Leçon 7 : Accessibilité et chemins dans un graphe. Applications

Pré-requis : graphe, parcours

I - Définition

Définition 1 : Chemin

Dans un graphe orienté G (resp. non orienté), on appelle chemin de longueur λ, une suite de $(\lambda+1)$ sommets $(s_0, s_1, \dots, s_\lambda)$ tel que pour $i \in [\![0, \lambda-1]\!]$, (s_i, s_{i+1}) est un arc (resp. une arête).

Remarque 2 : Par convention on dit qu'il y a un chemin de longueur 0 de tout sommet vers lui même.

Remarque 3 : Dans un graphe non orienté, les chemins sont aussi appelés chaînes.

Application 4 : Dans un graphe de flot de contrôle, le critère tous les chemins consiste à trouver des tests qui font tous les chemins possibles du graphe.

Définition 5 : Un chemin est dit élémentaire s'il ne contient pas plusieurs fois le même sommet.

Définition 6 : Dans un graphe orienté (resp. non orienté), un chemin $(s_0, \dots, s_\lambda)$ avec $s_0 = s_\lambda$ est appelé un circuit (resp. un cycle)

Exemple 7 : On dit qu'un circuit est hamiltonien s'il passe une et une seule fois par tous les sommets. Décider de l'existence d'un tel circuit est NP-complet (i.e. dur).

Exercice 8 : On dit qu'un graphe non orienté est eulérien s'il existe un cycle passant par chaque arête exactement une fois. Montrer qu'un graphe est eulérien si et seulement si tous ses sommets sont de degré pair.

Définition 9 : Etant donné un graphe G (orienté ou non) et deux sommets s et t, on dit que t est accessible depuis s s'il existe un chemin de s à t dans G.

II - Accessibilité

Déterminer l'accessibilité peut se faire en temps linéaire par un parcours.

II-1) Tri topologique

Définition 10 : Etant donné un graphe orienté $G = (S, A)$, on dit que $\sigma : S \to [\![1, |S|]\!]$ est un tri topologique si $(s, t) \in A \Rightarrow \sigma(s) \leq \sigma(t)$

Corollaire 11 : Si σ est un tri topologique sur G, s'il existe un chemin de s à t, alors $\sigma(s) \leq \sigma(t)$

Proposition 12 : G est acyclique si et seulement si il existe un tri topologique sur G.

Preuve 13 : Pour le sens direct, on montre par l'absurde par récurrence qu'un graphe acyclique a un sommet source.
On montre alors la propriété par récurrence sur $|S|$.

Algorithme 14 : Construction d'un tri topologique

```
Créer une pile vide
Faire un parcours en profondeur postfixe de G en appliquant la fonction empile
Renvoyer la pile.
```

Application 15 : ordonnancement de tâches :

Définition 16 : Soit $T = (t_1, \dots, t_n)$ un ensemble de tâches d'un ordinateur et $C = T^2$ un ensemble de contraintes ($(t_i, t_j) \in C$ veut dire que t_i doit être fait avant t_j). Un ordonnancement de ces tâches est un ordre d'exécution de ces tâches.

Propriété 17 : On peut trouver un ordonnancement en temps linéaire

Preuve 18 : On prend l'ordre d'un tri topologique du graphe (T, C).
(ce qui est linéaire par l'algorithme 14).

II-2) Connexité

Propriété 19 : La relation « il existe un chemin de s à t et de t à s » pour $t, s \in S$, est une relation d'équivalence (avec $G = (S, A)$ orienté ou non).

Exercice 20: Montrer que le graphe quotienté par cette relation est acyclique.

Définition 21: Dans un graphe non orienté (resp. orienté), les classes d'équivalence de cette relation sont appelés composantes connexes (resp. fortement connexe).

S'il n'y a qu'une seul classe, le graphe est dit connexe (resp. fortement connexe)

Exercice 22: Dans un graphe connexe, deux chemins élémentaires maximaux ont un noeud en commun

Propriété 23: Les composantes connexes (resp. fortement connexes) sont les sous-graphes connexes (resp. fortement connexe) maximaux.

Algorithme 24: Calcul des composantes connexes

Tant qu'il y a un sommet non visité:
- Créer une nouvelle composante connexe
- Parcourir le graphe des sommets non visités depuis un sommet v non visité en ajoutant les sommets que l'on parcours à la composante

Pour les composantes connexes il faudra plus qu'un simple parcours.

Algorithme 25: Algorithme de Kosaraju

Appliquer l'algorithme 14 pour obtenir une pile p (même si le graphe a des cycles)

Prendre G^T le graphe transposé de G (i.e. (u,v) devient (v,u))

Tant que p est non vide
- u = depiler(p)
- Si u n'a pas été visité
  - Créer une nouvelle composante fortement connexe
  - Visiter G^T depuis u en ajoutant les sommets visités non déjà attribués

Idée 26: C'est comme l'algorithme 24 mais en commençant le ~~deu~~ parcours (qui est le deuxième) par les "bons" sommets (ceux en bout de chaînes)

Propriété 27: Cet algorithme construit les composantes fortement connexes en temps linéaire

Définition 28: Le problème 2-SAT est le problème de savoir, étant donnée n variables $x_1, \dots, x_n$ et p clauses $C_1, \dots, C_p$ de taille 2 sur $x_1, \dots, x_n$, si $\varphi = \bigwedge_{i=1}^{p} C_i$ est satisfiable.

Propriété 29: 2-SAT est réductible à un problème d'accessibilité, décidable en temps linéaire grâce à l'algorithme de Kosaraju.

Développement 1: Résolution en temps linéaire de 2-SAT

III - Plus court chemin

Définition 30: Un graphe pondéré est un graphe $G = (S, A)$ muni d'une fonction de poids $w : A \to \mathbb{Z}$

Le poids d'un chemin est la somme des poids des arêtes qui le compose.

Remarque 31: On peut étendre w à S^2 en posant $w(u,v) = +\infty$ si $(u,v) \notin A$

Définition 32: Dans un graphe G, un plus court chemin (pcc) de u à v ($u, v \in S$) est un chemin de poids minimal de u à v.

Remarque 33: Si les poids sont unitaires, on peut trouver le pcc entre deux sommets u et v en faisant un parcours en largeur depuis u.

III-1) D'un sommet à tous les autres

Lorsque la fonction est à valeurs dans $\mathbb{N}$, on peut utiliser l'algorithme de Dijkstra.

Algorithme 34: Algorithme de Dijkstra (entrée : (S, V, s, w))

source — tableau de liste d'adjacence

```
distance ← tableau de taille |S| initialisé à +∞
distance[s] ← 0
F ← FilePriorité()
Insérer(F, s, 0)
Tant que F n'est pas vide :
    u, d_u ← ExtraireMin(F)
    Pour v ∈ V[u]          # pour v voisin de u
        d ← d_u + w(u, v)
        Si d < distance[v]
            Si distance[v] = +∞
                Insérer(F, v, d)
            Sinon
                Diminuer(F, v, d)
Renvoyer distance
```

Complexité 35: Si comme structure de file de priorité on utilise un tas-min, on obtient une complexité en $O(|A| \log |S|)$

Remarque 36: Si la structure d'entrée est une matrice d'adjacence, on peut ne pas utiliser de structures spécifiques pour F (simplement rechercher le minimum de distance non visités) en $O(|S|^2)$

Remarque 37: Lorsque la fonction est à valeurs dans $\mathbb{Z}$, on peut utiliser l'algorithme de Bellman Ford

Application 38: Détection des cycles de poids négatifs grâce à la convergence de l'algorithme de Bellman-Ford

III-2) De tous les sommets à tous les sommets

Lorsque le cycle ne contient aucun cycle de poids négatif, on peut utiliser l'algorithme de Floyd-Warshall utilisant de la programmation dynamique : (on suppose $S = [\![1, n]\!]$)

* Sous-problèmes : Notons $d^{(k)}(i,j)$ la distance du pcc de i à j avec pour seuls sommets intermédiaires des sommets de $[\![1, k]\!]$
* Relation de récurrence :

$$d^{(0)}(i,j) = w(i,j)$$
$$d^{(k+1)}(i,j) = \min\left(d^{(k)}(i,j),\ d^{(k)}(i,k) + d^{(k)}(k,j)\right)$$

* Résolution :
  On calcule pour chaque k en croissant, pour tout $(i,j) \in S^2$, $d^{(k)}(i,j)$
  On renvoie $d^{(n)}$

Complexité 39: Cet algorithme est en $O(n^3)$

Remarque 40: Si les poids sont positifs, on peut appliquer Dijkstra à chaque sommet en $O(n \times |A| \log n)$

Remarque 41: Cet algorithme utilise une matrice d'adjacence.

Algorithme 42: Si on a des listes d'adjacences, on peut utiliser l'algorithme de Bellman-Ford distribué qui comme son nom l'indique se distribue sur plusieurs machines facilement.

Application 43: L'algorithme de Bellman Ford distribué est utilisé en réseau pour le routage : pour trouver le chemin que doivent emprunter les paquets.

Développement 2: Présentation et terminaison de Bellman Ford distribué