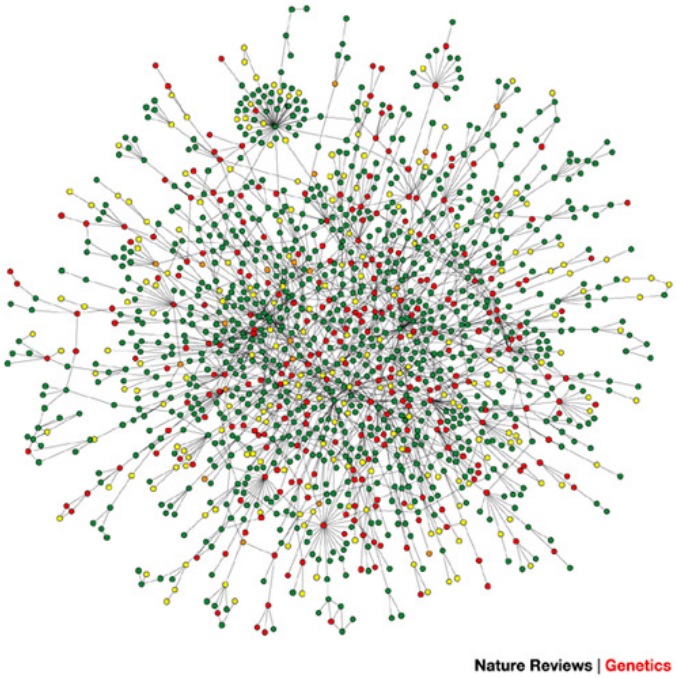
Graphes

Graphes non-orientés

Déf. Un **graphe non-orienté** est representé par un couple (V, E) où $E \subseteq \binom{V}{2}$ (paires non-ordonnées). V est l'ensemble des sommets et E est l'ensemble des arêtes.



Barabási & Oltvai *Nature Rev Genet* 5 :101, 2004

Graphes orientés

Déf. Un **graphe orienté** est representé par un couple (V, E) où $E \subseteq V \times V$ (paires ordonnées). V est l'ensemble des nœuds ou sommets, et E est l'ensemble des arcs.

Terminologie

Graphe non-orienté

L'arête $\{u, v\}$ est dénotée par uv.
Si $uv \in E$, alors v est adjacent à u.
L'arête $uv \in E$ est incidente aux sommets u et v.
Le degré de $u \in V$ est le nombre d'arêtes qui y sont incidentes.

Le graphe non-orienté $G = (V, E)$ avec $E = \binom{V}{2}$ est un **graphe complet**, dénoté par K_n où $n = |V|$.

Graphe orienté

L'arc (u, v) est dénotée par uv : l'arc part de u et arrive à v.
Si $uv \in E$, alors v est adjacent à u.
Le degré sortant de $u \in V$ est le nombre d'arcs qui y partent ; le degré rentrant est le nombre d'arcs qui y arrivent.

Chemins

Un **chemin** de longueur ℓ est une séquence $v_0, v_1, \ldots, v_\ell$ où $v_{i-1}v_i \in E$ pour tout $i = 1, \ldots, \ell$.

($\ell = 0$ est OK : chemin sans arêtes/arcs.)

Si $v_0 = v_\ell$, alors le chemin forme un **cycle**.

(Plus précisement, les sommets initial et final ne sont pas distingués dans le cycle.)

Le chemin $v_0 \cdots v_\ell$ est **élémentaire** ssi $v_1, \ldots, v_\ell$ sont distincts. Si $v_0 = v_\ell$, alors le chemin forme un **cycle élémentaire**.

Un graphe sans cycle est dit **acyclique**.

Un graphe non-orienté est **connexe** si chaque paire de sommets est relié par un chemin.

Un graphe non-orienté connexe acyclique est un **arbre**.

Sous-graphes

Le graphe $G' = (V', E')$ est un **sous-graphe** de $G = (V, E)$ ssi $V' \subseteq V$ et $E' \subseteq E$.

Étant donné un sous-ensemble de sommets $V' \subseteq V$, le sous-graphe de G **engendré** par V' est le graphe $G' = (V', E')$ avec $E' = \left\{uv \in E \colon u, v \in V'\right\}$.

Pondération

Graphe ponderé : chaque arc (ou arête) possède un **poids** ou coût associé, défini par la fonction de pondération $c\colon E \mapsto \mathbb{R}$.

Poids d'un sous-graphe : somme de poids des arcs dans le sous-graphe

Poids d'un chemin : somme de poids des arcs dans le chemin

Questions intéressantes

Stocker les graphes dans le mémoire d'un ordinateur

Parcours d'un graphe

Vérifier si le graphe est connexe

Plus court chemins

Arbres couvrants

Comment stocker le graphe ?

Matrice d'adjacence : matrice $V \times V$, $A[u, v]$ donne le poids de uv ($\pm\infty$ ou NaN pour arcs non-existants), ou valeurs booléennes pour noter juste présence

Listes d'adjacence : liste $\mathsf{Adj}[u]$ pour chaque sommet u qui stocke l'ensemble $\{v : uv \in E\}$ ou l'ensemble des couples $\{\langle v, c(u, v)\rangle : uv \in E\}$.

Usage de mémoire : dépend de la densité du graphe $\frac{|E|}{|V|^2}$.

Déterminer si $uv \in E$ ou $c(u, v)$: rapide avec la matrice mais plus lente avec les listes d'adjacence

Parcours

Technique essentielle : marquage/coloriage «jamais vu», «déjà vu»

```
     Algo PARCOURS-PROFONDEUR(u)
P1   // prévisite de u
P2   marquer u // «déjà vu»
P3   pour tout v adjacent à u
P4       si v n'est pas marqué alors PARCOURS-PROFONDEUR(v)
P5   // post-visite de u
```

(Généralisation du parcours préfixe ou postfixe sur les arbres.)

Parcours en largeur

Utiliser une queue Q

```
     Algo PARCOURS-LARGEUR(s)
L1   Q.enqueue(s)
L2   tandis que Q n'est pas vide
L3       u ← Q.dequeue()
L4       marquer u // «déjà vu»
L5       pour tout v adjacent à u faire
L6           si v n'est pas marqué alors Q.enqueue(v)
```

Temps de calcul : $O(|E|)$ avec listes d'adjacence ou $O(|V|^2)$ avec matrice d'adjacence

Parcours

Idée générale : suivre toujours une arête qui mène d'un sommet visité à un sommet non-visité

Parcours par profondeur : choix d'arête/arc uv où u est visité le plus récemment (pile)

Parcours par largeur : choix d'arête/arc uv où u est visité le moins récemment (queue)

Tri topologique

Tri topologique : tri des sommets d'un graphe orienté t.q. si $uv \in E$, alors v est énuméré après u.

(N'est pas possible s'il y a des circuits dans le graphe.)

P.e., précédence de taches — prérequis des cours

Méthode : maintenir le nombre de prérequis pour chaque sommet :
- initialiser par degré rentrant
- on peut énumérer un sommet sans prérequis pendants - décrementer quand prédecesseur est ajouté à la liste

Tri topologique (cont)

Solution avec une queue pour les sommets avec tous les prérequis satisfaits, en utilisant des listes d'adjacence

```
     Algo TRI-TOPOLOGIQUE
T1   pour tout u ∈ V faire prerequis(u) ← 0
T2   pour tout u ∈ V faire
T3       pour tout v ∈ Adj[u] faire prerequis(v) ← prerequis(v) + 1
T4   pour tout u ∈ V si prerequis(u) = 0 alors Q.enqueue(u)
T5   tandis que Q n'est pas vide
T6       u ← Q.dequeue()
T7       énumérer u
T8       pour tout v ∈ Adj[u] faire
T9           prerequis(v) ← prerequis(v) − 1
T10          si prerequis(v) = 0 alors Q.enqueue(v)
```

Tri topologique (cont)

Temps de calcul : $O(|V| + |E|)$.

enqueue et dequeue prend $O(1)$ pour chaque sommet

Parcours initiel T2–T3 en $O(|V| + |E|)$

Lignes T9–T10 executés $|E|$ fois

Arbre couvrant

Déf Un arbre couvrant du graphe non-orienté $G = (V, E)$ est un sous-graphe $T = (V, E')$ connexe acyclique.

Problème de l'arbre couvrant minimal (ACM) : arbre couvrant avec poids minimum quand les arêtes sont ponderées.

Méthode gloutonne : on construit l'ACM arête par arête, en incluant une petite arête ou en excluant une grande arête à la fois.

ACM (cont)

Déf. Une **coupure** $(X, \bar{X})$ d'un graphe non-orienté $G = (V, E)$ est une partition de ses sommets : $X \subset V, \bar{X} = V \setminus X$. L'arête uv traverse la cupure (ou y appartient) si $u \in X$ et $v \in \bar{X}$.

Idée de base : grandes arêtes dans les cycles sont rejetées, petite arêtes dans coupures sont acceptées.

Coloriage des arêtes : **rouge** (rejetée) ou **bleue** (acceptée).

Règle bleue : Choisir une coupure sans arête bleue. Choisir une arête non-coloriée avec poids minimal qui traverse la coupure et la colorier par bleue.

Règle rouge : Choisir un cycle élémentaire sans arête rouge. Choisir une arête non-coloriée dans le cycle avec poids maximal et la colorier par rouge.

ACM (cont)

Thm Soit G un graphe connexe.

1. Il existe toujours un ACM qui contient toutes les arêtes bleues et aucune arête rouge, quel que soit l'ordre d'application des règles.
2. On peut appliquer soit la règle rouge soir la règle bleue tandis qu'il existe des arêtes non-coloriées.

Preuve Induction pour 1. Au début c'est vrai : il existe un ACM T avec toutes les arêtes bleues et aucune arête rouge. Supposons qu'on applique la règle bleue en coloriant l'arête uv, et soit T l'ACM avant l'application. Si $uv \in T$, alors OK. Si $uv \notin T$, alors il y a un chemin $u \rightsquigarrow v$ dans T, avec une arête e' qui traverse la même coupure. On a $c(e) \leq c(e')$ et donc l'arbre $T' = T \cup \{e\} \setminus \{e'\}$ est un ACM. Argument similaire pour la règle rouge...

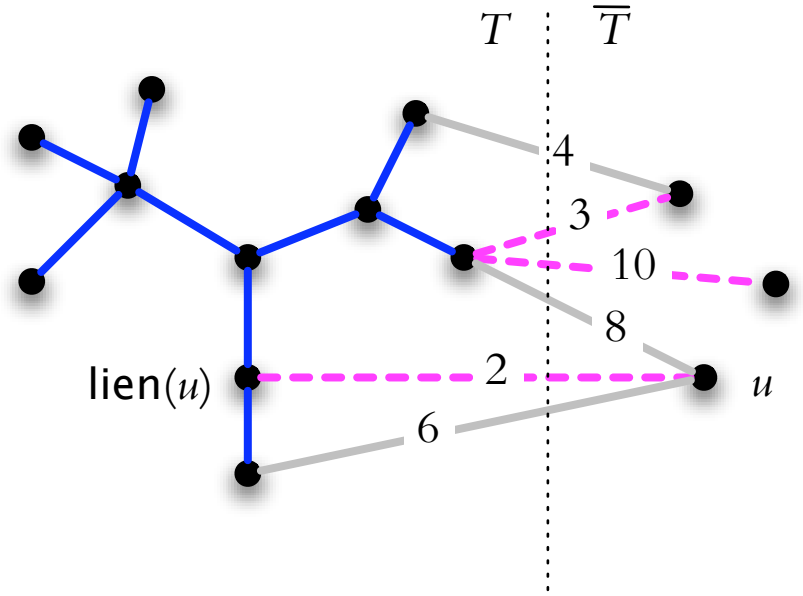
Pour démontrer 2, supposons par contradiction que la méthode finit avec des arêtes non-coloriées. Les arêtes bleues forment une forêt (ensemble d'arbres). S'il y a une arête e non-coloriée qui joint deux arbres bleus $\Rightarrow$ appliquer la règle bleue. S'il y a une arête e non-coloriée dont les extremités sont dans le même arbre $\Rightarrow$ appliquer la règle rouge. $\square$

ACM (cont)

Algorithme de **Kruskal** : pour toute arête uv dans l'ordre non-décroissant, si u et v appartiennent au même arbre bleu, alors la colorier par rouge ; sinon par bleue

Algorithme de **Prim** (+Jarník et Dijkstra) : on maintient l'arbre T défini par les arêtes bleueus — appliquer la règle bleue à la coupure $(T, \bar{T})$.

Prim



Idée de l'implantation : pour chaque $u \notin T$ la meilleure arête $(u, \mathsf{lien}(u))$ dans la coupure $(T, \bar{T})$ est placée dans un monceau

Prim : code

Implantation utilise un monceau H, on a besoin d'un sommet de départ s

```
      Algo ACM-PRIM(V, c, s)
AP1   pour tout u ∈ V initialiser cle(u) ← ∞ ; lien(u) ← null
AP2   cle(s) ← 0 ; initialiser H ← ∅ ; H.insert(s)
AP3   tandis que H ≠ ∅
AP4       u ← H.deleteMin() ; cle(u) ← −∞
AP5       ajouter arête (u, lien(u)) si lien(u) ≠ null
AP6       pour tout v ∈ Adj[u] faire
AP7           si c(u, v) < cle(v) alors
AP8               cle(v) ← c(u, v) ; lien(v) ← u
AP9               si v ∉ H alors H.insert(v) sinon H.decreaseKey(v)
```

Prim (cont)

$|V| = n$, $|E| = m$

n fois deleteMin(),
n fois insert(),
$m - n + 1$ fois decreaseKey().

Temps de calcul avec tas d-aire :

$O(nd \log_d n + m \log_d n)$, choisir $d = \lceil 2 + m/n \rceil$.
$\Rightarrow$ temps de $O(m \log_{2+m/n} n)$. Quand $m = \Omega(n^{1+\epsilon})$ avec $\epsilon > 0$, on a $O(m/\epsilon)$.

Temps de calcul avec tas Fibonacci :

$O(m + n \log n)$

Plus court chemins

Graphe $G = (V, E)$ orienté, arcs pondérés
(si graphe non-orienté, alors remplacer chaque arête par deux arcs aller-retour)

Rappel : poids d'un chemin $v_0, v_1, \ldots, v_\ell$ est $\sum_{i=1}^{\ell} c(v_{i-1}, v_i)$.

a trouver le plus court chemin de s à t
b trouver le plus court chemin de s à chaque sommet
c trouver le plus court chemin de chaque sommet à t
d trouver le plus court chemin entre chaque paire de sommets

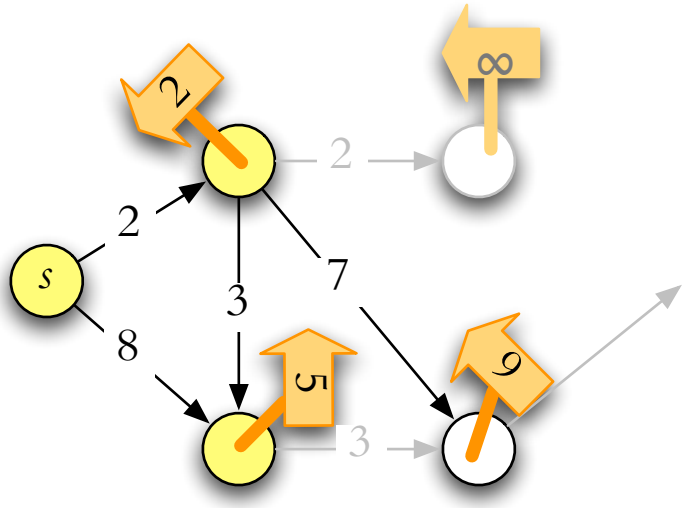
c équivaut **b** (renverser les arcs) ; toutes les solutions connues de **a** donnent une solution partielle à **b** ou **c**

Problème : cycle à poids négatif — le plus court chemin n'est pas défini

PCC - source

Plus court chemin d'une source s à chaque sommet

Idée d'étiquetage : maintenir un «panneau» à chaque sommet v : «il existe un chemin de longueur d à partir de sommet u»



Étiquetage : **si** $d(u)+c(u,v) < d(v)$ **alors** $d(v) \leftarrow d(u)+c(u,v)$; $p(v) \leftarrow u$

PCC - source

Amélioration : quand un sommet u est déjà étiqueté, on peut explorer tous les arcs partants — enregistrer que u est «exploré»

État d'un sommet : « non-étiqueté », « étiqueté » (mais pas exploré), ou « exploré »

Exploration : choisir un sommet u qui est «étiqueté», et faire

Procédure EXPLORER(u)
E1 pour tout $v \in \mathsf{Adj}[u]$
E2 **si** $d(u) + c(u,v) < d(v)$ **alors**
E3 $d(v) \leftarrow d(u) + c(u,v)$
E4 $p(v) \leftarrow u$
E5 changer l'état de v à «étiqueté»

PCC - exploration

Ordre d'exploration :

- dans le cas d'un graphe acyclique, utiliser le tri topologique
- s'il n'y a pas d'arcs à poids négatif, choisir le sommet étiqueté avec d minimum
  — implantation avec monceau
- s'il y a des arcs à poids négatif, choisir le sommet étiqueté le moins récemment
  — implantation avec queue

Algorithme de Dijkstra

```
     Algo PCC-DIJKSTRA(V, Adj[], c, s)
D1   pour tout u ∈ V initialiser d(u) ← ∞ ; p(u) ← null
D2   d(s) ← 0
D3   initialiser H ← ∅ ; H.insert(s) // (d est la clé dans H)
D4   tandis que H ≠ ∅ // (H est l'ensemble des sommets «étiquetés»)
D5       u ← H.deleteMin() ;
D6       pour tout v ∈ Adj[u] // (exploration de u)
D7           si d(u) + c(u, v) < d(v) alors
D8               d(v) ← d(u) + c(u, v) ; p(v) ← u
D9               si v ∉ H alors H.insert(v) sinon H.decreaseKey(v)
```

À la fin, $d(u)$ est la longueur du plus court chemin de s à chaque u, le chemin est donné en reculant $p(u), p(p(u)), \ldots$.

Dijkstra (cont)

Algorithme de Dijkstra (plus court chemin) est presque identique à celui de Prim (arbre couvrant minimal) !

Analyse comme avant : $|V| = n$, $|E| = m$

n fois deleteMin() (chaque sommet est exploré une fois),
n fois insert(),
$m - n + 1$ fois decreaseKey().

Temps de calcul avec tas d-aire :

$O(nd \log_d n + m \log_d n)$, choisir $d = \lceil 2 + m/n \rceil$.
$\Rightarrow$ temps de $O(m \log_{2+m/n} n)$. Quand $m = \Omega(n^{1+\epsilon})$ avec $\epsilon > 0$, on a $O(m/\epsilon)$.

Temps de calcul avec tas Fibonacci : $O(m + n \log n)$

Temps de calcul sans monceau : $O(n^2)$ [pour trouver $\min d$, il faut parcourir tous les sommets étiquetés n fois]

Bellman-Ford [+ Moore]

Et s'il y a des arcs à poids négatif ? (mais pas de cycles à poids négatif !)

Dijkstra ne trouve pas les plus court chemins

Il faut peut-être explorer un sommet plus qu'une fois — stocker les sommets dans une queue

Bellman-Ford (cont)

Algo PCC-BELLMAN$(V, \mathsf{Adj}[], c, s)$

B1 **pour tout** $u \in V$ initialiser $d(u) \leftarrow \infty$; $p(u) \leftarrow \mathsf{null}$

B2 $d(s) \leftarrow 0$

B3 initialiser $Q \leftarrow \emptyset$; $Q.\mathsf{enqueue}(s)$

B4 **tandis que** $Q \neq \emptyset$ // *(Q est l'ensemble des sommets «étiquetés»)*

B5 $\quad u \leftarrow Q.\mathsf{dequeue}()$;

B6 $\quad$ **pour tout** $v \in \mathsf{Adj}[u]$ // *(exploration de u)*

B7 $\quad\quad$ **si** $d(u) + c(u, v) < d(v)$ **alors**

B8 $\quad\quad\quad d(v) \leftarrow d(u) + c(u, v)$; $p(v) \leftarrow u$

B9 $\quad\quad\quad$ **si** $v \notin Q$ **alors** $Q.\mathsf{enqueue}(v)$

Notez qu'on doit tester si un sommet est sur la queue (une valeur booléenne stocké avec chaque sommet)

Bellman-Ford (cont)

Temps de calcul : $|V| = n$, $|E| = m$
définir les tournées $0, 1, 2, \ldots$

- en tournée 0, le sommet s est exploré
- en tournée $j > 0$, tous les sommets sont explorés qui sont dans la queue à la fin de tournée $(j - 1)$

Chaque tournée prend $O(m)$ temps

À la fin de tournée j, $d(u)$ est la longueur du plus court chemin pour chaque u t.q. le chemin $s \rightsquigarrow u$ contient j arcs

Si la queue n'est pas vide à la fin de tournée $(n - 1)$, alors il y a un cycle à poids négatif et l'algorithme ne finit jamais

Bellman-Ford (cont)

$\Rightarrow$ il faut compter les tournées (ou le nombre de fois chaque sommet est défilé)

À la fin de tournée $n - 1$:

- si la queue est vide, alors chaque $d(u)$ est la longueur du plus court chemin
- s'il y a un sommet $u \in Q$, alors il existe un cycle à poids négatif en reculant $u, p(u), p(p(u)), \ldots$

$O(nm)$ temps de calcul

Tous les chemins

S'il n'y a pas d'arcs négatifs : Dijkstra pour tout $s \in V$ en $O(n(m + n \log n))$

S'il y en a ?

1. ajouter un sommet s et les arcs su pour chaque $u \in V$ avec $c(s, u) = 0$
2. calculer la longueur $d(u)$ du plus court chemin de s à chaque $u \in V$ par Bellman-Ford (on aura $d(u) \leq 0$)
3. soit $c'(u, v) = c(u, v) + d(u) - d(v)$
4. Dijkstra pour chaque sommet dans le graphe original mais avec pondération c'

- pour tout $uv \in E$ on a $c'(u, v) \geq 0$
- pour chaque chemin $u \rightsquigarrow v$, on a $c'(u \rightsquigarrow v) = c(u \rightsquigarrow v) + d(u) - d(v)$, ce qui ne dépend pas du chemin actuel mais seulement de ses extrémités, donc le plus court chemin selon c' est un plus court chemin selon c.

Temps de calcul : $O(n(m + n \log n))$