925

Graphes : représentations et algorithmes

Motivations : Réseaux de transport, routage, internet, le web, circuits imprimés.

I] Structures d'un graphe.

Déf 1 : Un graphe (non-orienté) est un couple (S,A) où :
- S est un ensemble fini. C'est l'ensemble des sommets.
- A est un sous-ensemble des paires de S est l'ensemble des arêtes.

→ Notation : $(u,v) \in A$ pour une arête du graphe.

Déf 2 : un graphe orienté est un couple (S,A) où :
- S est un ensemble fini. On fera souvent l'abus $S = [\![1, |S|]\!]$.
- A est un sous-ensemble des couples de S.

Rq 3 : En l'absence de mention contraire, les graphes considérés dans la suite sont orientés ou non-orientés.

Déf 4 : un poids sur un graphe G est une fonction $w : A \to \mathbb{R}$

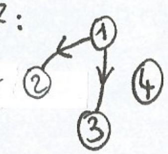
Déf 5 : l'arité d'un sommet s est le nombre d'arêtes parmi lesquelles il apparaît

1) Représentation par matrice d'adjacence.

Déf 6 : La matrice d'adjacence d'un graphe G est la matrice M de taille $|S| \times |S|$ à valeurs dans $\{0,1\}$ tq :

$$M[u,v] = 1 \quad \text{ssi} \quad (u,v) \in A$$

ex 7 :

$G_1 =$ 

$$\rightsquigarrow M = \begin{pmatrix} 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

prop 8 : Déterminer si deux sommets sont adjacents s'exécute en temps constant.
Déterminer la liste des adjacents d'un sommet est en $\mathcal{O}(|S|)$.
La mémoire occupée par un stockage en matrice d'adjacence est $\mathcal{O}(|S|^2)$.

2) Représentation par listes d'adjacence.

Déf 9 : La liste d'adjacence est un tableau de $|S|$ listes. Pour chaque sommet u, la liste correspondante contient tous les sommets adjacents à u.

ex 10 :

$G_1 \rightsquigarrow ([2,3], [\,], [\,], [\,])$

prop 11 : Déterminer si deux sommets sont adjacents s'exécute en $\mathcal{O}(|S|)$ (pire cas : le graphe est complet)
Déterminer la liste des adjacents d'un sommet est en temps constant.
La mémoire occupée par un stockage par liste d'adjacence est $\mathcal{O}(|S| + |A|)$.

Rq 12 : Il existe d'autres représentation (matrice d'incidence).

II] Parcours dans un graphe.

1) Parcours en largeur et en profondeur.

* Parcours en largeur : C'est un algorithme de recherche qui progresse à partir d'un sommet s en parcourant successivement les sommets par distance à s croissante.
Une implémentation itérative est la suivante

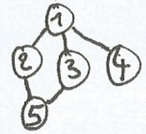
Algo 13 :

☆ PL(G)

```
PL (G, s) :
    marquer tous les noeuds en blanc
    Initialiser une file F = [s]
    Tant que F ≠ []
        u = F.défile
        Si u est blanc :
            marquer u en noir (*)
            Pour v successeur de u :
                F.enfile (v)
```

(*) : on peut traiter le noeud ici, par exemple en stockant les dates de traitement en $\mathcal{O}(1)$

ex 14 :

G_2 =  ⇝ ordre de parcours : (1, 2, 3, 4, 5)

prop 15 : le PL s'exécute en $O(|S|+|A|)$ avec une liste d'adjacence, et en $O(|S|^2)$ avec une matrice d'adjacence.

* Parcours en profondeur : C'est un algorithme de recherche qui progresse à partir d'un sommet s en parcourant successivement les sommets par profondeur croissante.

Algo 16 : Il suffit de remplacer la file de priorité F par une pile P dans l'algorithme précédent. On le note PP.

☆ PP(G,s) ↓ PP(G)

prop 16 : le PP s'exécute en $O(|S|+|A|)$ avec une liste d'adjacence.

ex 17 :

G_2 ⇝ ordre de parcours : (1, 2, 5, 3, 4)

App 18 : l'algorithme du tri topologique utilise un PP sur un graphe orienté pour lister un ensemble de tâches partiellement ordonné par ordre de priorités.

2) Parcours exhaustifs

Déf 19 un chemin de s à t dans un graphe G, de longueur n, est une succession d'arêtes de G de la forme (s_i, s_{i+1}) où $s_0 = s$; $s_n = t$. Si G est pondéré, le poids d'un tel chemin est la sommes des poids de ses arêtes.

Déf 20 : un chemin de s à s de longueur non-nulle est appelé un cycle.

Déf 21 : un cycle est dit eulérien s'il passe une unique fois par chaque arête.

pb 22 : (EULERPATH) Etant donné un graphe G, existe-t-il un cycle eulérien ?

prop 23 : Un graphe connexe non-orienté G contient un cycle eulérien ssi tous ses sommets sont d'arité paire. Un graphe connexe orienté G est eulérien ssi pour chaque sommet, nbre d'arêtes entrantes = nbre d'arêtes entrantes

→ Ceci permet de prouver que le problème EULERPATH est dans P.

Déf 24 : un cycle est hamiltonien s'il passe une unique fois par chaque sommet

pb 25 : (HAMPATH). Etant donné un graphe G, existe-t-il un cycle hamiltonien ?

prop 26 : le problème HAMPATH est NP-complet.

3) Parcours minimisants

pb 27 : (TSP) (voyageur de commerce). Etant donné G non-orienté, pondéré, déterminer un cycle hamiltonien de poids minimal.

prop 28 : le problème de décision associé à TSP est NP-complet. Le problème TSP n'est P(n)-approximable pour aucune fonction P(n) calculable en temps polynômial.

prop 29 : (TSP euclidien) Si le poids vérifie l'inégalité triangulaire, le problème devient 2-approximable par un algorithme glouton.

Déf 30 : un arbre de G couvrant S est un arbre dont les sommets contiennent S et dont les arêtes sont incluses dans celles de G. Le poids d'un tel arbre est la somme des poids de ses arêtes.

pb 31 : (ACM). Etant donné un graphe G pondéré, non-orienté, déterminer un arbre couvrant de poids minimal, enraciné en un sommet r ∈ S donné.

algo 32 : L'algorithme de Prim construit un tel arbre couvrant, en stockant les noeuds restant à ajouter avec une file de priorités F selon le poids minimal d'une arête reliant ce noeud à l'arbre déjà construit. Le tableau clé contient cette valeur.

```
Prim (G, w, r) :      # G = (S,A)
  pour chaque sommet u ∈ S :
  |  clé[u] ← ∞
  |  π[u] ← NIL       # le tableau des pères.
  clé[0] ← 0
  F ← S               # implanter une file de priorités.
  Tant que F ≠ ∅ :
  |  u ← Extraire_min (F)
  |  pour chaque sommet v adjacent à u :
  |  |     Si v ∈ F et w(u,v) < clé[v] :
  |  |     |   π[v] ← u
  |  |     |   clé[v] ← w(u,v)
  Renvoyer π
```

* Correction : on n'ajoute que des arêtes sûres.

* Complexité : Elle varie en fonction de la structure de données de la file F et du choix de la représentation R du graphe. Dans le cas où F est un tas binaire et R une liste d'adjacence : $O(A \log S)$

pb 33 : (STEINER) Etant donné un graphe $G=(S,A)$ et une partition des sommets $S = S_1 \sqcup S_2$, déterminer un arbre de G couvrant S_1 de poids minimal.

thm 34 : Le problème de décision associé à STEINER est NP-complet.

> **DEV 1 :** Réduction isofacteur de STEINER à STEINER euclidien + 2-approximation par arbre couvrant minimal.

4) Plus court(s) chemin(s)

pb 35 : (SHORTPATH). Etant donné un graphe orienté pondéré, sans cycle de poids négatif, et un sommet origine $s \in S$. Déterminer, s'il existe, un chemin de poids minimal de s à n'importe quel sommet $v \in S$.

Algo36 : L'algorithme de Bellman-Ford utilise la technique du relâchement, diminuant progressivement une estimation $d[v]$ du poids d'un plus court chemin depuis l'origine s vers chaque sommet $v \in S$.

```
Bellman-Ford(G, w, s) :  # G=(S,A)
    pour chaque sommet v ∈ S :          } initialisation des
    |   d[v] ← ∞                        } distances et des
    |   π[v] ← NIL                      } pères (dans un
    d[s] ← 0                            }    chemin)
    Pour i = 1 à |S|-1 :
    |   Pour chaque arc (u,v) ∈ A :
    |   |   Si d[v] > d[u] + w(u,v) :       } relâchement
    |   |   |   d[v] ← d[u] + w(u,v)        } de l'arête (u,v)
    |   |   |   π[v] ← u                    }
    Retourner π (ou d si on ne veut que les distances)
```

* **Correction :** si $p=(v_0 \dots v_k)$ est un plus court chemin de $s=v_0$ à v_k et si les arêtes de p sont relâchées dans l'ordre $(v_0,v_1), \dots, (v_{k-1},v_k)$, alors $d[v_k] = \delta(s,v_k)$.

* **Complexité :** $O(|S|\times|A|)$ avec des listes d'adjacence, $O(|S|^3)$ avec des matrices d'adjacence.

Algo37 : L'algorithme de Dijkstra présuppose que le poids est positif. Il utilise un parcours en profondeur qui met en oeuvre une file de priorités. Avec une liste d'adjacence, il a une complexité $O(|S|^2)$.

Algo38 : L'algorithme de Floyd-Warshall calcule tous les plus courts chemins à l'aide de la programmation dynamique, sans hypothèse sur le signe du poids. Il utilise fondamentalement la représentation par matrices d'adjacence et sa complexité est $\Theta(|S|^3)$.

III) Connexité et flot dans un réseau

1) Connexité et calcul de composante(s) connexe(s).

prop 39 : Si un graphe non-orienté $G=(S,A)$ est connexe, alors $|A| \geq |S|-1$

App40 : le nombre minimal de transpositions pour engendrer G_n est $(n-1)$.

Rq 41 : le calcul de CC dans un graphe non-orienté peut se faire à l'aide d'un simple parcours en largeur, ou profondeur, en $O(|S|+|A|)$. Les algorithmes de calcul d'arbre couvrant permettent aussi ce calcul.

Algo 42 : L'algorithme de Kosaraju calcule les composantes fortement connexes d'un graphe non-orienté en effectuant deux parcours en profondeur : un sur le graphe G, l'autre sur son transposé $G^{\perp}$ défini avec les mêmes sommets et des arêtes $A^{\perp} = \{(u,v) \text{ pour } (v,u) \in A\}$.

```
Kosaraju(G) :
(1) | Appeler PP(G) pour calculer les dates de fin de traitement f[u]
    | pour chaque sommet u ∈ S.
(2) | Calculer G⊥
(3) | Appeler PP(G⊥) en traitant les sommets par ordre de f[u] décroissant.
(4) | Renvoyer la liste des sommets traités en ligne (3).
```

* **Correction :** les k premières arborescences produites en ligne (3) donnent des CFC

* **Complexité :** avec des listes d'adjacence : $O(|S|+|A|)$.

Algo 43 : L'algorithme de Tarjan est aussi de complexité linéaire, en n'utilisant qu'un seul parcours en profondeur, à l'aide d'un marquage spécifique.

2) Flot maximal et coupe minimale.

Def 43 : une coupe dans un graphe orienté G est une partition $S = S_1 \sqcup S_2$.
→ le poids d'une coupe est la sommes des capacités des arêtes reliant S_1 à S_2.
un flot est une fonction $f : A \to \mathbb{R}$, dans un graphe pondéré, satisfaisant :

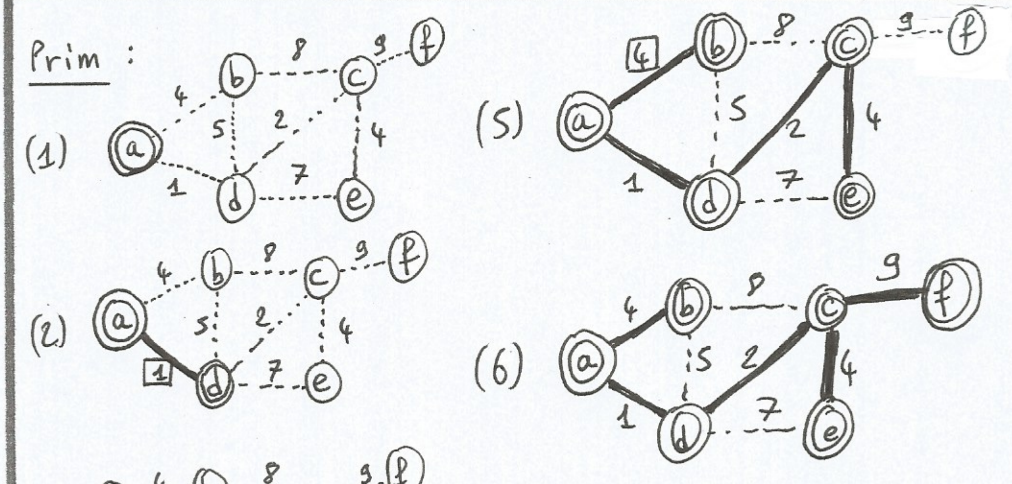
1. la contrainte de capacité : $\forall a \in A,\ 0 \leq f(a) \leq w(a)$
2. la conservation du flot : $\forall x \in S,\ \sum_{a=(x,\cdot)} f(a) = \sum_{a=(\cdot,x)} f(a)$

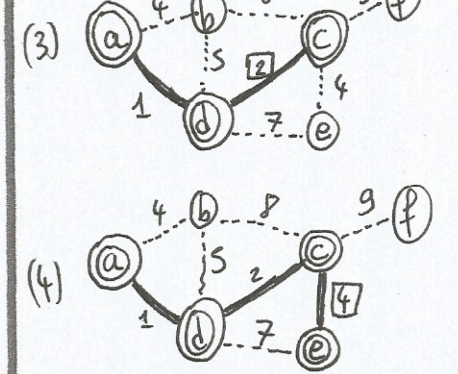
On se donne deux sommets s et t tq il existe un chemin de s à t.

thm 44 : (Ford-Fulkerson) Le flot maximum $\max_{f \text{ flot}} \{\sum_{a=(s,\cdot)} f(a)\}$ est égal au poids minimal d'une coupe (S_1,S_2) telle que $s \in S_1$ et $t \in S_2$.

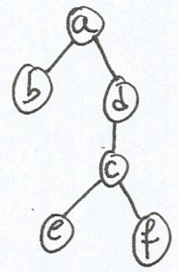
> **DEV 2 :** Algorithme d'Edmonds-Karp. Correction. Complexité en $O(|A| \times |S|^2)$

Prim :



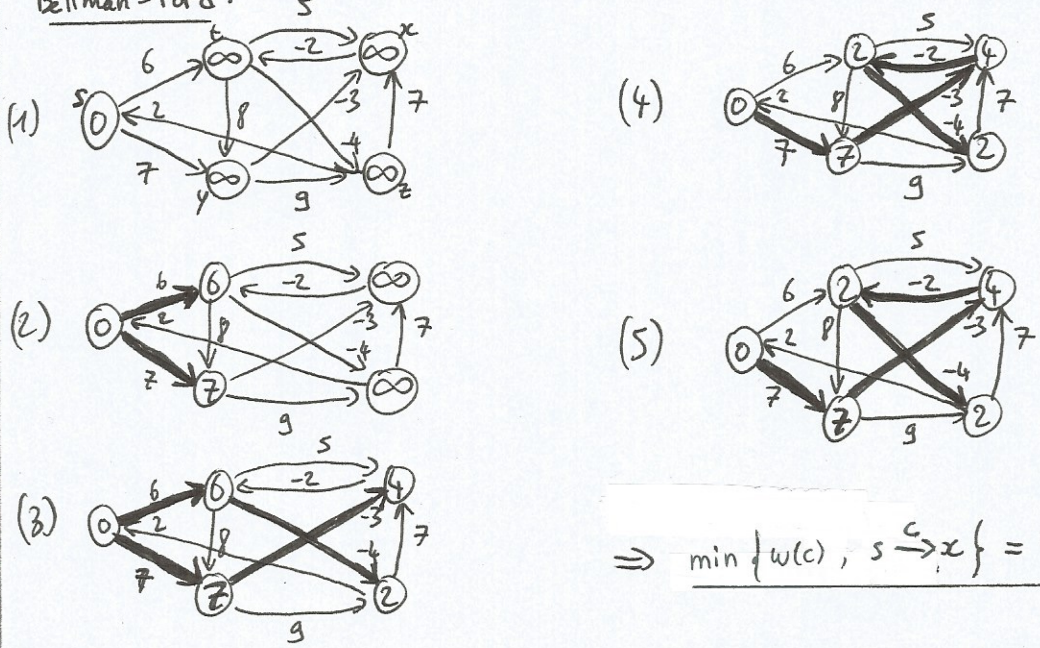


Sortie :



$w = 27$

Bellman-Ford :



$$\Rightarrow \min \{ w(c) \, , \, s \xrightarrow{c} x \} = 7 - 3 = 4$$