont essentiels. Plus d'exercices sur `https://algs4.cs.princeton.edu/41graph/`.

Exercice 1 : Degré

Étant donnée une représentation par listes d'adjacences d'un graphe orienté connexe (une seule composante connexe), quelle la complexité en temps pour calculer le degré sortant de tous les sommets ? Même question avec la représentation par matrice d'adjacences. Et pour calculer les degrés entrants ?

On obtient la taille d'une liste d'adjacences en $O(|G.adj[v]|)$ sur un sommet v. Il faut donc un temps $O(E)$ pour parcourir tous les sommets et calculer le degré sortant de tous les sommets. En effet, dans un graphe connexe $|E| = O(V)$ et donc on simplifie $O(V + E)$ en $O(E)$.

DEGRÉ-SORTANT(G)

pour $v \in G.V$ **faire**
  $v.ds \leftarrow 0$
  pour $u \in G.adj[v]$ **faire**
    $v.ds \leftarrow v.ds + 1$

Le degré entrant d'un sommet v nécessite de parcourir chaque autre liste d'adjacences, ce qui prend un temps $O(E)$. Un algorithme naïf sera donc en $O(VE)$ mais on peut optimiser pour rester en $O(E)$.

DEGRÉ-ENTRANT(G)

pour $v \in G.V$ **faire**
  $v.de \leftarrow 0$
pour $v \in G.V$ **faire**
  pour $u \in G.adj[v]$ **faire**
    $u.de \leftarrow u.de + 1$

Avec la représentation par matrice d'adjacences, on calcule soit la somme des lignes, soit la somme des colonnes, ce qui prend un temps $O(V^2)$ dans les 2 cas.

Exercice 2 : Carré d'un graphe

Le carré d'un graphe orienté $G = (V, E)$ est le graphe $G^2 = (V, E^2)$ tel que $(u, w) \in V^2$ si et seulement s'il existe un $v \in V$, tel que $(u, v) \in V$ et $(v, w) \in V$. Autrement dit, G^2 possède un arc entre u et w chaque fois que G contient un chemin de longueur deux exactement entre u et w. Décrire des algorithmes efficaces permettant de calculer G^2 à partir de G, quand G est représenté par listes d'adjacences, puis par matrice d'adjacences. Analyser le temps d'exécution de vos algorithmes.

Listes d'adjacences : pour chaque sommet, on parcourt les sommets de la liste d'adjacences et pour chacun, on récupère tous les sommets de leurs listes d'adjacences qu'il faut alors fusionner pour éliminer les redondances. On peut borner grossièrement la complexité en $O(E^2)$ en ignorant les fusions : pour chaque arête, on aboutit sur un sommet et on considère toutes ses arêtes. La fusion peut se faire en temps linéaire $O(E)$ si on tri toutes les listes d'adjacences initialement, ce qui prend un temps $O(E \log E)$.

CARRÉ-GRAPHE(G)

pour $v \in G.V$ **faire**
  Trier la liste d'adjacence $G.adj[v]$
  $G'.adj[v] \leftarrow NIL$
pour $v \in G.V$ **faire**
  pour $u \in G.adj[v]$ **faire**
    $G'.adj[v] \leftarrow G'.adj[v] \cup G.adj[u]$
retourner G'

Matrice d'adjacences : on réalise le produit de la matrice d'adjacences avec elle-même et on transforme toutes les valeurs positives en 1. L'algorithme de base pour la produit matriciel est en temps $O(V^3)$ et on peut le réduire un peu avec un meilleur algorithme.

Pour un graphe dense, la seconde représentation sera à préférer.

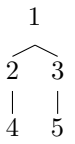
Exercice 3 : Arbre de parcours en largeur

On rappelle qu'un arbre de parcours en largeur d'un graphe orienté connexe $G = (V, E)$ est l'arbre obtenu par un parcours en largeur sur un sommet origine $s \in V$: c'est un sous-ensemble des arcs $E_\pi \subseteq E$ tels que, pour chaque sommet $v \in V$, le chemin unique dans E_π de s à v soit un plus court chemin dans G.

Donner un exemple de graphe ayant un arbre de parcours qui ne puisse être obtenu en exécutant un parcours en largeur sur G, quel que soit l'ordre des sommets dans chaque liste d'adjacences.

	1	2	3	4	5
1	0	1	1	0	0
2	0	0	0	1	1
3	0	0	0	1	1
4	0	0	0	0	0
5	0	0	0	0	0

En commençant par $s = 1$, aucun parcours en largeur ne pourra obtenir l'arbre :



Exercice 4 : Labyrinthe

Soit $G = (V, E)$ un graphe non orienté et connexe. Donner un algorithme en $O(E)$ pour calculer un chemin dans G qui traverse chaque arête dans E exactement une fois dans chaque direction. Décrire comment on peut trouver son chemin dans un labyrinthe si on dispose d'une grande quantité de pièces de monnaie.

Le labyrinthe se modélise avec un sommet par intersection et une arête pour un chemin. On réalise un parcours en profondeur sur le graphe en faisant marche-arrière à chaque étape. Lorsqu'un sommet a déjà été exploré, on ne l'explore pas, comme dans un parcours en profondeur. Il faut un code : 1 pièce face à un chemin exploré, 2 pièces face à un chemin d'où l'on vient. On arrive à un sommet soit en l'explorant, soit en ayant fait demi-tour. Lorsqu'on explore un sommet :

— si le sommet ne possède aucune pièce, on marque le chemin d'où l'on vient avec 2 pièces : on choisit un autre chemin en le marquant avec 1 pièce et s'il n'y a pas d'autre chemin,

on fait demi-tour ;
— si le sommet possède des pièces, c'est qu'il est en cours d'exploration et on marque le chemin d'où l'on vient avec 1 pièce, puis on fait demi-tour.

Lorsqu'on fait demi-tour : s'il y a un chemin sans pièce, on l'explore ; sinon, on continue à faire demi-tour en prenant le chemin ayant 2 pièces.

Exercice 5 : Trou noir

Quand on utilise la représentation par matrice d'adjacences, la plupart des algorithmes de graphes s'exécutent en $\Theta(V^2)$, mais il y a des exceptions. Montrer qu'il est possible de déterminer en $O(V)$ si un graphe orienté contient un trou noir, c'est-à-dire un sommet de degré entrant $V-1$ et de degré sortant 0 si on utilise une représentation par matrice d'adjacences.

L'idée est de partir du premier sommet et de parcourir n'importe quelle arête allant vers un sommet d'indice supérieur. On descend alors dans le graphe potentiellement jusqu'à un trou noir. Lorsque la descente est finie, il faut alors vérifier si le dernier sommet parcouru est un trou noir ou pas.

CHERCHER-TROU-NOIR(A)

$s \leftarrow 1$
pour $i = 2$ **jusqu'à** $A.n$ **faire**
  si $a_{si} = 1$ **alors**
    $s \leftarrow i$
pour $i = 1$ **jusqu'à** $A.n$ **faire**
  si $a_{si} = 1$ **ou** $a_{is} = 0$ **et** $i \neq s$ **alors**
    retourner NIL
retourner s

Exercice 6 : Table de hachage

Supposons qu'au lieu d'une liste chaînée, chaque entrée du tableau $G.adj[u]$ soit une table de hachage contenant les sommets v pour lesquels $(u, v) \in E$, les collisions étant résolues par chaînage. Dans l'hypothèse d'un hachage indépendant uniforme, si toutes les recherches d'arêtes ont la même probabilité, quel est le temps attendu pour déterminer si une arête se trouve dans le graphe ? Quels sont les inconvénients de ce système ? Proposer une autre structure de données pour chaque liste d'arêtes qui résout ces problèmes. Votre solution présente-t-elle des inconvénients par rapport à la table de hachage ?

On aurait en moyenne un accès directe à la table de hachage d'un des sommets de l'arête puis une requête en $\Theta(1)$. L'inconvénient de ce système est l'espace mémoire utilisé et la possibilité d'une collision.

Pour éviter les collisions, on pourrait utiliser un tableau indicé par les sommets, mais cela reviendrait à une représentation par matrice d'adjacences et cela prendrait encore plus d'espace.

Pour consommer moins d'espace, on pourrait utiliser un arbre binaire de recherche équilibré pour le même espace que la liste d'adjacences. La requête précédente prendrait alors un temps $O(|adj[v]| \log |adj[v]|)$, ce qui serait alors plus coûteux en temps qu'une table de hachage, mais moins en espace.

Exercice 7 : Arbre de parcours en profondeur

Le docteur Alex Ample souhaite trouver des graphes qui montrent que les conjectures suivantes sont fausses. La première conjecture affirme que s'il existe un chemin entre u et v dans un graphe orienté G, et si $u.d < v.d$ lors d'un parcours en profondeur de G, alors v est un descendant de u dans la forêt de parcours en profondeur obtenue. La seconde conjecture indique que s'il existe un chemin entre u et v dans un graphe orienté G, alors tout parcours en profondeur donne forcément $v.d \leq u.f$. Alex Ample se rend compte qu'un même graphe avec 3 sommets et avec le même parcours peut servir dans les deux cas. Lequel ?

L'exemple est le même dans les 2 cas : $V = \{1, 2, 3\}$ et $E = \{(1, 2), (2, 1), (1, 3)\}$. On commence par 1, puis 2. On revient sur 1 et on parcourt enfin 3. Il existe un chemin entre 2 et 3 et on a $2.d = 2$ et $3.d = 4$. Hors, 3 n'est pas un descendant de 2 dans l'arbre de parcours en profondeur. Par ailleurs, $3.d = 4$ et $2.f = 3$.

Exercice 8 : Catcheurs

Il existe deux types de catcheurs : les "bons" et les "méchants". Entre deux catcheurs donnés, il peut ou non y avoir une rivalité. Supposer que l'on ait n catcheurs et une liste de r paires de catcheurs telle qu'il y ait rivalité entre les deux membres de chaque paire. Donner un algorithme à temps $O(n + r)$ qui détermine s'il est possible de désigner certains catcheurs comme étant les bons et les autres comme étant les méchants, de telle sorte que chaque rivalité oppose un bon à un méchant. S'il est possible de faire ce genre de classification, votre algorithme doit la calculer.

Il s'agit d'une extension du parcours en largeur qui alterne les classes des catcheurs en parcourant les rivalités. En cas d'incompatibilité, une erreur est déclenchée.

CLASSIFIER-CATCHEURS(G)

```
pour v ∈ G.V faire
  v.classe = −1
pour v ∈ G.V faire
  si v.classe = −1 alors
    v.classe = 0
    Q ← ∅
    ENFILE(Q, v)
    tant que Q ≠ ∅ alors
      u ← DEFILE(Q)
      pour w ∈ G.adj[u] faire
        si w.classe = −1 alors
          w.classe = 1 − u.classe
          ENFILE(w)
        si w.classe ≠ 1 − u.classe alors
          erreur "impossible de trouver une solution"
```

Exercice 9 : Arbre binaire

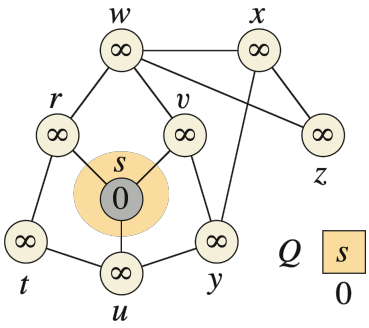
Donner une représentation par listes d'adjacences pour un arbre binaire complet à 7 sommets. Donner la représentation équivalente par matrice d'adjacences. On suppose que les sommets sont numérotés de 1 à 7 comme dans un tas binaire.

On suppose un arbre non orienté. Pour les listes d'adjagences, on a $G.adj[1] = \{2, 3\}$, $G.adj[2] = \{1, 4, 5\}$, $G.adj[3] = \{1, 6, 7\}$, $G.adj[4] = \{2\}$, $G.adj[5] = \{2\}$, $G.adj[6] = \{3\}$, $G.adj[7] = \{3\}$.

	1	2	3	4	5	6	7
1	0	1	1	0	0	0	0
2	1	0	0	1	1	0	0
3	1	0	0	0	0	1	1
4	0	1	0	0	0	0	0
5	0	1	0	0	0	0	0
6	0	0	1	0	0	0	0
7	0	0	1	0	0	0	0

Exercice 10 : Parcours en largeur

Donner les valeurs d et π qui résultent d'un parcours en largeur du graphe non orienté de la figure suivante, en prenant pour origine le sommet u.



Exercice 11 : Diamètre

Le diamètre d'un arbre $T = (V, E)$ est donné par $\max_{u,v \in V} d(u, v)$. Autrement dit, le diamètre est la plus grande de toutes les distances de plus court chemin dans l'arbre. Donner un algorithme efficace permettant de calculer le diamètre d'un arbre, et analyser son temps d'exécution.

On lance un parcours en largeur sur T à partir d'un nœud quelconque. On sélectionne ensuite un des nœuds avec la plus grande valeur $v.d$. On relance un parcours en largeur sur T à partir de ce nœud. Le diamètre est donné par la valeur maximale $v.d$ et obtenu en temps $O(V + E)$.

Exercice 12 : Indépendance

Montrer que, dans un parcours en largeur, la valeur $u.d$ affectée à un sommet u est indépendante de l'ordre dans lequel les sommets apparaissent dans chaque liste d'adjacences.