Algorithmique 2

Anne Benoit

L3, Département Informatique, ENS Lyon

Contents

Fonctionnement du cours

Planning du cours: `http://graal.ens-lyon.fr/~abenoit/algo2/`:
2h cours / 2h TD par semaine.

Evaluation

Contrôle continu: 1 partiel et des DMs (environ 3), peut être des devoirs sur table "surprise". Examen à la fin du semestre, note finale: (CC+NE)/2. Questions de cours/TD. Sur le fond, attention à la présentation, à la rédaction (cf Algo1).
Pas de TPs, implémentation dans le cours *Enseignement en Programmation Sportive* (EPS).

Résumé du cours

Suite du cours Algo1, grands principes de conception/analyse d'algorithmes illustrés sur différents thèmes:

- Paradigmes généraux de conception (diviser pour régner, algos gloutons, prog dyn) vus en Algo1
- Paradigmes généraux d'analyse (invariants, analyse pire cas / amortie / en moyenne), déjà vus en Algo1 et approfondis ici
- Algorithmique des ensembles (structures de données), des graphes, des mots, ...

Objectif: concevoir et analyser mathématiquement des algorithmes.
Science jeune, plein de problèmes ouverts.

Recommandations

Ecrire et analyser un algorithme: associer un théorème à chaque algorithme!

Algo Machin: préciser entrées, sorties, puis pseudo-code ou prose.

Théorème/Proposition: L'algorithme Machin calcule le bidule avec la complexité truc en temps dans le pire cas (préciser éventuellement nuances: avec telle probabilité, sous l'hypothèse H, à un facteur ε près...).

Démonstration: correction (l'algo termine en calculant ce qu'on veut) et analyse de complexité, avec détails d'implémentation si besoin.

Rédaction: trouver le juste milieu entre "l'algo suffit, pas de preuve nécessaire", et "raisonnement ultra-détaillé". Trouver le juste dosage par l'expérience: DM/partiel/TDs.

Analyse d'algorithmes: analyse de correction / de complexité.

Identifier des *invariants* (propriétés restant vraies pendant tout ou une partie de l'exécution).

Justifier la terminaison et compter les opérations (comptabilité exacte si possible).

Identifier les événements marquants se produisant pendant l'exécution.

Analyse amortie (vu en Algo 1): complexité dans le pire cas d'une séquence d'opérations. Les méthodes classiques: Agrégat, Comptable (crédits), Potentiel. Application aux tableaux dynamiques (vu en Algo1).

Modèle de calcul: RAM, comme en Algo1: coût d'accès mémoire uniforme, pas d'accès concurrent, mémoire illimitée. But: minimiser le nombre d'opérations: accès mémoire et opérations arithmétiques. Opérations élémentaires en $O(1)$.

Plan du cours

- Structures de données (3-4 séances)
- Algorithmique des graphes (majorité des séances)
- Algorithmique des mots (2-3 séances)

Bibliographie

- {Tout le cours} Introduction to Algorithms (Cormen, Leiserson, Rivest, Stein)
- {Partie Graphes} Introduction to Graph Theory (West)
- {Partie Mots} Elements d'algorithmique (Beauquier, Berstel, Chrétienne), disponible librement à l'adresse `http://www-igm.univ-mlv.fr/~berstel/Elements/Elements.html`

1 Structures de données

Objectif: maintenir un ensemble de données, pour répondre vite à des requêtes, avec d'éventuelles mises à jour rapides, en utilisant peu d'espace.

1.1 Opérations sur les SDs et premières SDs

Notations:

- S: ensemble courant des données;
- n: cardinal de S, i.e., nombre d'éléments dans la SD;
- x: élément de S;
- i: position dans S;
- k: rang dans l'ensemble (si ensemble totalement ordonné).

1.1.1 Opérations de base

- Manipulation de la SD: Create, Destroy, Copy, ...
- Manipulation des éléments: Insert, Delete, Find, Find-Youngest, Find-Smallest...
- Et plein d'autres! (Liste exhaustive dans "Compared to what?" de A. Rawlins.)

1.1.2 SDs classiques

Quelle SD qui implémente les opérations suivantes connaissez-vous?

Insert, Find-Youngest, Delete-Youngest – Stack - Pile: ensemble dynamique, élément à supprimer défini par l'ensemble. Element supprimé = le dernier inséré; LIFO; pile d'assiette. SD simple utilisant des pointeurs.

Insert, Find-Oldest, Delete-Oldest – Queue - File: une tête et une queue; FIFO; file d'attente à la caisse.

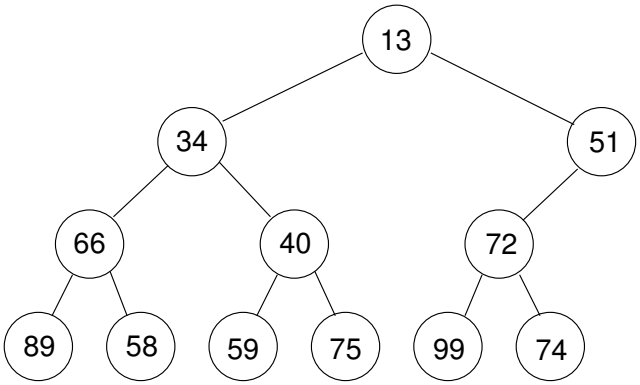
Insert, Find-Smallest, Delete-Smallest – Priority queue - File de priorité.

Insert, Delete, Find – Dictionnaire.

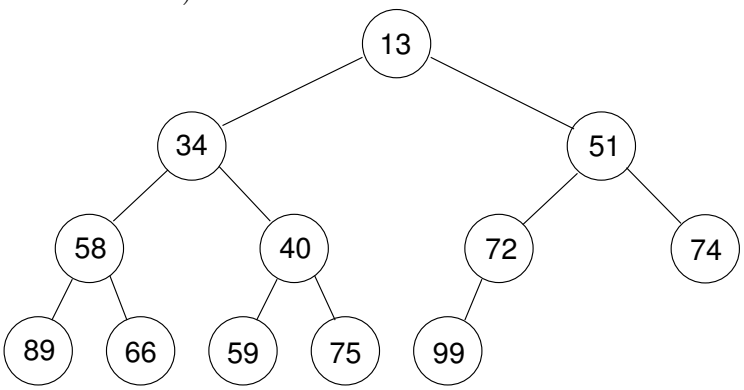
Union, Find-Structure – Partition (Union-Find).

1.2 Les SDs de base

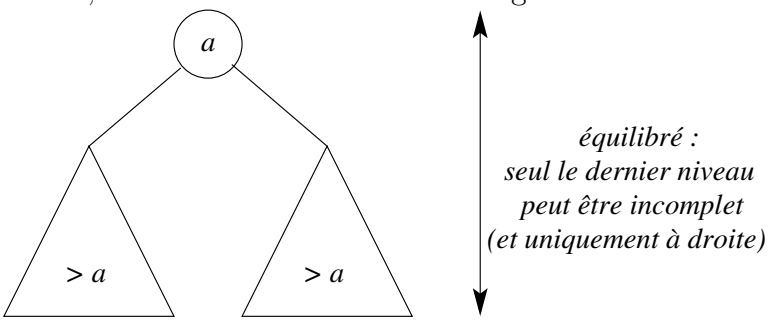
1.2.1 Une première SD: Tas binaire



Ressemble à la SD de tas binaire, mais deux erreurs : l'avant-dernier niveau devrait être plein (ici il manque un fils droit à 51), les pères doivent être plus petits que leurs fils (pas le cas de 66).



Et voilà, c'est un tas binaire! Vu en Algo1.



structure récursive dans les sous−arbres

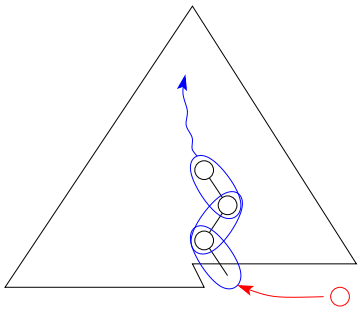
Profondeur $p = \lfloor \log_2 n \rfloor$ (arbre à n noeuds).

Opérations d'une file de priorité.

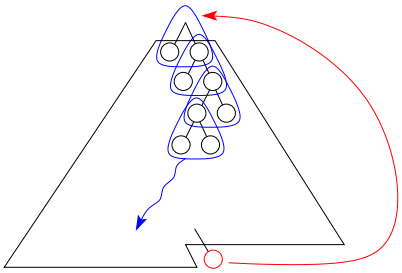
- INSERT: $O(\log n)$
- FIND-SMALLEST: $O(1)$
- DELETE-SMALLEST: $O(\log n)$

- BUILD FROM SCRATCH: $O(n)$

Ascension/descente le long des branches via des échanges père/fils en $O(1)$



INSERT = insertion par le bas



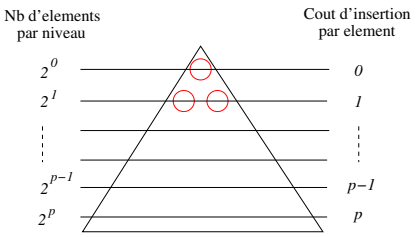
DELETE-SMALLEST = insertion par le haut

Légende:

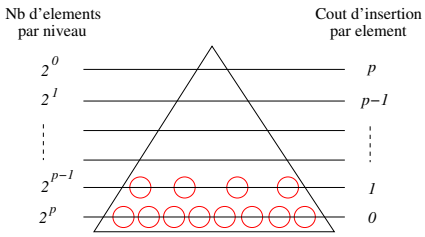
- Fléche rouge = position où insertion.
- Patate bleue = paquet d'éléments comparés pour choisir quels éléments échanger et conserver ainsi la structure de tas.

Construction d'un tas. (niveaux de 0 à $p = \lfloor \log_2 n \rfloor$).

Deux versions:



Remplissage par le haut
(avec insertions par le bas)



Remplissage par le bas
(avec insertions par le haut)

coût total au pire :
$\sum_{k=0}^{p} k2^k = \Theta(n \log n)$

coût total au pire :
$\sum_{k=0}^{p} k2^{p-k} = \Theta(n)$

Indication pour le calcul des coûts: $\sum_{k=0}^{p} 2^k = 2^{p+1} - 1$.
Somme pour le remplissage par le haut :

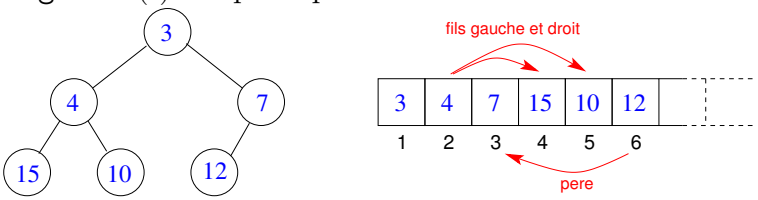
$$
\begin{array}{llllllll}
 & 2^1 & & & & & & \\
+ & 2^2 & + & 2^2 & & & & \\
+ & 2^3 & + & 2^3 & + & 2^3 & & \\
+ & \cdots & & & & & & \\
+ & 2^p & + & 2^p & + & \cdots & + & 2^p \\
\hline
= & 2^{p+1} - 2 & + & 2^{p+1} - 2^2 & + & \cdots & + & 2^{p+1} - 2^p \quad = p2^{p+1} - 2^{p+1} + 2
\end{array}
$$

Somme pour le remplissage par le bas :

$$\begin{array}{cccccccccc}
 & 2^0 & + & \cdots & + & 2^0 & + & 2^0 & + & 2^0 \\
+ & 2^1 & + & \cdots & + & 2^1 & + & 2^1 & & \\
+ & 2^2 & + & \cdots & + & 2^2 & & & & \\
+ & \cdots & & & & & & & & \\
+ & 2^{p-1} & & & & & & & & \\
\hline
= & 2^p-1 & + & \cdots & + & 2^3-1 & + & 2^2-1 & + & 2^1-1 \quad = 2^{p+1}-2-p
\end{array}$$

Philosophie de cette construction from scratch : pour faire des économies, il faut offrir une insertion pas chère au plus grand nombre et non pas l'inverse.

Implémentation. Plusieurs implémentations possibles pour `père`(i), `fils-droit`(i), `fils-gauche`(i) via p.ex. pointeurs ou tableaux.



`fils-gauche`(i)=$2i$, `fils-droit`(i)=$2i+1$, `père`(i)=$\lfloor i/2 \rfloor$

Attention aux indices dans la représentation d'un arbre binaire complet par un tableau: indicer de 1 à n, ne pas commencer par l'indice 0!

Pour construire le tas sur place:
CONTRUIRE-TAS(S,n)
1. $taille(S) \leftarrow n$;
2. pour i de $\lfloor n/2 \rfloor$ à 1 faire INSÉRER(S,i);

Implémentation du tas binaire via tableaux:
INSÉRER↓(S,i)
1. $\ell \leftarrow$ `fils-gauche`(i); $r \leftarrow$ `fils-droit`(i);
2. Si $\ell \leq taille(S)$ et $S[\ell] < S[i]$ alors $min \leftarrow \ell$ sinon $min \leftarrow i$;
3. Si $r \leq taille(S)$ et $S[r] < S[min]$ alors $min \leftarrow r$;
4. Si $min \neq i$ alors échanger $S[i] \leftrightarrow S[min]$ et INSÉRER↓(S,min)

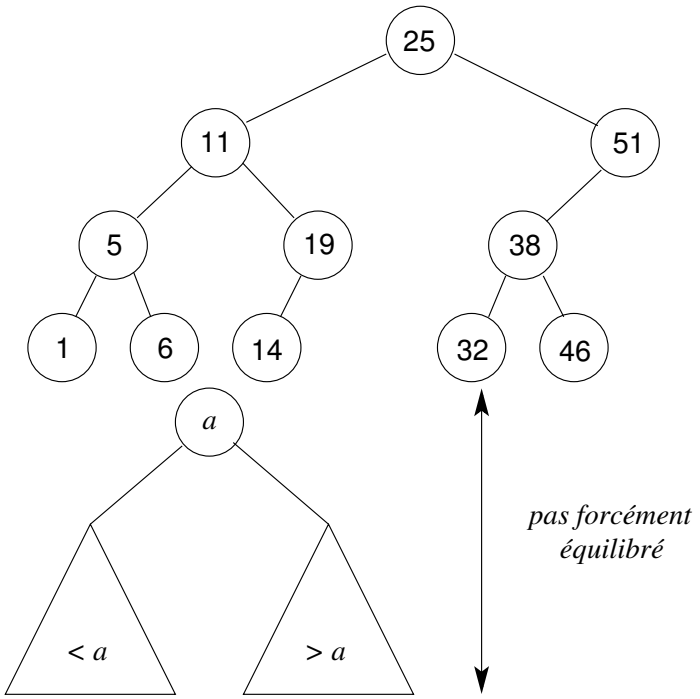
Utilisation : TRI-PAR-TAS(S,n) sur place
1. CONTRUIRE-TAS(S,n);
2. pour i de n à 2 faire
{échanger $S[1] \leftrightarrow S[i]$, $taille(S) \leftarrow taille(S)-1$, INSÉRER↓($S$,1)}

Ordre du tri? Décroissant. Comment trier sur place par ordre croissant? Echanges, ou bien tas binaire max au lieu de min.

Autres files de priorité .

	Insert	Find-Smallest	Delete-Smallest
Liste doubl. chaînée	$O(1)$	$O(n)$	$O(1)$
Idem + triée	$O(n)$	$O(1)$	$O(1)$
Tas binaire	$O(\log n)$	$O(1)$	$O(\log n)$
Tas binomial	$O(\log n)$	$O(1)$	$O(\log n)$
Tas de Fibonacci	$O(1)$	$O(1)$	$O(\log n)$ amorti

1.2.2 Un deuxième exemple (Arbres binaires de recherche, ABR)



structure récursive dans les sous−arbres

Opérations sur les ABR .

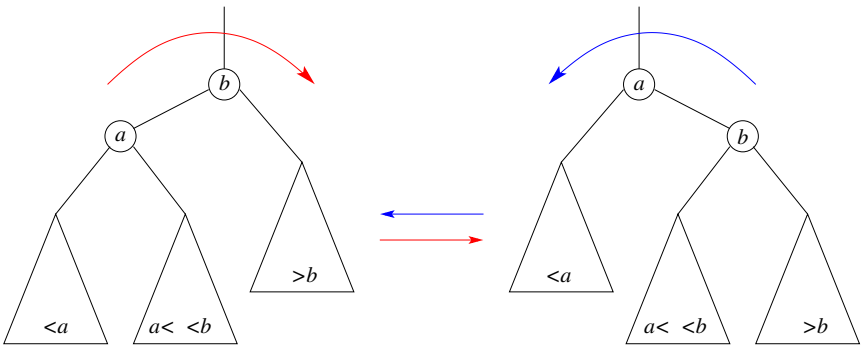
SD de dictionnaire et file de priorité.

- Insert
- Delete
- Find
- Find-Smallest

Performance des opérations dépendent de la profondeur h de l'arbre.

Toutes les opérations sont en $O(h)$. Dichotomie pour aiguiller les recherches dans le bon sous-arbre. Pour la suppression, si i admet deux sous-arbres non vides, rechercher r, la position du successeur immédiat de l'élément en i (aller un coup à droite puis parcourir les fils gauche). Remonter l'élément en r vers i, supprimer le nœud r (qui a au plus un sous-arbre à droite).

ABR équilibrés. Idée: maintenir une profondeur en $O(\log n)$ en ré-équilibrant l'arbre pendant ou après les opérations Insert et Delete: rotations en $O(1)$.



Se convaincre que rotation vers la droite est en $O(1)$ (idem pour la gauche) : en notant α le fils droit de a et β le père de b, en supposant que b est `fils-machin` de β, alors la rotation vers la droite s'écrit :

$$\texttt{père}(a) \leftarrow \beta;\ \texttt{fils-machin}(\beta) \leftarrow a;$$
$$\texttt{père}(\alpha) \leftarrow b;\ \texttt{fils-gauche}(b) \leftarrow \alpha;$$
$$\texttt{père}(b) \leftarrow a;\ \texttt{fils-droit}(a) \leftarrow b;$$

Plusieurs variantes d'ABRs équilibrés, on discutera notamment des arbres rouge-noir et des treaps dans la suite du cours.

Autres dictionnaires ?

	Insert	Delete	Find
Listes doubl. chaînées	$O(1)$	$O(1)$	$O(n)$
Tableaux dynamiques	$O(1)$ amorti	$O(1)$ amorti	$O(n)$
Arbres AVL	$O(\log n)$	$O(\log n)$	$O(\log n)$
Arbres rouge-noir	$O(\log n)$	$O(\log n)$	$O(\log n)$
Arbres "déployés"	$O(\log n)$ amorti	$O(\log n)$ amorti	$O(\log n)$ amorti
B-Arbres	$O(\log n)$	$O(\log n)$	$O(\log n)$

Autres dictionnaires pour des données spécifiques :

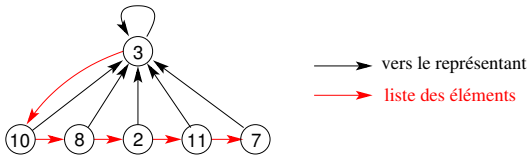
- trie / arbre préfixe (mots),
- arbres de van Emde Boas (entiers),
- tables de hachage (entiers) ...

1.2.3 Union-Find et complexité amortie

But : gérer une famille de parties disjointes $\mathcal{S} = \{S_1, \ldots, S_p\}$ pour

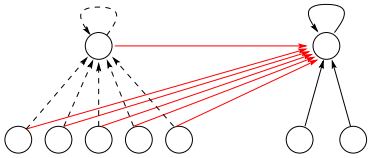
- FIND(x) : cherche la partie $S_i \ni x$.
- UNION(S_i,S_j) : fusionne les parties S_i et S_j.

Choix : identifier chaque partie par un de ses éléments et stocker la partie sous forme d'un arbre où nœuds = éléments et racine = identificateur (+ SD annexe : liste des éléments de la partie).



Version compressée : chaque élément pointe directement vers son représentant (profondeur 1) → FIND en $O(1)$.

Arbres avec compression immédiate (naïf). **Algo Union :** mettre à jour immédiatement tous les pointeurs d'une des deux parties vers l'identifiant de l'autre.



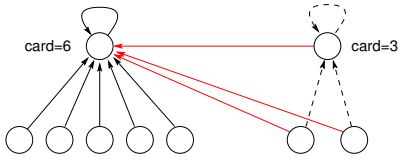
Théorème 1. *Partant de la partition avec uniquement des singletons, l'algo naïf avec compression immédiate a pour complexités pire cas :*

- *après u* UNION *:* FIND *en $O(1)$,* UNION *en $O(u)$,*
- *total pour u* UNION *et f* FIND *: $O(u^2 + f)$.*

Analyse de complexité : on a juste besoin du petit lemme + corollaire suivant sur le cardinal des parties après u opérations UNION.

- Lemme Partant des singletons, après u UNIONS, exactement u éléments ont modifié leur pointeur au moins une fois. En effet à chaque UNION, un seul élément met à jour pour la première fois son pointeur : c'est le représentant de la partie qui disparaît.
- Corollaire Partant des singletons, après u UNIONS, la taille maximum d'une partie est majorée par $u + 1$ (tous ses éléments ont mis à jour leur pointeur au moins une fois, excepté le représentant de la partie).

Arbres avec compression immédiate (pondéré). **Algo Union :** chaque partie est pondérée par son cardinal, c'est la plus légère qui met à jour ses pointeurs.

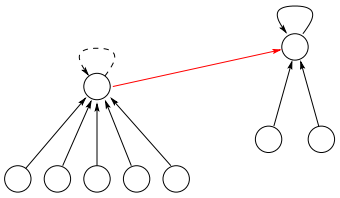


Théorème 2. *Partant de la partition avec uniquement des singletons, l'algo pondéré avec compression immédiate a pour complexité pire cas :*

- *après u* Union *:* Find *en $O(1)$,* Union *en $O(u)$,*
- *total pour u* Union *et f* Find *: $O(u \log_2 u + f)$.*

Analyse de complexité : compter le nb total de mises à jour des pointeurs vers les représentants des parties. Une technique (typique de l'analyse amortie) : se focaliser sur un objet et analyser les événements qu'il a subi.
▷ Si pointeur d'un élément mis à jour, c'est qu'il y a union de deux parties et qu'il est dans la plus petite, donc la taille de sa partie double au moins. Un élément subit donc au plus $\lfloor \log_2 card \rfloor$ mises à jour de son pointeur, où *card* est la taille finale de la partie contenant cet élément.
▷ Or le lemme + corollaire déjà vu (Théorème 1) s'applique toujours : partant des singletons, après u Unions, exactement u éléments ont modifié leur pointeur au moins une fois et la taille maximum d'une partie est majorée par $u + 1$.

Arbres sans compression (naïf). **Algo Union :** une des deux parties fait pointer son identifiant vers l'autre identifiant.



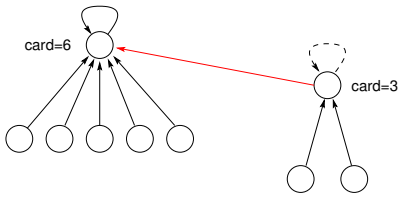
Théorème 3. *Partant de la partition avec uniquement des singletons, l'algo naïf sans compression a pour complexité pire cas :*

- *après u* Union *:* Find *en $O(u)$,* Union *en $O(1)$,*
- *total pour u* Union *et f* Find *: $O(u + fu)$.*

Analyse de complexité : certains arbres représentant les parties peuvent ressembler à des peignes de profondeur $\Theta(u)$ d'où le coût du Find, l'opération Union lie juste les deux racines en $O(1)$.

Ici SD annexe qui liste tous les éléments d'une partie n'est plus nécessaire!

Arbres sans compression (pondéré). Algo Union : chaque partie est pondérée par son cardinal (la profondeur de l'arbre marche aussi), c'est la plus légère qui fait pointer son identifiant vers l'autre (assure profondeur en $\log_2$).



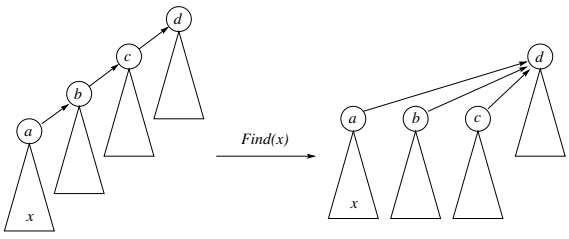
Théorème 4. *Partant de la partition avec uniquement des singletons, l'algo pondéré sans compression a pour complexité pire cas :*

- *après u* Union *:* Find *en $O(\log_2 u)$,* Union *en $O(1)$,*
- *total pour u* Union *et f* Find *: $O(u + f \log_2 u)$.*

Analyse de complexité : analyse de la forme de l'arbre. Invariant (valable pour les deux stratégies) : à tout instant, tout arbre construit par l'algo et représentant une partie vérifie $profondeur \leq \log_2 cardinal$ (le cardinal compte tous les éléments, c'est à dire tous les sommets de l'arbre, pas uniquement les feuilles). L'invariant est vrai pour un arbre à un noeud, de profondeur $0 = \log_2(1)$.

- Preuve de l'invariant pour la stratégie sur le cardinal : considérer une Union entre deux arbres de profondeur $prof_1$ (resp. $prof_2$) et cardinal $card_1$ (resp. $card_2$), et notons $prof$ (resp. $card$) la profondeur (resp. le cardinal) du nouvel arbre après fusion. Supposons que $card_1 \leq card_2$, alors :
  (1) si $card_1 < card_2$, $prof = \max(prof_1 + 1, prof_2)$.
  D'une part, $prof_1 + 1 \leq \log_2 card_1 + 1 = \log_2(2card_1) \leq \log_2 card$ à cause de la stratégie de fusion.
  D'autre part $prof_2 \leq \log_2 card_2 \leq \log_2 card$. Donc $prof \leq \log_2 card$.
  (2) si $card_1 = card_2$, $prof = \max(prof_1, prof_2) + 1 \leq \max(\log_2 card_1, \log_2 card_2) + 1 = \log_2(2card_2) = \log_2 card$ car $card_1 = card_2$ et $card = card_1 + card_2$.
- Preuve de l'invariant pour la stratégie sur la profondeur: mêmes notations que pour la stratégie sur le cardinal. Supposons que $prof_1 \leq prof_2$, alors :
  (1) si $prof_1 < prof_2$, $prof = prof_2 \leq \log_2 card_2 \leq \log_2 card$.
  (2) si $prof_1 = prof_2$, sans perte de généralité supposons $card_1 \leq card_2$,
  écrire alors $prof = prof_1 + 1 \leq \log_2 card_1 + 1 \leq \log_2(2card_1) \leq \log_2 card$.

Arbres à compression paresseuse. Amélioration : paresseux comme avant (Union pondérée sans compression) mais opportuniste (compression à chaque Find), version étudiée en TD d'Algo1.



Théorème 5. *Partant de la partition avec uniquement des singletons, l'algo de compression paresseuse a pour complexité pire cas :*

- *après u* UNION *:* FIND *en $O(\log_2 u)$,* UNION *en $O(1)$,*
- *total pour u* UNION *et f* FIND *: $O((u+f)\alpha(u+f,u))$.*

Et on a $\alpha(u+f,u) = \min\{i \geq 1 | Ackermann(i, \lfloor 1+f/u \rfloor) > \log u\} \leq \log^* u \leq 5$ ...

Récapitulatif Union-Find. Complexité totale pour f FIND et u UNION suivant les stratégies choisies:

FIND	UNION	Complexité
Sans poids	Sans compression	$O(u+fu)$
Sans poids	Avec compression	$O(u^2+f)$
Avec poids	Sans compression	$O(u+f\log_2 u)$
Avec poids	Avec compression	$O(u\log_2 u+f)$
Avec poids	Compression paresseuse	$O((u+f)\alpha(u+f,u))$

Meilleures complexités pour u et f qcq: les 3 versions avec poids

Meilleures complexités quand $f \geq 2u$ obtenu avec poids et compression (paresseuse ou non). Cas fréquent, par exemple pour UNION(x,y) qui fusionne les parties $S_i \ni x$ et $S_j \ni y$ et commence a priori par deux FIND sur x et y.

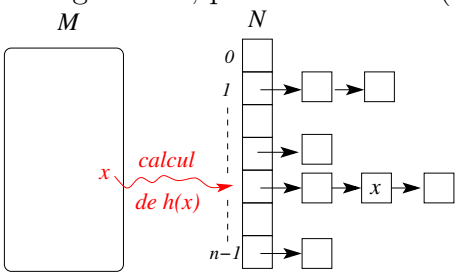
Il est parfois rentable d'être paresseux ;-), mais plein de contextes différents, parfois la compression immédiate est bonne (par exemple, si le processeur est souvent à l'arrêt, il peut en profiter pour compresser).

1.2.4 Tables de hachage et complexité en moyenne

Citation de Udi Manber, chef à Yahoo: "The 3 most important data structures in practice are 1. hash tables; 2. hash tables; 3. hash tables!"

- Fonctionnalités / Morphologie : dictionnaire via des fonctions (de hachage).
- L'approche "arithmétique" : se ramener à des données de type "nombre entier" $S \subseteq M = \{0,\ldots,m-1\}$, avec m fixé, sur lesquelles on peut potentiellement faire du calcul arithmétique.
- La clé : le modèle RAM ou ses variantes, qui inclut opérations arithmétiques, accès directs et indirects en mémoire en $O(1)$.
- La SD : une fonction de hachage h de M dans $N = \{0,\ldots,n-1\}$ qui donne l'adresse de stockage dans un tableau indicé par N.
- L'intérêt : un espace adapté (travailler avec espace proche des $|S|$ de la pratique plutôt que le $|M|$ théorique) accessible rapidement (si h calculable efficacement).
- Un souci : résoudre les collisions ($x, y \in M$ avec $x \neq y$ mais $h(x) = h(y)$) $\rightarrow$ une SD secondaire.

Recherche d'élément par hachage. Si $n = m$, on a une grande table et on utilise l'adressage direct, pas de collisions ($h(x) = x$). Dans le cas général, $n < m$.



SD secondaire : souvent une simple liste chaînée où recherche en O(longueur).

Différentes alternatives: Adressage ouvert, $h(x, i)$ pour $1 \leq i \leq n$ (permutation de $[1..n]$): un seul élément par case. Différentes techniques: sondage linéaire ($h(x, i) = h'(x) + i \mod n$), quadratique ($h(x, i) = h'(x) + c_1 i + c_2 i^2 \mod n$), ou double hachage ($h(x, i) = h_1(x) + i h_2(x) \mod n$).

Longueur max L d'une liste : pour toute fonction h, $L \geq m/n$ (atteint par exemple si $h(x) = x \mod n$).

Complexité pire cas pour une suite de s INSERT et f FIND :

$$\max(\text{complexité en temps}) \geq s + f \times L \geq f \min(s, \frac{m}{n}).$$

Analyse de la complexité pire cas : quelque soit la fonction de hachage choisie, il existe une valeur de hachage qui concentre plus de m/n éléments de M, le pire cas arrive si les s insertions correspondent à ces éléments, ils sont tous insérés dans la même liste et dont la longueur devient $\geq \min(s, \frac{m}{n})$.

Peut-on se débarrasser du facteur $\min(s, \frac{m}{n})$ pour avoir du $\frac{s}{n}$?
$\rightarrow$ randomiser pour avoir des SD secondaires de cette taille *en moyenne*.

Hachage universel. Définition: Soient $M = \{0, \ldots, m-1\}$ et $N = \{0, \ldots, n-1\}$ avec $m \geq n$. Une famille H de fonctions de M dans N est dite *universelle* si pour tout $x, y \in M$, $x \neq y$, en choisissant h au hasard uniformément dans H,

$$\mathbb{P}(h(x) = h(y)) \leq \frac{1}{n}.$$

Hachage randomisé: étant donnée une suite R de mises à jour et de recherches, tirer au hasard uniformément une fonction de hachage $h \in H$ et l'utiliser pour toute la suite R.

Théorème 6. *Partant d'une table vide, pour une suite R de longueur $s+f$, avec s INSERT et f FIND, en piochant h au hasard uniformément dans une famille universelle H,*

$$\mathbb{E}(\textit{complexité en temps}) \leq (s + f)\big(1 + \frac{s}{n}\big).$$

Dimensionnement : si $n \geq s$, alors temps moyen par opération ≤ 2.

Proof. Les s insertions se font en temps 1, et à chaque recherche, il y a au plus s éléments dans la table, qui est remplie à un facteur s/n. Reste à évaluer le temps d'une recherche, que l'on peut borner par $1 + s/n$.

X_{kl}: variable indicatrice qui vaut 1 s'il y a collision entre k et l. Par définition de la famille universelle, on a $E[X_{kl}] \leq 1/n$.

Y_k: variable aléatoire égale au nombre de clés autres que k hachées vers l'alvéole de k: $Y_k = \sum_{l \neq k} X_{kl}$, et par linéarité des espérances, $E[Y_k] \leq \sum_{l \neq k} 1/n$.

$M_{h(k)}$: taille de la liste de l'alvéole de k. Le temps de recherche de l'élément k est en moyenne $E[M_{h(k)}]$.

Si k n'est pas dans la table, $M_{h(k)} = Y_k$ et on somme au plus s éléments, donc $E[M_{h(k)}] \leq s/n$.

Si k est dans la table, $M_{h(k)} = Y_k + 1$ et on somme au plus $s - 1$ éléments, donc $E[M_{h(k)}] = E[Y_k] + 1 \leq (s-1)/n + 1 \leq 1 + s/n$, d'où le résultat. □

Construction d'une famille universelle. Pas trop compliqué, décrit dans le Cormen si vous voulez voir les détails.

Hachage parfait. Ensemble de s clés statiques, par exemple les noms de fichier sur un CD, les mots réservés d'un langage de programmation.

Hachage parfait: Find en $O(1)$ dans le cas le plus défavorable, et espace nécessaire pour stocker la table en $O(s)$.

Table de taille $n = s^2 \rightarrow$ soit X une variable aléatoire qui compte le nombre de collisions.

$$E(X) = \binom{s}{2} \times \frac{1}{n} = \frac{s^2 - s}{2} \times \frac{1}{s^2} < \frac{1}{2}.$$

(linéarisation des espérances, compter le nombre de couples et pour chaque couple l'espérance d'une collision est $1/n$: premier choisi n'importe où, et proba $1/n$ que le deuxième ait la même valeur).

Idée du hachage parfait : table de hachage de taille $n = s$, et résolution des collisions en utilisant une deuxième table de hachage de taille M_i^2, où M_i est le nombre d'éléments stockés en i.

Temps du find? $O(1)$ car pas de collisions dans la deuxième table.

Taille utilisée? $\sum M_i^2 = O(\sum \binom{M_i}{2}) = O(\text{nombre de collisions})$.

Autre façon de compter le nombre de collisions? Hachage universel:
$\mathbb{E}(\text{nombre de collisions}) = \binom{s}{2}\frac{1}{s} = O(s)$, d'où le résultat!

Conclusion sur cette partie: en SD vous pouvez être faible (tas), paresseux (union-find), joueur (hachage universel)... si c'est pour la bonne cause. Bien choisir ses SDs: ne pas hésiter à convertir la SD fournie en entrée en une autre plus adaptée (phase de précalcul). Exemple typique du tri déjà évoqué.

1.3 Comment adapter des SD connues à vos besoins?

1.3.1 Dictionnaire avec statistiques d'ordre

Par exemple, on veut un dictionnaire avec statistiques d'ordre.

Données : éléments d'un ensemble totalement ordonné.
Opérations souhaitées :

- INSERT(x,S), DELETE(x,S), FIND(x,S),
- FIND-ORDER(k,S) : trouver le k-ième plus petit élément dans S,
- FIND-RANK(x,S) : trouver le rang de x dans S, dans l'ordre croissant.

Remarque : FIND-RANK et FIND-ORDER sont exactement les opérations inverses l'une de l'autre.

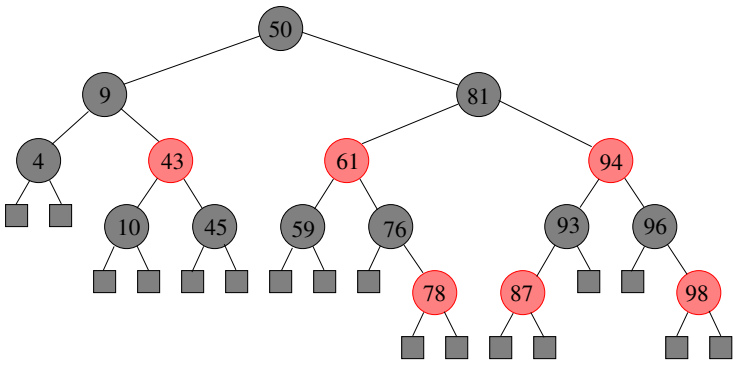
Les arbres rouge-noir. Définition: Un arbre rouge-noir est un arbre binaire de recherche vérifiant les propriétés suivantes:

1. les données sont stockés sur les nœuds internes et les feuilles ne portent rien (NIL),
2. chaque nœud est soit rouge, soit noir,
3. les feuilles sont noires,
4. si un nœud est rouge alors ses deux fils sont noirs,
5. tous les chemins descendants d'un même nœud vers les feuilles ont le même nombre de nœuds noirs.

Proposition 1. *Profondeur d'un arbre rouge-noir à n nœuds* $\leq 2\log_2(n+1)$.

Preuve: soit N le nombre de nœuds noirs par branche et p la profondeur de l'arbre,
▷ les N premiers niveaux de l'arbre binaire sont complets étant donné la condition (5), et donc $n \geq 2^N - 1$,
▷ la condition (4) implique qu'il y a au plus N rouges par branche et donc $p \leq 2N$,
▷ au final $n \geq 2^{p/2} - 1$ et donc $p \leq 2\log_2(n+1)$.

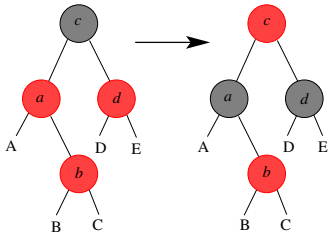
Exemple d'arbre rouge-noir:



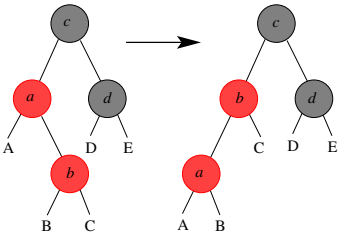
4

Règles pour l'insertion: INSERT↓(x,S):
1. Insérer x par le haut comme dans tout arbre binaire de recherche et colorer le nouveau sommet en rouge;
2. Rééquilibrer par des rotations/recolorations en remontant depuis le nouveau sommet vers la racine pour éliminer tout motif qui ne respecte pas la structure des arbres rouges-noirs.

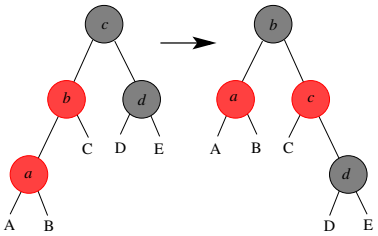
Implémentation détaillée: description d'un nombre fini de règles de "réécriture" de l'arbre par rotations/recoloration en fonction du motif local, détaillée dans le Cormen.



Règle 1 : recoloration



Règle 2 : rotation



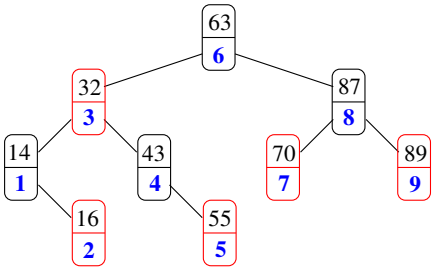
Règle 3 : rotation & recoloration

Suppression: Supprimer comme dans tout arbre binaire de recherche en allant chercher le successeur immédiat, puis suite de rotations/recolorations en remontant depuis la position du successeur immédiat vers la racine pour éliminer tout motif qui ne respecte pas la structure des arbres rouges-noirs. Détails dans le Cormen.

L"insertion, la suppression et la recherche se font en $O(\log n)$, comme sur les ABR.

Ajuster pour traiter FIND-RANK et FIND-ORDER .

→ Rajouter à chaque noeud le rang de l'élément stocké.



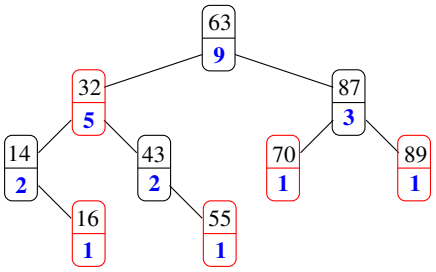
FIND: $O(\log n)$ ☺

RANK: $O(1)$ ☺ (Si on connaît position de x, sinon il faut ajouter un Find)

ORDER: $O(\log n)$ ☺ (Faire Find sur les rangs des éléments plutôt que sur les valeurs; l'arbre des rangs est un ABR!)

INSERT and DELETE: $\Omega(n)$, pas bon! Mise à jour trop coûteuse, par exemple si l'élément inséré/supprimé est le plus petit, tous les rangs doivent être changés.

→ Rajouter à chaque noeud le cardinal de son sous-arbre.



FIND: Toujours en $O(\log n)$ ☺

RANK: $O(\log n)$ ☺ Rang dans son sous-arbre: card(filsgauche) + 1; récurrence en remontant vers la racine (fils gauche, on ne fait rien; fils droit, on rajoute card(filsgauche(père))+1)

FIND-RANK(x) connaissant la position i de $x \in S$

1. $r \leftarrow card(\texttt{filsgauche}(i)) + 1$;
2. tant que $i \neq \texttt{racine}$ faire
3. — si i est un fils droit, alors
4. — $\quad r \leftarrow r + card(\texttt{filsgauche}(\texttt{père}(i))) + 1$;
5. — $i \leftarrow \texttt{père}(i)$;
6. renvoyer r

(i est fils droit = test en $O(1)$ par $\texttt{filsdroit}(\texttt{père}(i))=i$.)

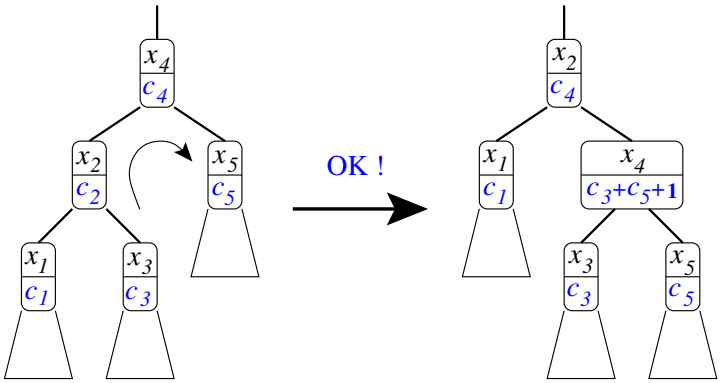
ORDER: $O(\log n)$ ☺ Algo récursif: FindOrder(k, i) renvoie l'élément de rang k du sous-arbre enraciné en i.

FIND-ORDER(k,i) version récursive à lancer avec $i = \texttt{racine}$

1. $r \leftarrow card(\texttt{filsgauche}(i)) + 1$;
2. Case $k = r$: renvoyer $valeur(i)$;
3. Case $k < r$: FIND-ORDER(k,$\texttt{filsgauche}(i)$);
4. Case $k > r$: FIND-ORDER($k - r$,$\texttt{filsdroit}(i)$);

INSERT and DELETE: $O(\log n)$ ☺ Il faut pouvoir mettre à jour les informations pour une rotation. Solution qui marche pour tout type d'ABR mis à jour par rotations (AVL, splay trees): modifier routine de rotation, et incréments/décréments de 1 lors des insertions/suppressions.

Modifier les informations de cardinal lors d'une rotation (en temps constant):



1.3.2 Treaps

Treap = Tree + Heap

Elements = couples; structure d'ABR sur la première coordonnée (filsgauche $\leq$ père $\leq$ filsdroit), et structure de tas sur la deuxième (père $\leq$ filsgauche, père $\leq$ filsdroit).

Lemme: pour n'importe quel ensemble de paires avec unicité de la deuxième coordonnée, il y a exactement un Treap.

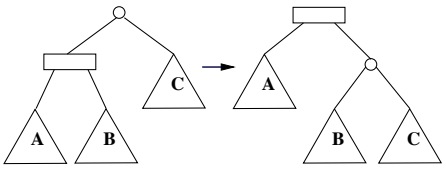
Preuve par récurrence sur le nombre d'éléments. Cas de base: Treap à un seul noeud. Sinon, le noeud avec la plus petite clé de tas doit être la racine, et grâce aux clés des arbres, on partitionne le reste en deux plus petits Treaps.

Avantage: quelque soit l'ordre des insertions, le Treap est le même.

Idée: tirer les clés de tas aléatoirement, ce qui permet d'obtenir une hauteur en $O(\log n)$ avec forte probabilité.

Un split est bon si la clé arbre de la racine partitionne entre n/4 et 3n/4 de chaque côté. Probabilité d'être bon: 1/2. Si tous les splits sont bons, $T(n) = T(3n/4) + 1 = O(\log n) = \log_{4/3} n \rightarrow$ Avec une forte probabilité, tous les deux niveaux, le sous-arbre le plus grand a été divisé par un facteur au moins 4/3, et on obtient une hauteur en $O(\log n)$.

Insertion et suppression faciles. Insertion: on tire au hasard la clé de tas. On insère comme dans un ABR, puis rotations pour corriger le tas: on fait remonter la clé fautive d'un niveau à chaque fois. Pour la suppression, on met la clé de tas à $+\infty$, puis rotations pour descendre la clé en position de feuille, ce qui permet de la supprimer.



2 Graphes

Graphe $G = (V, E)$: ensemble de sommets V et ensemble d'arêtes (ou arcs) $E \subseteq V \times V$. Version orientée ou non. On note usuellement pour un graphe fini $n = |V|$ et $m = |E|$. En cas d'ambiguité, on utilisera les notations $V(G)$ et $E(G)$.

Petites définitions classiques:

- $N(x)$: voisinage de x ($N^+(x)$ et $N^-(x)$ pour cas orienté, sortant et entrant);
- $d(x)$: degré de x ($d^+(x)$ et $d^-(x)$).

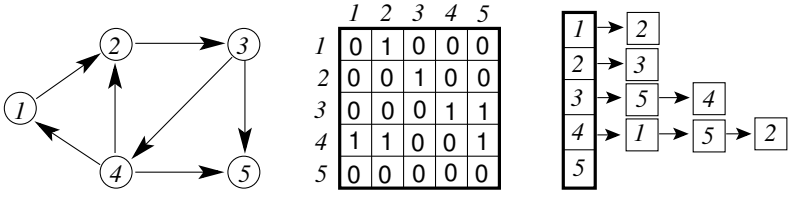
Un premier théorème (handshaking lemma): $\sum_{x \in V} d(x) = 2m$, et $\sum_{x \in V} d^+(x) = \sum_{x \in V} d^-(x) = m$.

Preuve: dans la somme $\sum_{x \in V} d(x)$, toute arête (x, y) est comptée exactement deux fois (une fois dans le terme $d(x)$, une fois dans le terme $d(y)$). En orienté, dans la somme des degrés sortants, tout arc (x, y) est compté exactement une fois (dans le terme $d^+(x)$). Idem pour les degrés entrants.

2.1 Représentation des graphes

Deux façons classiques de représenter un graphe $G = (V, E)$: ensemble de listes d'adjacences, ou matrice d'adjacences. On représente le voisinage (sortant) de chaque sommet.

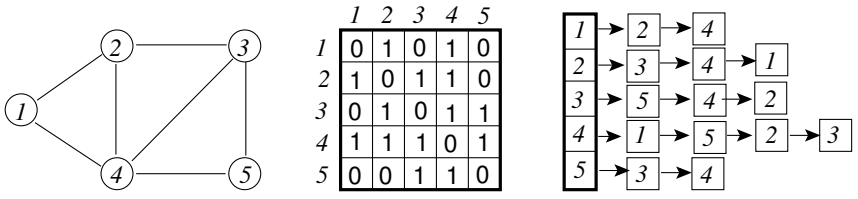
Cas orienté.



Graphe orienté — Matrice d'adjacence — Listes d'adjacence

	Matrice d'adj.	Listes d'adj. $N^+(x)$
Espace de stockage	$\Theta(n^2)$	$\Theta(n+m)$
Test d'adjacence $(x, y) \in E$	$O(1)$	$O(d^+(x))$

Cas non orienté.

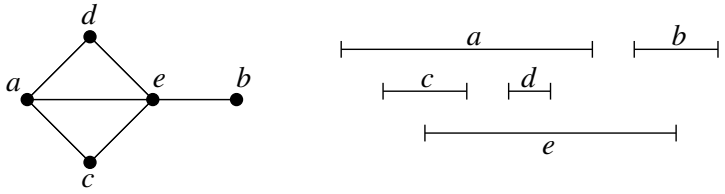


Graphe non oriente — Matrice d'adjacence — Listes d'adjacence

	Matrice d'adj.	Listes d'adj. $N(x)$
Espace de stockage	$\Theta(n^2)$	$\Theta(n+m)$
Test d'adjacence $(x, y) \in E$	$O(1)$	$O(d(x))$

Graphes pondérés. Poids associés aux arêtes/arcs. On peut rajouter des poids dans les deux types de représentation classique.

Graphes d'intervalle.



sommets = intervalles de $\mathbb{Z}$
adjacence = intersection non vide

	Graphe d'intervalles
Espace de stockage	$\Theta(n)$
Test d'adjacence $(x, y) \in E$	$O(1)$

2.2 Parcours de graphes

But: Explorer les sommets accessibles à partir d'un sommet de départ source/racine, en parcourant/traversant les arêtes/arcs du graphe (progression locale, de voisins en voisins).

- **En entrée :** graphe donné a priori par des informations locales (p.ex. listes/matrice d'adjacence), un sommet d'origine s.
- **En sortie :** un arbre de parcours enraciné en s ($pere[x]$: origine de l'arc emprunté pour aller visiter x), et éventuellement d'autres informations, comme par exemple une numérotation des sommets selon leur ordre de découverte/traitement ($id[x]$).

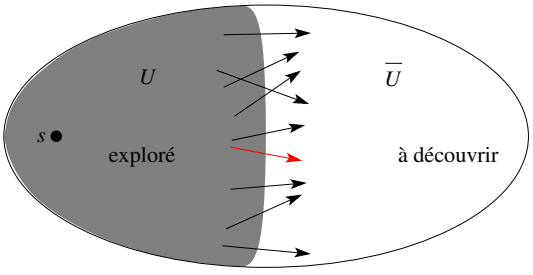
2.2.1 Schéma général

Terminologie: sommet "vu": on a vu un voisin que l'on pourra ensuite choisir de visiter; sommet "visité": on passe par ce sommet pour explorer le graphe un coup plus loin.

Coupe $[U, \overline{U}] = \{(u, v) \in E \mid u \in U, v \notin U\}$ ($= \emptyset$ si $U = V$).
Parcours(G, s) : algorithme de parcours depuis source s:

1. $U \leftarrow \{s\}$;
2. Tant que $[U, \overline{U}] \neq \emptyset$ faire : choisir $(u, v) \in [U, \overline{U}]$; $U \leftarrow U \cup \{v\}$;

Choix d'implémentation : comment gérer la coupe ? quelle SD ? quel choix d'arête/arc dans la coupe pour avancer ?



Un premier algorithme de parcours. S est un ensemble de couples de sommets qui stocke la coupe ($\perp$ = racine factice d'initialisation).

1. $S := \{(\perp, s)\}$;
2. Tant que $S \neq \emptyset$ faire
3. extraire (u, v) de S;
4. pour tout w voisin de v, insérer (v, w) dans S;

Cet algorithme ne termine pas toujours (graphe orienté avec cycles ou graphe non orienté)!

Solution: numéroter les sommets pour éviter de boucler: $id[x] \in \{1, \ldots, n\} \cup \{NIL\}$. Utilisation d'un chronomètre t.

1. $t := 0$; $id[x] := NIL$ pour tout $x \in V$; $S := \{(\perp, s)\}$;
2. Tant que $S \neq \emptyset$ faire
3. extraire (u, v) de S;
4. si $(id[v] = NIL)$ alors $\{t := t + 1$; $id[v] := t$;
5. pour tout w voisin de v, insérer (v, w) dans S;$\}$

Sommet numéroté = sommet visité. Les sommets vus sont dans S.

Finalement, on rajoute aussi les informations $pere[v]$, initialement NIL, et mise à jour $pere[v] := u$ lors de la numérotation de v ligne 4.

Proposition 2. *Cet algo de parcours calcule l'ensemble des sommets accessibles depuis s. De plus, si les extractions et insertions dans S sont en $O(1)$, sa complexité est linéaire en la taille de l'entrée, càd $O(n+m)$ pour les listes d'adjacences et $O(n^2)$ pour la matrice d'adjacences.*

Proof. ▷ Que contient S? Est-ce que c'est exactement la coupe entre les sommets numérotés/visités et les autres? Non, cette implémentation fait une *gestion paresseuse de la coupe*: à tout instant S contient la coupe mais potentiellement aussi des couples (u, v) qui ne sont pas empruntables car v est numéroté.
▷ Invariant [coupe]: à chaque fin de boucle, S = la coupe entre les sommets numérotés et les autres + des couples (u, v) tels que v est numéroté (conservation de l'invariant facile à vérifier).
▷ Invariant [arbre]: à chaque fin de boucle, le sous-graphe $(pere[v], v)$ est un arbre enraciné en s, qui couvre tous les sommets déjà numérotés (càd remonter selon *pere* depuis tout sommet numéroté ramène à l'origine du parcours s, conservation de l'invariant facile à vérifier).
▷ Terminaison & complexité:
Chaque couple (u, v) entre au plus une fois dans S, car entrée uniquement si voisinage de u parcouru, càd quand u visité, se produit à l'extraction d'un couple (w, u) avec u non numéroté, mais dans ce cas u devient numéroté $\Rightarrow$ événement se produisant au plus

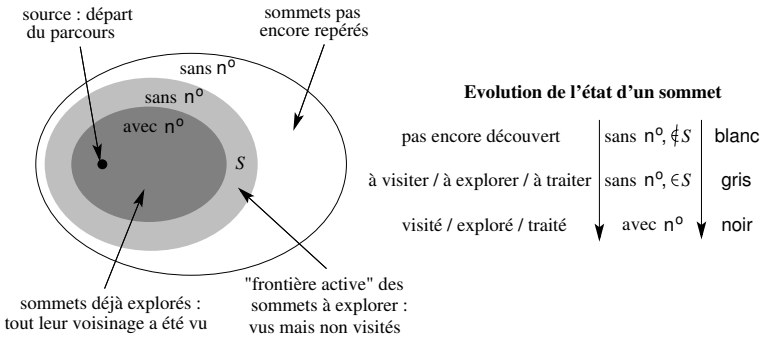
une fois. Donc, au total, nb extractions borné par $\sum_x d^+(x) = m$, resp., $\sum_x d(x) = 2m$ (idem pour nb insertions). L'algo termine forcément, avec complexité :
• coût total extractions & insertions = $O(m)$ si extractions & insertions en $O(1)$,
• coût du balayage des voisins = $O(n + m)$ pour les listes d'adj (resp $O(n^2)$ pour la matrice d'adj).
▷ Correction de l'algorithme: montrer que sommets accessibles depuis s = sommets numérotés à la fin de l'algo.
Sens $\supseteq$: corollaire de l'invariant [arbre]
Sens $\subseteq$ [par l'absurde]: supp. $\exists$ chemin P de s à x, mais x non numéroté à la fin de l'algo. Soit v le 1er sommet non numéroté de P en partant de s (donc $v \neq s$) et soit u son prédécesseur sur P (donc numéroté). Faisons un saut dans le temps : quand u a été numéroté, son voisinage a été parcouru et le couple (u, v) a été ajouté à S. L'algo termine avec $S = \emptyset$, donc (u, v) a été extrait à un moment. A cet instant, soit v n'était pas numéroté mais alors il est numéroté immédiatement, soit v s'était déjà fait numéroter (déjà visité par un autre chemin). Dans les deux cas, contradiction avec l'hypothèse v non numéroté en fin d'algo. Raisonnement par l'absurde équivalent à une récurrence sur la longueur d'un chemin (qcq) depuis s.
▷ Remarque: invariant [coupe] non utilisé dans notre preuve de correction (même si on aurait pu l'invoquer pour annoncer l'ajout de (u, v) dans S), mais utile pour justifier le fait que l'algo est bien une implémentation de notre schéma général. □

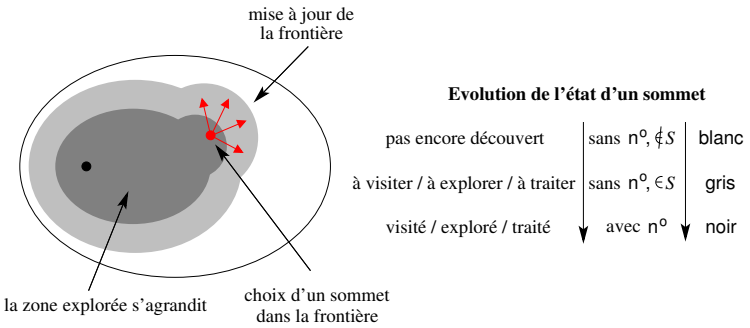
Note: Gestion *paresseuse* de la coupe. Pour avoir dans S la vraie coupe, il faut, lors de la visite de chaque nouveau sommet u,

- ne pas rajouter (u, v) tel que v déjà numéroté (rajouter un simple test);
- enlever de S les (v, u) (quittent la coupe car u visité).

On peut toujours avoir $|S|$ en $O(m)$, même si on espère être un peu mieux que la version paresseuse, et besoin de SD annexe.

Sous-jacent à tout parcours : 3 zones de sommets





Précédemment: 2 types de sommets (numérotés ou pas) + S qui donne l'information des sommets gris.

Autre approche: 3 couleurs de sommets (blanc, gris, noir). SD pour gérer efficacement extractions et insertions dans l'ensemble des sommets gris.

Parcours du graphe entier. Tant qu'il reste un sommet $s \in V$ non numéroté, faire Parcours(G, s).

2.2.2 Parcours en largeur

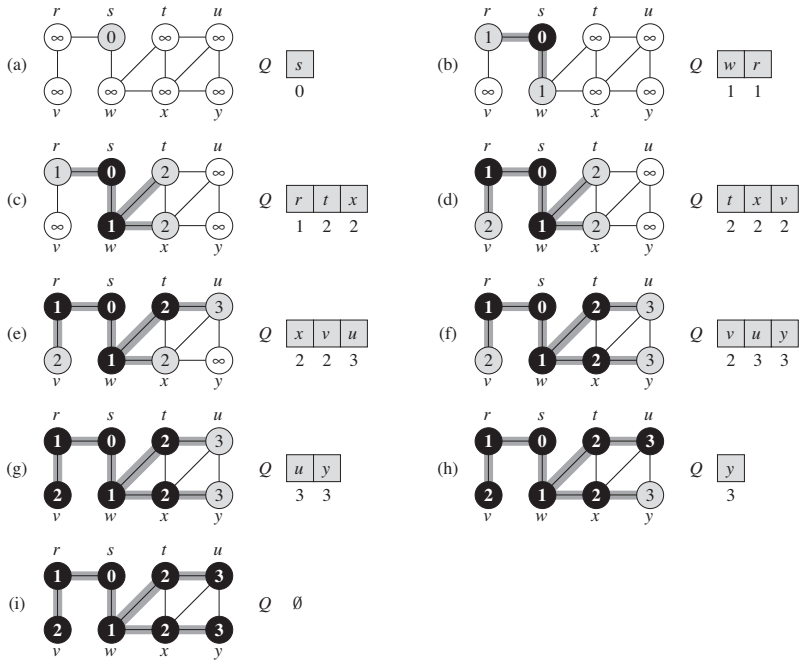
Un des algorithmes de parcours le plus simple, à la base de nombreux algorithmes importants sur les graphes (comme Dijkstra pour calculer les plus courts chemins, et Prim pour trouver l'arbre couvrant minimum, cf cours suivants).

Idée: découvrir tous les sommets situés à une distance k de s avant de découvrir tous les sommets situés à une distance $k+1$. On calcule l'arborescence de parcours avec $pere[u]$, et également les plus courts chemins avec $d[u]$, la distance entre u et s.

BFS: Breadth-First Search: la frontière S doit être gérée comme une file FIFO: on avance en extrayant toujours le sommet le plus ancien dans la coupe. Marche sur graphes orientés ou non orientés.

Initialement, $couleur[u] := blanc$, $d[u] := +\infty$, et $pere[u] := NIL$ pour tout $u \in V$, et $couleur[s] := gris$, $d[s] := 0$, $S := \{s\}$.
Algo (avec listes d'adjacence):
while $(S \neq \emptyset)$ {
  $u := defiler(S)$;
  pour tout $v \in Adj[u]$, si $couleur[v] = blanc$ {
    $couleur[v] := gris$;
    $d[v] := d[u] + 1$;
    $pere[v] := u$;
    $enfiler(S, v)$;
  }
  $couleur[u] := noir$;
}

Exemple sur un graphe non orienté:



Invariant: la file S est constituée des sommets gris: vrai initialement, puis on n'enfile que des sommets gris, et on défile un sommet avant de le colorier en noir.

Complexité linéaire (par rapport à la taille de la représentation par listes d'adjacence de G): enfiler et défiler en $O(1)$, et chaque sommet est enfilé/défilé au plus une fois (on ne recolorie jamais un sommet en blanc), d'où du $O(n)$, également pour l'initialisation. Balayage des listes d'adjacence: coût en $O(m)$ (chaque arête est examinée au plus 1 fois en orienté, 2 fois en non orienté). Au final, $O(n+m)$.

Plus courts chemins. Soit $dist(s,v)$ la distance de plus court chemin entre s et v, i.e., le nombre minimal d'arcs d'un chemin reliant s à v, ou $+\infty$ s'il n'existe pas de chemin. On peut montrer que $d[v] = dist(s,v)$ pour tout v.

De façon informelle, on voit que tous les sommets à distance 1 de s sont enfilés en premier, puis via ces sommets on atteint tous les sommets à distance 2, etc. Ainsi, on aurait une contradiction si BFS atteignait un sommet v par un chemin plus long que le plus court chemin, car le dernier sommet u sur le plus court chemin aurait été enfilé d'abord, et on aurait atteint v via u.

Formellement, la preuve suit le schéma suivant:

Lemme 1: Etant donné un graphe $G = (V, E)$ et $s \in V$, pour tout arc $(u, v) \in E$, $dist(s, v) \leq dist(s, u) + 1$. Preuve: Si u est accessible depuis s, alors v l'est aussi, et il y a un chemin de s à v qui passe par u de longueur $dist(s, u) + 1$. Sinon, $dist(s, u) = +\infty$. Dans les deux cas, l'inégalité est vérifiée.

Lemme 2 (majoration des distances): A la fin de l'exécution de BFS, pour tout sommet $v \in V$, on a $d[v] \geq dist(s, v)$. Preuve: Récurrence sur le nombre de sommets insérés dans S. Hypothèse: l'inégalité est vérifiée pour tout $v \in V$. Vrai initialement

(sommet s inséré dans S, on a $d[s] = 0 = dist(s, s)$, et $d[v] = +\infty \geq dist(s, v)$ pour tout $v \in V \setminus \{s\}$).

Récurrence: sommet v blanc découvert: successeur d'un sommet u ($(u, v) \in E$). On a donc, par hypothèse de récurrence, $d[u] \geq dist(s, u)$. On modifie $d[v]$ uniquement, et $d[v] = d[u] + 1 \geq dist(s, u) + 1 \geq dist(s, v)$ (Lemme 1). Ensuite, $d[v]$ n'est plus modifié, l'hypothèse est maintenue.

Pour prouver $d[v] = dist(s, v)$, on caractérise le comportement de la file S:

Lemme 3: Si S contient les sommets $(v_1, v_2, \ldots, v_r)$, alors $d[v_r] \leq d[v_1] + 1$ et $d[v_i] \leq d[v_{i+1}]$ pour $1 \leq i \leq r - 1$ (i.e., les valeurs de d des sommets dans S sont $a, a, \ldots, a, a + 1, a + 1, \ldots, a + 1$).

Preuve: Récurrence sur le nombre d'opérations sur la file.

Au départ, la file ne contient que s, lemme vérifié.

Défilement de v_1: nouvelle tête de file v_2. Par récurrence, $d[v_1] \leq d[v_2]$, et $d[v_r] \leq d[v_1] + 1 \leq d[v_2] + 1$, donc tout reste valable.

Enfilement de $v = v_{r+1}$: on l'a enfilé en examinant u, avec $(u, v) \in E$, et on avait déjà supprimé u de la file, donc par hypothèse de récurrence, $d[v_1] \geq d[u]$. Donc, $d[v] = d[u] + 1 \leq d[v_1] + 1$. On a aussi, par hypothèse de récurrence, $d[v_r] \leq d[u] + 1 = d[v]$, et donc le lemme est vérifié.

Corollaire: si v_i est enfilé avant v_j, alors $d[v_i] \leq d[v_j]$.

Théorème (validité de BFS): BFS découvre chaque sommet $v \in V$ accessible depuis s, et à la fin $d[v] = dist(s, v)$ pour tout $v \in V$. De plus, pour tout sommet $v \neq s$ accessible depuis s, l'un des plus courts chemins de s à v est le plus court chemin de s à $pere[v]$ complété par l'arc $(pere[v], v)$.

Preuve par l'absurde: sommet v qui reçoit $d[v] \neq dist(s, v)$, avec $dist(s, v)$ minimal. On a $v \neq s$, et $d[v] > dist(s, v) \rightarrow v$ est accessible depuis s. Soit u qui précède immédiatement v sur un plus court chemin de s à v: $d[u] = dist(s, u)$, et on a

$$d[v] > dist(s, v) = dist(s, u) + 1 = d[u] + 1.$$

Lorsque BFS défile le sommet u, v est soit blanc (et alors $d[v] = d[u] + 1$), soit noir (et alors $d[v] \leq d[u]$ car v avait été défilé avant u), soit gris (colorié lors du défilement d'un sommet w, tel que $d[w] \leq d[u]$ et $d[v] = d[w] + 1 \leq d[u] + 1$), ce qui contredit dans tous les cas l'inégalité précédente.

Tous les sommets accessibles sont découverts, sinon ils auraient un d infini et on a prouvé que $d[v] = dist(s, v)$.

Finalement, si $pere[v] = u$, alors $d[v] = d[u] + 1$, et donc on peut obtenir un plus court chemin de s à v en prenant un plus court chemin de s à u puis en traversant l'arc (u, v).

Arcs de l'arborescence de BFS = arcs de **liaison**.

2.2.3 Parcours en profondeur

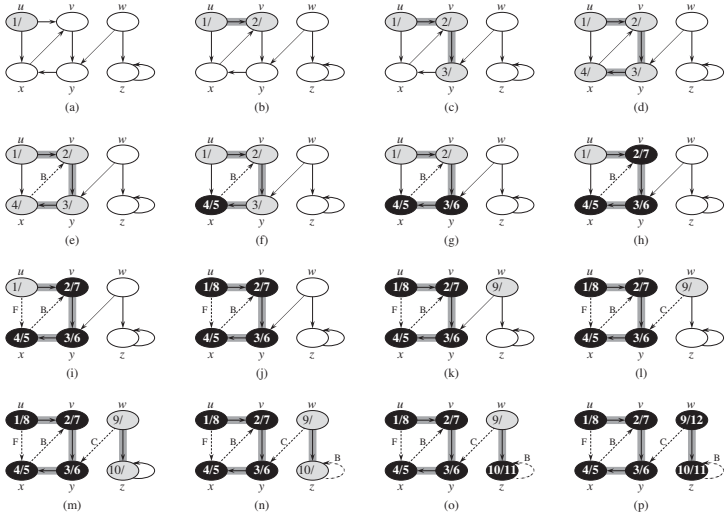
DFS: depth-first search. Idée: descendre plus profondément dans le graphe chaque fois que c'est possible. Revenir en arrière pour explorer les arcs non encore explorés. S est une pile plutôt qu'une file.

Traditionnellement, pour DFS, on répète le parcours à partir de plusieurs origines, et donc on obtient une forêt (au lieu d'un simple arbre) avec les arcs $(pere[u], u)$.

Pas de distances comme pour BFS, mais deux dates par sommet v: date de première découverte $d[v]$ (coloriage en gris du sommet), et date de fin de traitement $f[v]$ (coloriage en noir). Utilisées dans de nombreux algorithmes de graphes, et permettent l'analyse du comportement de l'algorithme. Dates: entiers entre 1 et $2|V|$ (chaque sommet a 2 dates), et $d[u] < f[u]$ pour tout $u \in V$.

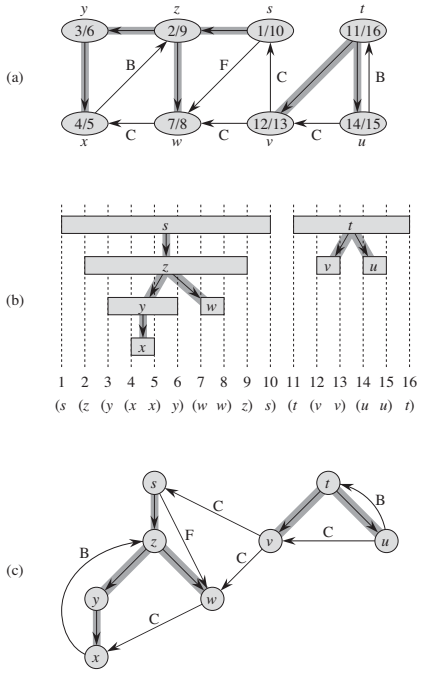
Initialement, tous les sommets sont blancs, $pere[u] := NIL$, et $t := 0$ (le chronomètre). Pour chaque sommet $u \in V$, faire si $couleur(u) = blanc$ alors Visiter-DFS(u).
Visiter-DFS(u):
$couleur[u] := gris$;
$t := t + 1; d[u] := t$;
pour tout $v \in Adj[u]$, si $(couleur[v] = blanc)$ {
    $pere[v] := u$; Visiter-DFS(v);
}
$couleur[u] := noir$;
$t := t + 1; f[u] := t$;

Exemple sur un graphe orienté:



Complexité linéaire (par rapport à la taille de la représentation par listes d'adjacence de G): $\Theta(n)$ pour les initialisations, et Visiter-DFS est appelée exactement une fois pour chaque sommet, car elle est invoquée uniquement sur les sommets blancs et commence par les colorier en gris. Balayage des listes d'adjacence en $\Theta(m)$, d'où au final du $\Theta(n + m)$.

Théorème des parenthèses. Les dates de début et de fin forment une structure correctement parenthésée: découverte = "(u", et fin = "u)", voir exemple ci-dessous:



Formellement, pour deux sommets quelconques u et v, soit les intervalles $[d[u], f[u]]$ et $[d[v], f[v]]$ sont complètement disjoints, et ni u ni v n'est un descendant de l'autre, soit $[d[u], f[u]]$ est inclus entièrement dans $[d[v], f[v]]$ et u est un descendant de v, soit le contraire (v descendant de u).

Preuve: On suppose $d[u] < d[v]$ (autre cas symétrique). Si $d[v] < f[u]$, v a été découvert pendant que u était encore gris; v est donc un descendant de u. En plus, découverte de v plus récente que celle de u, donc le traitement de v se termine avant que le parcours ne revienne à u et finisse de le traiter, donc $[d[v], f[v]]$ est inclus dans $[d[u], f[u]]$. Sinon, si $f[u] < d[v]$, les intervalles sont forcément disjoints: aucun sommet n'a été découvert pendant que l'autre était gris, et donc aucun sommet n'est un descendant de l'autre.

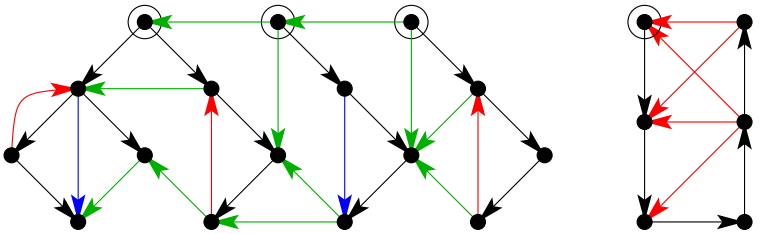
Ainsi, le sommet v est un descendant du sommet u si et seulement si $d[u] < d[v] < f[v] < f[u]$.

Théorème du chemin blanc. Théorème qui permet de caractériser les descendants d'un sommet: Un sommet v est un descendant de u ssi, au moment $d[u]$ où le parcours découvre u, il existe un chemin de u à v composé uniquement de sommets blancs.

Preuve:
$\Rightarrow$ Si v est un descendant de u, soit w un sommet sur le chemin de u à v: w est un descendant de u. Alors, $d[u] < d[w]$ et donc w est blanc à l'instant $d[u]$.
$\Leftarrow$ Si à l'instant $d[u]$ il existe un chemin blanc, mais v ne devient pas un descendant de u, supposons que tous les autres sommets deviennent un descendant de u (sinon on change v par le premier sommet que ne devient pas un descendant). Soit w le père de v sur le chemin: w est un descendant de u et $f[w] \leq f[u]$. v doit être découvert après u, mais avant que le traitement de w soit terminé, et donc $d[u] < d[v] < f[w] \leq f[u]$. D'après le théorème des parenthèses, v doit donc être un descendant de u.

Classification des arcs.

- Arc de liaison (bold): arc de la forêt de parcours. (u, v) liaison si v découvert la première fois pendant le parcours de (u, v) (v blanc);
- Arc arrière (B, rouge): relie u à un ancêtre v (v gris);
- Arc avant (F, bleu): relie u à un descendant v, mais pas un arc de liaison (v noir);
- Arc transverse (C, vert): tous les autres arcs: deux sommets d'un même arbre non descendants l'un de l'autre, ou deux sommets d'arbres différents (v noir).

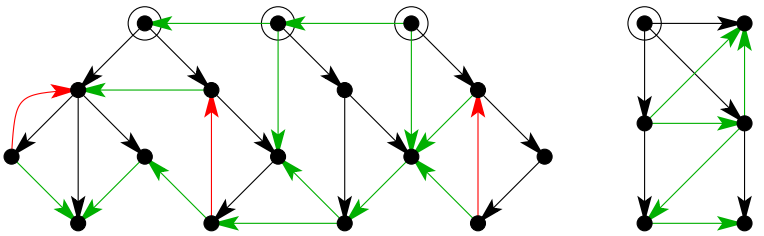


Cas non orienté: on ne retient qu'un sens pour les arêtes = sens de la première fois qu'une arête a été visitée. Uniquement des arcs de liaison et des arcs arrière.

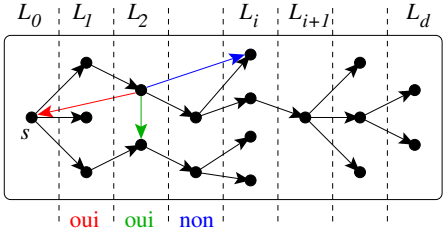
Possible de classifier les arcs à la volée ou en fin d'algorithme, en $O(1)$ (pas de surcoût à la volée). A la volée = dans DFS(u), "pour tout v voisin de u ..." (i.e., (u, v) est une arête explorée).

Test	**A la volée**	**En fin d'algo**
liaison	immédiat	$pere[v] = u$
avant	$d[u] < f[v]$	$d[u] < d[v] < f[v] < f[u]$ $pere[v] \neq u$
arrière	$d[v] \neq NIL$ $f[v] = NIL$	$d[v] < d[u] < f[u] < f[v]$
transverse	$f[v] < d[u]$	$f[v] < d[u]$

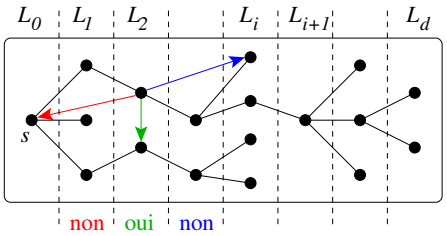
Classification en BFS ?



Uniquement des arcs de liaison, des arcs transverse et des arcs arrière en orienté:



Uniquement des arcs de liaison et des arcs transverse en non-orienté:



2.2.4 Applications des parcours

Rappel: composantes connexes, en non-orienté: ensemble de sommets accessibles entre eux. Graphe connexe: tous les sommets sont accessibles entre eux.

Cas orienté: on parle de forte connexité: u et v sont mutuellement accessibles s'il y a un chemin de u vers v et un chemin de v vers u. Composante fortement connexe = ensemble de sommets mutuellement accessibles entre eux.

1. En non-orienté.

1. G est-il connexe? Linéaire: parcours depuis source quelconque, puis vérifier que tous les sommets ont été visités.

2. Calculer les composantes connexes? Linéaire: parcourir tout le graphe, en sortie 1 arbre de la forêt = 1 composante connexe.

3. Tester si G a un cycle en $O(n)$.
   1 - Faire un parcours de tout le graphe (càd en repartant si besoin d'un nouveau sommet s'il y a plusieurs composantes connexes) et s'arrêter dès qu'on trouve une arête qui amène à un sommet déjà numéroté.
   2 - Annoncer alors qu'il existe un cycle : l'arête crée un cycle via le chemin dans l'arbre de parcours, en notant aussi que les deux extremités sont forcément dans la même comp. connexe.
   3 - Si toutes les arêtes sont vues sans que ce cas se présente, annoncer que le graphe est sans cycle : la forêt de parcours est le graphe dans sa totalité.
   Complexité : $O(n)$, car tout graphe sans cycle (forêt) a $\leq n-1$ arêtes et donc le nb d'arête vues par l'algo sans trouver de cycle est au maximum $n-1$. On pourrait d'ailleurs avoir un algo qui commence par tester si $m \leq n-1$ pour traiter rapidement une première partie des cas, mais ce test n'est pas suffisant (on pourrait avoir de nombreux sommets isolés, ce qui augmente n mais pas m, et il faut donc aller voir de plus près ce qui se passe là où il y a des arêtes).

2. En orienté.

1. G est-il fortement connexe? Linéaire: fixer un sommet s, parcourir G depuis s et vérifier que tous les sommets sont accessibles dans G depuis s, calculer G^{-1} (arcs inversés), parcourir G^{-1} depuis le même sommet s et vérifier que tous les sommets sont accessibles, autrement dit que s est accessible dans G depuis tous les sommets.

   Graphe G^{-1}: même ensemble de sommets V que G, mais toutes les arêtes sont inversées, i.e., si $(u,v) \in E(G)$, alors $(v,u) \in E(G^{-1})$.

2. Calculer les composantes fortement connexes de G orienté? $\rightarrow$ linéaire ! un peu plus difficile, cf après ...

3. Tester si G a un circuit? $\rightarrow$ linéaire ! facile en choisissant bien son parcours, cf après ...

3. Reconnaissance des DAGs. DAG = graphe orienté sans circuit.

Proposition 3. *G est un DAG ssi il n'y a aucun arc arrière dans un parcours DFS de G.*

Corollaire: on peut donc trouver un circuit et reconnaître un DAG en temps linéaire.

Solution simple de détection d'un circuit à la volée, en ajoutant au DFS récursif un test d'arc arrière:

DFS(u): routine d'exploration
1. $d[u] \leftarrow t \leftarrow t+1$;
2. pour tout v voisin de u,
Si $d[v]$=NIL, faire DFS(v);
Sinon si $f[v]$=NIL, annoncer "circuit !";
3. $f[u] \leftarrow t \leftarrow t+1$;

Attention, plein d'algos faux utilisant d'autres parcours comme BFS, ou ne travaillant pas avec des informations suffisantes pour repérer les arcs arrière du DFS.

Preuve: $\boxed{\Rightarrow}$ [contraposée] $\exists$ arc arrière $uv \Rightarrow \exists$ circuit, par déf arc arrière, en empruntant le chemin de v à u dans arbre de parcours puis (u,v).
$\boxed{\Leftarrow}$ [contraposée] Si G admet un circuit C, soit u_0 le 1er sommet de C visité par le DFS. Alors, en notant $C = u_0u_1 \dots u_\ell u_0$, l'arc $u_\ell u_0$ est forcément un arc arrière: grâce au théorème du chemin blanc, comme il existe un chemin blanc vers u_ℓ au moment où u_0 est visité (c'est le 1er à être visité), u_ℓ sera un descendant de u_0 dans leur arbre de parcours, et quand u_ℓ sera visité, l'arc $u_\ell u_0$ sera listé parmi les arcs arrière.

Preuve du corollaire (reconnaissance): Faire un DFS et repérer s'il existe un arc arrière à la volée ou a posteriori (cf classification). Si oui, calcul explicite d'un circuit à partir de cet arc arrière via la fonction *pere*[ ].

4. Tri topologique: ordre total sur V donné par `ordre`: $V \rightarrow \mathbb{Z}$ (typiquement $V \rightarrow \{1,\dots,n\}$) tel que $(u,v) \in E \Rightarrow$ `ordre`$[u] <$ `ordre`$[v]$.

Proposition 4. *G est un DAG ssi G admet un tri topologique. Dans ce cas avec un DFS,* `ordre`$[v] = -f[v]$ *fournit un tri topo en temps linéaire.*

$\boxed{\Leftarrow}$ [absurde] Si G admet un tri topo `ordre`[ ], s'il existait un circuit $C = u_0u_1 \dots u_\ell u_0$, il faudrait `ordre`$[u_0] <$ `ordre`$[u_1] < \cdots <$ `ordre`$[u_\ell] <$ `ordre`$[u_0]$, impossible.
$\boxed{\Rightarrow}$ Si G est un DAG, exhibons un tri topo : montrons que pour tout DFS, $-f[\,]$ marche. D'après la classification des arcs dans un DFS, tous les arcs (u,v) vérifient $f[v] < f[u]$, excepté les arcs arrière mais il n'y en a pas dans un DAG d'après la 1ère proposition.

Remarque: "Tout est plus simple sur les DAGs". Optimisation sur les DAGs souvent plus facile à résoudre que sur les graphes orientés quelconques. Commencer souvent par un tri topologique qui permet d'organiser la suite, p.ex. calcul glouton en balayant les sommets selon ce tri topologique.

5. Composantes fortement connexes. Algorithme de Kosaraju (1978) / Sharir (1981): sera fait en TD! Joli algorithme qui calcule les composantes fortement connexes de G en temps linéaire, en utilisant encore le graphe G^{-1} avec les arêtes inversées.

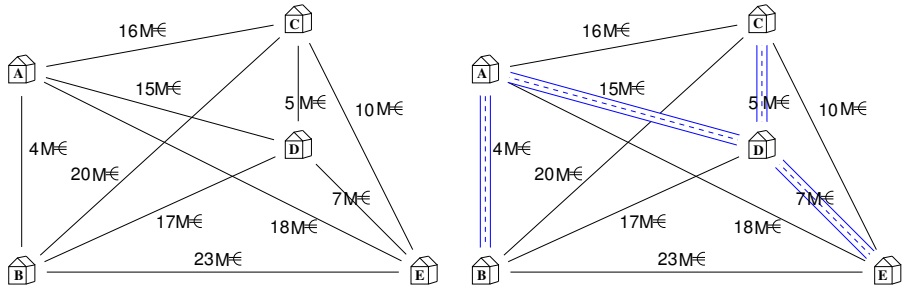
2.3 Arbres couvrants de poids min

Arbre couvrant = sous-graphe qui est un arbre et qui contient tous les sommets. On travaille sur des arbres non orientés (et non enracinés).

Proposition: G admet un arbre couvrant ssi G est connexe. Preuve: si on a un arbre couvrant, on a l'accessibilité entre toutes les paires de sommets. Si G est connexe, un parcours de graphe permet de construire un arbre couvrant tous les sommets.

Arbre couvrant de poids min (ACM):
Soit $G = (V, E)$ non orienté connexe où $\forall e \in E$, poids $w(e) \in \mathbb{R}_+$, trouver un arbre couvrant T qui minimise le poids de T: $w(T) = \sum_{e \in T} w(e)$.

Entrée: listes d'adjacence incluant les poids au niveau de chaque voisin.



Exemple: déployer un réseau routier qui relie plusieurs sites, sans embranchement en dehors de ces sites et à moindre coût.

Solution: algorithmes gloutons, et suivant les SD utilisées, on obtient des performances plus ou moins bonnes.

Nous présentons d'abord un algorithme générique, avant de détailler les deux algorithmes classiques de Kruskal et Prim.

2.3.1 Algorithme générique

On fait pousser l'ACM arête par arête. Ensemble d'arêtes A, qui conserve l'invariant suivant: avant chaque itération, A est un sous-ensemble d'un ACM.

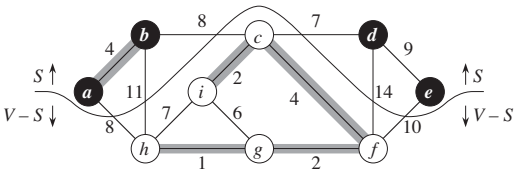
A chaque étape, on rajoute une arête (u, v) qui est **sûre** pour A, i.e., on peut l'ajouter à A sans détruire l'invariant.

Algo générique:
1. $A := \emptyset$
2. Tant que (A ne forme pas un arbre couvrant)
3. Trouver une arête (u, v) sûre pour A;
4. $A := A \cup \{(u, v)\}$;
5. Retourner A;

Comme on ne rajoute que des arêtes sûres, l'invariant est conservé et l'algorithme termine. La difficulté consiste à trouver une arête sûre en ligne 3. Il en existe vu que l'invariant impose qu'il y ait un arbre couvrant T tel que $A \subseteq T$.

Rappel de la définition d'une coupe $(S, V \setminus S)$: elle partitionne l'ensemble des sommets V (sommets vus/ non vus pour reprendre la terminologie des parcours).
• Arête qui traverse la coupe: une extrémité dans S, l'autre dans $V \setminus S$.
• Arête minimale pour la coupe: arête de poids min qui traverse la coupe.
• Une coupe respecte un ensemble d'arêtes A si aucune arête de A traverse la coupe.

Exemple de coupe qui respecte un ensemble d'arêtes en gras:



Théorème 7. *Soit $G = (V, E)$ un graphe non orienté connexe pondéré par w. Soit $A \subseteq E$ inclus dans un ACM de G, soit $(S, V \setminus S)$ une coupe de G qui respecte A, et soit (u, v) une arête minimale traversant cette coupe. Alors, (u, v) est une arête sûre pour A.*

Preuve: Soit T un ACM contenant A. Si T contient (u, v) c'est fini; on suppose donc que T ne contient pas (u, v). On sait alors que (u, v) forme un cycle dans T, et u et v sont de chaque côté de la coupe. Donc il existe au moins une arête de T qui traverse la coupe, (x, y), qui n'est pas dans A.

Soit $T' = T \setminus \{(x, y)\} \cup \{(u, v)\}$: la suppression de (x, y) coupe T en 2, et l'ajout de (u, v) le réassemble pour former un nouvel arbre couvrant. (u, v) étant minimale pour la coupe, $w(u, v) \leq w(x, y)$, et donc $w(T') \leq w(T)$. T étant un ACM, T' est également un ACM. De plus, $A \cup \{(u, v)\} \subseteq T'$, donc (u, v) est une arête sûre pour A.

Algo générique: gère une forêt d'arbres couvrants, un arbre couvrant par composante connexe. Boucle effectuée $n - 1$ fois, car le nombre d'arbres (initialement n) diminue de 1 à chaque itération. En effet, une arête sûre relie forcément deux composantes connexes distinctes.

On déduit du théorème qu'une arête minimale reliant deux composantes connexes est sûre (coupe avec les sommets d'une composante connexe).

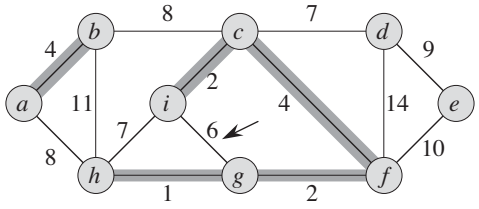
2.3.2 Algorithme de Kruskal (1956)

Idée de Kruskal:
1. Trier les arêtes selon leur poids.
2. Ajouter les arêtes une à une par poids croissant, en éliminant les arêtes qui créent un cycle avec celles précédemment ajoutées.
3. Renvoyer l'ensemble des arêtes conservées.

Clé de la complexité: choix de la SD d'union-find pour l'étape 2.

Algo: initialement $A = \emptyset$, $MakeSet(v)$ pour chaque sommet $v \in V$, et tri des arêtes par ordre croissant de poids w.
Pour chaque arête (u, v), si $FindSet(u) \neq FindSet(v)$ alors
    $A := A \cup \{(u, v)\}$;
    $Union(u, v)$;
Retourner A.

Exemple sur le graphe suivant:



Analyse. n opérations $MakeSet$, tri en $O(m \log m)$, puis au plus $2m$ opérations $FindSet$ et n opérations $Union$.

Union-Find	Listes	Arbres compressés	Arbres de Tarjan
Complexité	$O(mn)$	$O(m + n \log n)$	$O(m\alpha(m, n))$

(tri non compté)

Analyse de complexité Rien à dire : c'est de l'Union-Find pur, on récupère les complexités déjà vues. Noter qu'en ligne 2, on peut s'arrêter dès que l'on a retenu $n - 1$ arêtes, c'est conseillé en pratique mais cela ne change pas la complexité pire cas.

2.3.3 Algorithme de Jarník (1930) / Prim (1957)

Autre algorithme glouton, qui construit un arbre en choisissant à chaque fois l'arête attachée à l'arbre de poids min. Initialement, uniquement un sommet et on fait croître l'arbre: on maintient une clé pour chaque sommet et on rajoute à l'arbre le sommet de clé minimum à chaque étape.

1. Choisir une racine/source arbitraire $r \in V$;
2. $\forall u \in V \neq r$, $c[u] := +\infty$ et $\text{pere}[u] := NIL$; $c[r] := 0$;
3. $S := V$
4. Tant que $S \neq \emptyset$, faire
5. — extraire $u \in S$ de poids $c[u]$ minimum;
6. — pour tout v voisin de u dans S,
7. — si $w(u, v) < c[v]$, alors $\{c[v] := w(u, v)$; $pere[v] := u;\}$
8. Renvoyer l'arbre sous forme enracinée avec la fonction $pere[\,]$

Clé de la complexité: gestion efficace de S

- ExtraireMin et DiminuerValeur → File de Priorité p.ex. tas
- Test $y \in S$ en $O(1)$ → Dictionnaire Statique p.ex. tableau booléen

Proposition 5. *L'algo de Jarník/Prim calcule un arbre couvrant de poids min, avec une complexité qui dépend du choix de la File de Priorité :*

File de Priorité	Tableau	Tas binaire	Tas de Fibonacci
Complexité	$O(n^2)$	$O(m \log n)$	$O(m + n \log n)$

- Analyse de complexité.

  Tous les sommets vont finir par être extraits de S et chaque voisinage va être balayé une fois (et une seule). Par conséquent la ligne 6 sera exécutée $\sum_u d(u) = 2m$ fois

et, si on regarde bien, la ligne 7 ne sera exécutée que m fois à cause du test "dans S" à la ligne 6 qui implique qu'une arête ne sera examinée en ligne 7 que la 1ère fois où l'une de ses extrémités est enlevée de S. Le test d'appartenance à S est réalisé via une SD supplémentaire de dictionnaire statique : on prend ici un tableau booléen.

Nb d'opérations		Coût unitaire Tableau	Tas binaire	Tas Fibonacci
CréerFile	$\times 1$	$O(n)$	$O(n)$	$O(n)$
ExtraireMin	$\times n$	$O(n)$	$O(\log n)$	$O(\log n)$
DiminuerValeur	$\times m$	$O(1)$	$O(\log n)$	$O(1)$ amorti
Test $\in S$	$\times 2m$	$O(1)$	$O(1)$	$O(1)$
Total		$O(n^2)$	$O(m \log n)$	$O(m + n \log n)$

- Complexité de DiminuerValeur dans un tas binaire : faire remonter par échanges la valeur sur sa branche vers la racine jusqu'à ce qu'elle trouve sa bonne position, soit une complexité en $O(\log_2 n)$. Pour que DiminuerValeur s'exécute au mieux, il faut aussi idéalement avoir une SD annexe qui permet d'accèder en $O(1)$ à la position de chaque élément dans la File de Priorité (prendre p.ex. un tableau de pointeurs).

2.3.4 Variantes

Question : est-ce que les variantes suivantes sont plus compliquées ou bien se ramènent-elles facilement à la version étudiée jusqu'à présent ?

1. Trouver un arbre couvrant de poids minimum, avec poids dans $\mathbb{R}$ au lieu de $\mathbb{R}_+$ (poids négatifs autorisés).

   Il y a deux manières d'argumenter pour dire que prendre les poids dans $\mathbb{R}$ ne change rien :
   • **Par réduction :** Tous les arbres couvrants ont le même nb d'arêtes, à savoir $n-1$, donc ajouter une constante identique W à toutes les arêtes ne change pas les arbres couvrants optimaux (et la valeur optimale est juste décalée de $+(n-1)W$). Il suffit donc d'ajouter W pour que tous les poids deviennent positifs : avec $W = \max\{|w(e)| \mid w(e) < 0\}$, c'est une réduction polynomiale.
   • **Par les algorithmes classiques :** en regardant bien l'analyse de correction des algos vus en cours, on voit qu'on n'utilise pas l'hypothèse que les poids sont positifs, les algos fonctionnent donc correctement pour des poids dans $\mathbb{R}$.

2. Trouver un arbre couvrant de poids maximum, avec poids dans $\mathbb{R}$.

   Remplacer tous les poids $w(e)$ par leur opposé $-w(e)$, et chercher un arbre couvrant de poids min
   $$\max_{\text{arbre } T} \sum_{e \in T} w(e) = - \min_{\text{arbre } T} \sum_{e \in T} (-w(e))$$
   Noter que les poids sont dans $\mathbb{R} \to$ cf la variante précédente.

3. Trouver un arbre couvrant de poids minimum, où le poids de l'arbre est $\max_{e \in T} w(e)$ au lieu de $\sum_{e \in T} w(e)$.

Exo pour plus tard : "Tout arbre couvrant min $\sum$ est un arbre couvrant min max" (à démontrer en TD), donc la recherche d'arbre couvrant min $\sum$ classique suffit à résoudre cette variante. En fait on peut faire mieux : il existe un algo linéaire.

2.4 Plus courts chemins

2.4.1 Définitions

Définition: graphe orienté et fonction de poids $w : E \to \mathbb{R}$.
Poids d'un chemin $p =< v_0, v_1, \ldots, v_k >$: $w(p)$ est la somme des poids des arêtes sur le chemin:

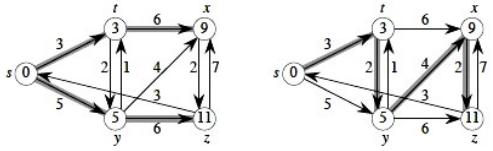
$$w(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i) \ .$$

Alors le poids d'un plus court chemin de u vers v est

$$\delta(u, v) = \begin{cases} \min\{w(p) : u \overset{p}{\rightsquigarrow} v\} & \text{s'il existe un chemin } u \rightsquigarrow v, \\ +\infty & \text{sinon.} \end{cases}$$

Un plus court chemin (PCC) de u vers v est *un* chemin tel que $w(p) = \delta(u, v)$ (pas d'unicité).

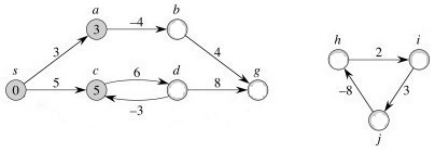
Exemple. Plus courts chemins en partant de s, les valeurs de δ sont indiquées dans les sommets.



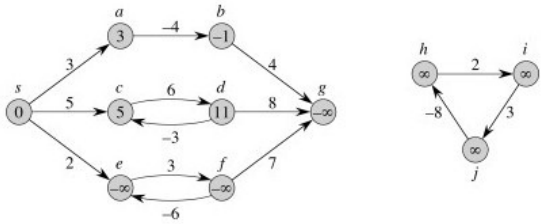
Variantes.

1. **Origine unique:** sommet origine s, on veut trouver un plus court chemin vers tout autre sommet $v \in V$ (Section 2.4.2);
2. **Destination unique:** problème identique au précédent (inverser tous les arcs);
3. **Un couple de sommets:** même complexité que pour le problème à origine unique (pourquoi?);
4. **Tous couples de sommets:** problème intéressant, peut être résolu en exécutant algo à origine unique depuis tous les sommets, ou résolution directe (voir Section 2.4.3).

Arcs de poids négatif. Ils ne sont pas gênant tant qu'il n'y a pas de circuit de poids négatif atteignable depuis la source s. Exemple à compléter:



Avec un circuit de poids négatif:



Certains algorithmes peuvent détecter des circuits de poids négatif, d'autres ne peuvent pas, mais quand il y en a, les PCC ne sont pas bien définis.

Circuits. Un PCC ne peut pas contenir de circuits:

- circuits de poids négatifs déjà éliminés;
- circuit de poids positif dans un PCC: en supprimant le circuit on obtient un chemin plus court, c'est absurde;
- circuit de poids nul: pas de modification du coût si on le supprime, donc on suppose que les solutions ne les utilisent pas.

Ainsi, tous les PCC sont de longueur au plus $n-1$.

Sous-structure optimale. Un plus court chemin entre 2 sommets contient d'autres plus courts chemins, ce qui permet d'utiliser des algos gloutons (Dijkstra, origine unique) ou de la programmation dynamique (Floyd-Warshall, tout couple de sommet). Formellement, si $< v_1, v_2, \ldots, v_k >$ est un PCC de v_1 vers v_k, alors pour tout $1 \leq i \leq j \leq k$, $p_{ij} =< v_i, v_{i+1}, \ldots, v_j >$ est un PCC de v_i à v_j. En effet, s'il existait un chemin plus court de v_i vers v_j, alors on l'emprunterait pour aller de v_1 vers v_k en passant par v_i et v_j.

2.4.2 A origine unique

Tous les algorithmes ont plein de choses en commun.

En sortie. Pour chaque sommet $v \in V$, on maintient les attributs suivants:

- $d[v]$ est une estimation de PCC: initialement, $d[v] = +\infty$, puis la valeur peut décroître lorsque l'algorithme progresse, mais on a toujours $d[v] \geq \delta(s, v)$. A la fin de l'algorithme, $d[v] = \delta(s, v)$.
- $pere[v]$ est le prédécesseur de v sur le PCC parcouru jusqu'à présent. S'il n'y a pas de prédécesseur, $pere[v] = NIL$, et à la fin on a une arborescence des PCC sur G.

Initialisation. Procédure Init-SS (single source):
Init-SS(G, s):
Pour tout $v \in V$, $d[v] := +\infty$ et $pere[v] := NIL$;
$d[s] := 0$;

Relaxation. Tous les algos utilisent la procédure de relaxation qui se pose la question: peut-on améliorer l'estimation de PCC courante pour v en passant par u et en prenant l'arc (u, v)?
Relax(u, v, w):
Si $d[v] > d[u] + w(u, v)$ alors $d[v] := d[u] + w(u, v)$ et $pere[v] := u$.

Propriétés des PCC. On suppose implicitement que le graphe est initialisé par Init-SS, puis que l'unique façon de modifier les estimations est d'utiliser Relax.

- **Inégalité triangulaire**: pour tout $(u, v) \in E$, $\delta(s, v) \leq \delta(s, u) + w(u, v)$. Preuve: considérer le cas où v est accessible depuis s et le cas où il n'est pas accessible.
- **Propriété du majorant**: on a toujours $d[v] \geq \delta(s, v)$, et une fois que $d[v]$ a atteint la valeur $\delta(s, v)$, elle ne change plus. Preuve par récurrence sur le nombre d'appels à Relax; le résultat $d[v] \geq \delta(s, v)$ est toujours vrai. Une fois la borne inférieure atteinte, la valeur n'est jamais augmentée et elle ne peut plus diminuer.
- **Propriété aucun-chemin**: s'il n'y a pas de chemin de s à v, alors on a toujours $d[v] = \delta(s, v) = +\infty$. Preuve: corollaire de la propriété du majorant.
- **Propriété de convergence**: Si $s \rightsquigarrow u \rightarrow v$ est un PCC dans G et si $d[u] = \delta(s, u)$ avant la relaxation de (u, v), alors $d[v] = \delta(s, v)$ en permanence après la relaxation. Preuve: utiliser la propriété du majorant: après le Relax, on a $d[v] \leq \delta(s, u) + w(u, v) = \delta(s, v)$.
- **Propriété de relaxation de chemin**: si $p =< v_0, v_1, \ldots, v_k >$ est un PCC de $s = v_0$ à v_k, et si les arcs de p sont relaxés dans l'ordre (v_0, v_1), (v_1, v_2), $\ldots$, (v_{k-1}, v_k), alors $d[v_k] = \delta(s, v_k)$. Cette propriété reste vraie indépendamment de tous les autres appels à Relax, même s'ils s'entremêlent avec des Relax d'arcs de p. Preuve par récurrence: après un relax, on a $d[v_i] = \delta(s, v_i)$, et cette égalité est conservée ensuite quelques soient les autres opérations Relax effectuées.
- **Propriété de sous-graphe prédécesseur**: une fois que $d[v] = \delta(s, v)$ pour tout v, le sous-graphe défini par *pere* est une arborescence de plus courts chemins de racine s. Preuve formelle un peu laborieuse, se référer au Cormen pour les détails. Intuitivement, le sous-graphe n'a pas de circuits (sinon ils seraient de poids négatif), et il est facile de voir ensuite qu'il existe exactement un chemin de s à v pour tout v. Et ce sont des PCC (preuve à faire).

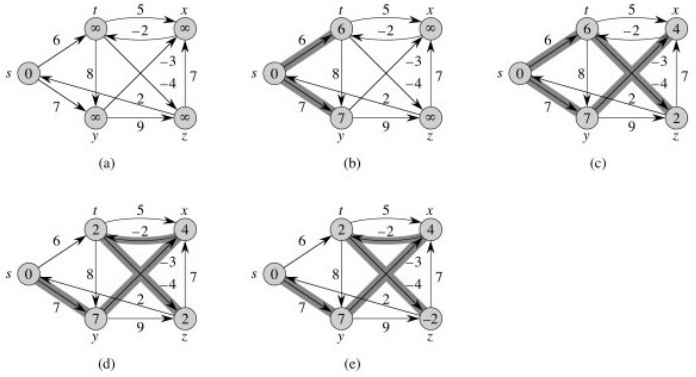
Bellman-Ford. Algorithme *brute-force* qui effectue suffisamment de Relax pour être sûr d'avoir trouvé tous les PCC. Peut être également considéré comme un algorithme de *programmation dynamique*.

Caractéristiques de Bellman-Ford: autorise les arcs de poids négatifs, calcule $d[v]$ et $pere[v]$ pour tout $v \in V$, et renvoie True (et une solution) s'il n'y a pas de circuit de poids négatif atteignable depuis s, et False sinon.

Bellman-Ford(G, w, s):
1. Init-SS(G, s);
2. Pour $i = 1$ à $n - 1$
   Pour chaque arc $(u, v) \in E$, Relax(u, v, w);
3. Pour chaque arc $(u, v) \in E$, si $d[v] > d[u] + w(u, v)$ alors return False;
4. return True;

La boucle en ligne 2 effectue les Relax et calcule les PCC. On effectue $n - 1$ passages sur tous les arcs du graphe. La boucle en ligne 3 teste la présence d'un circuit de longueur strictement négative.

Exemple: les 4 itérations de la boucle en ligne 2.



Temps d'exécution: initialisation en $O(n)$, puis chacun des $n-1$ passages de la boucle en ligne 2 prennent $O(m)$, et la dernière boucle prend $O(m)$, d'où un temps en $O(nm)$.

Validité de l'algorithme: les valeurs d et $pere$ vont converger après $n - 1$ passes, s'il n'y a pas de circuit de poids négatif. On utilise simplement la propriété de relaxation de chemin: étant donné que tous les arcs sont relaxés autant de fois que la longueur max possible d'un PCC $(n-1)$, il doit y avoir convergence. Pour un PCC $< v_0, v_1, \ldots, v_k >$, la première passe relaxe (v_0, v_1), puis la deuxième passe relaxe (v_1, v_2), ainsi de suite jusqu'à la k-ième passe qui relaxe (v_{k-1}, v_k). La propriété aucun-chemin garantit que l'algorithme renvoie une distance $+\infty$ pour les sommets non accessibles depuis s.

Algo de programmation dynamique: après la première passe, tous les PCC de longueur 1 sont corrects, et ils sont utilisés pour construire les PCC plus longs, et ainsi de suite jusqu'à ce que tous les PCC de longueur $n - 1$ soient correct.

Validité: il faut encore montrer que le résultat True/False est correct. L'idée est que si $d[v]$ continue à décroître alors qu'il aurait dû converger, alors il doit y avoir un circuit de poids négatif. Preuve: s'il n'y a pas de circuit de poids négatif, alors on a la convergence comme expliqué précédemment et la dernière boucle renvoie True. Sinon, si on avait un circuit de poids négatif $< v_0, v_1, \ldots, v_k >$ avec $v_0 = v_k$, et si l'algorithme avait renvoyé True, $d[v_i] \leq d[v_{i-1}] + w(v_{i-1}, v_i)$ pour $1 \leq i \leq k$. En sommant ces inégalités, on obtient

$$\sum_{i=1}^{k} d[v_i] \leq \sum_{i=1}^{k} d[v_{i-1}] + \sum_{i=1}^{k} w(v_{i-1}, v_i),$$

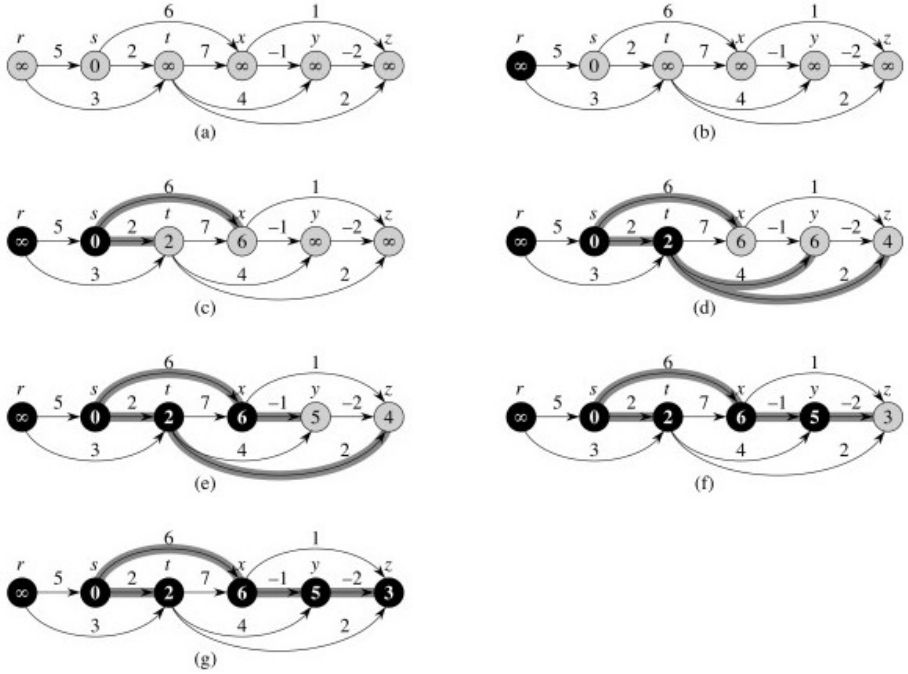
et comme $v_0 = v_k$, les deux premières sommes sont égales, et finalement le circuit doit être de poids positif, ce qui conclut la preuve.

Cas des DAGs. Rappel: la vie est toujours plus simple avec les DAGs! Pas de circuit, et donc pas de risque d'avoir un circuit de poids négatif, même s'il y a des arcs de poids négatif. On peut donc utiliser une version simplifiée de Bellman-Ford:

PCC-DAG(G, w, s):
1. Faire un tri topologique des sommets de G;
2. Init-SS(G, s);
3. Pour chaque sommet $u \in V$ pris dans l'ordre topologique,
4. Pour chaque sommet $v \in Adj[u]$, Relax(u, v, w);

Complexité: tri topologique en $\Theta(n+m)$ (parcours DFS), et c'est le terme qui domine (ligne 2 en $O(n)$, et lignes 3-4 en $O(m)$ avec une analyse par agrégat).

Exemple d'application, avec un sommet non atteignable:



S'il existe un chemin allant de u vers v, alors u précède v dans l'ordre topologique, et on n'a besoin que d'un seul passage.

Correction: de par l'utilisation du tri topologique, les arcs de chaque chemin sont forcément relaxés dans leur ordre d'apparition dans le chemin, et donc les arcs de chaque PCC sont relaxés dans l'ordre. On utilise une fois de plus la propriété de relaxation des chemins (et la propriété aucun-chemin) pour conclure.

Exemple d'application: les arcs représentent des tâches, avec leur coût, et la tâche (u, v) doit être effectuée avant (v, w). On cherche un plus long chemin, ou chemin critique, qui correspond au temps maximal requis pour effectuer une séquence ordonnée de tâches. Pour trouver un plus long chemin, prendre l'opposé des poids des arcs, ou bien modifier $+\infty$ par $-\infty$ dans Init-SS, et $>$ par $<$ dans Relax.

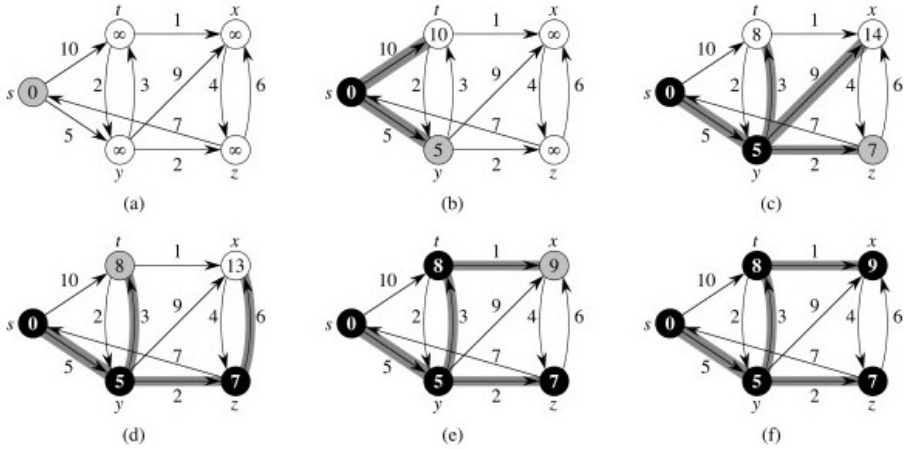
Dijkstra. Algorithme meilleur que Bellman-Ford lorsque tous les poids sont positifs ou nuls. C'est une version pondérée de BFS, mais au lieu d'utiliser une file FIFO, on utilise une file de priorité. Similarités également avec l'algorithme de Prim: algorithme glouton qui utilise les mêmes itérations.

Structures utilisées: S est un ensemble de sommets pour lesquels on connaît déjà un PCC; Q est une file de priorité contenant les sommets de $V \setminus S$. Les clés des sommets dans Q sont les estimations de PCC $d[v]$.

Dijkstra(G, w, s):
1. Init-SS(G, s);
2. $S := \emptyset$;
3. $Q := V$;
4. Tant que $Q \neq \emptyset$
5. $\quad u := ExtractMin(Q)$;
6. $\quad S := S \cup \{u\}$;
7. $\quad$ Pour chaque $v \in Adj[u]$, Relax(u, v, w);

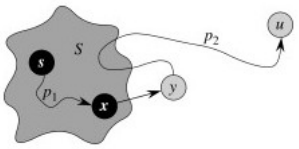
En ligne 5, on choisit le sommet u le plus proche de s que l'on rajoute dans S. La différence avec Prim, c'est que l'on choisit à chaque étape le sommet "le plus proche de s", alors que Prim choisissait "l'arc de coût minimum".

Exemple:



Correction: on considère l'invariant au démarrage de la boucle en ligne 4: $d[v] = \delta(s, v)$ pour tout $v \in S$.
Preuve: Vrai initialement car $S = \emptyset$. Il suffit ensuite de montrer que $d[u] = \delta(s, u)$ quand u est rajouté à S, puis le résultat reste vrai par la propriété du majorant.
Supposons qu'il existe u tel que $d[u] \neq \delta(s, u)$ quand u est rajouté à S. Sans perte de généralité, u est le premier tel sommet. On a:

- $u \neq s$ car s est le premier sommet rajouté à S, et $d[s] = 0 = \delta(s, s)$ à ce moment;
- il y a un chemin de s vers u, sinon $d[u] = +\infty = \delta(s, u)$;
- il y a donc un PCC p de $s \in S$ vers $u \in V \setminus S$, et donc un arc (x, y) de ce chemin qui traverse la coupe (p_1 ou p_2 peuvent ne pas avoir d'arc);



- par hypothèse, $d[x] = \delta(s, x)$ quand x est rajouté, et alors par relaxation de (x, y), on sait que $d[y] = \delta(s, y)$ lorsque u est rajouté à S. Par inégalité triangulaire et majorant, $d[y] = \delta(s, y) \leq \delta(s, u) \leq d[u]$;
- mais u a été choisi avant y dans Q, et donc $d[u] \leq d[y]$, d'où l'égalité $d[y] = \delta(s, y) = \delta(s, u) = d[u]$, qui contredit l'hypothèse $d[u] \neq \delta(s, u)$.

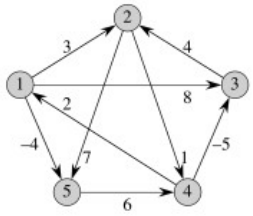
Analyse: comme pour l'algorithme de Prim, le temps d'exécution dépend de l'implémentation de la file de priorité. Avec un tas binaire, ExtractMin et DecreaseKey (qu'on doit faire dans Relax) se font en temps $O(\log n)$. La boucle en ligne 4 est appelée n fois (on extrait un sommet à chaque fois et on ne remet jamais de sommets dans Q), ce qui correspond à $O(n \log n)$ pour les ExtractMin. Le Relax est appelé $O(m)$ fois (analyse par agrégat), soit un coût total de $O(m \log n)$.

Au total, on a du $O((n+m) \log n)$, soit $O(m \log n)$ si tous les sommets sont accessibles depuis s. Avec un tas de Fibonacci, DecreaseKey ne prend qu'un temps $O(1)$ amorti, et on obtient donc $O(n \log n + m)$.

2.4.3 Pour tout couple de sommets

Objectif: trouver un PCC et le poids associé pour chaque couple $(u, v) \in V^2$.

Exemple: On doit obtenir n^2 poids (matrice de poids), et autant de chemins.



Exemples d'applications: calculs de distances entre villes, ou bien également dans des réseaux sociaux: identifier les personnes qui apparaissent dans de nombreux PCC (personnes charnières).

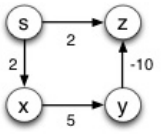
Utilisation des algorithmes à origine unique. Simplement itérer n fois les algorithmes précédents, pour toute source possible!

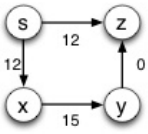
Avec Bellman-Ford: complexité de $n \times O(nm) = O(n^2 m) = O(n^4)$ sur des graphes denses. C'est coûteux mais ça marche avec des arcs de poids négatif.

Avec Dijkstra, meilleure complexité: $O(nm \log n)$ avec l'utilisation d'un tas binaire, mais pas d'arêtes de poids négatif. Ne peut-on pas se débarrasser des poids négatifs?

Comment supprimer les poids négatifs? Soustraire le poids min à tous les poids! On a ainsi rajouté le même nombre à tous les poids. Va-t-on obtenir le même résultat?

Exemple: le PCC de s à z est $s \to x \to y \to z$, mais Dijkstra défile d'abord z et trouve $s \to z$! Avec le nouveau graphe... Dijkstra trouve encore $s \to z$!

Avant: 

Après: 

Les chemins contenant beaucoup d'arcs sont pénalisés car on rajoute un coût constant à chaque arc. Idée de Johnson en 1977 qui a permis de sauver l'algorithme glouton (même si avant cela, Floyd et Warshall ont eu l'approche programmation dynamique que l'on discutera après).

L'algorithme de Johnson... ou comment se débarrasser des poids négatifs! Le but est de trouver une fonction de poids modifiée, $\hat{w}$, qui ait les propriétés suivantes:

1. Pour tout $u, v \in V$, p est un PCC de u à v en utilisant w ssi p est un PCC de u à v en utilisant $\hat{w}$;
2. Pour tout $(u, v) \in E$, $\hat{w}(u, v) \geq 0$.

On peut donc, grâce à la propriété 1, chercher les PCC pour $\hat{w}$, et grâce à la propriété 2, on peut utiliser Dijkstra.

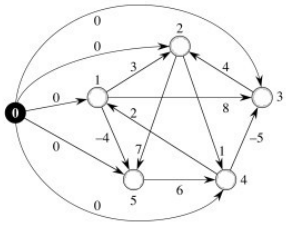
Lemme de repondération: soit $h : V \to \mathbb{R}$ une fonction quelconque. Pour chaque arc $(u, v) \in E$, on définit $\hat{w}(u, v) = w(u, v) + h(u) - h(v)$. Soit $p =< v_0, v_1, \ldots, v_k >$ un chemin de v_0 à v_k. Alors p est un PCC avec w ssi p est un PCC avec $\hat{w}$. De plus, G a un circuit de poids négatif avec w ssi G a un circuit de poids négatif avec $\hat{w}$.

Preuve: par télescopage des sommes,

$$\hat{w}(p) = \sum_{i=1}^{k} \hat{w}(v_{i-1}, v_i) = w(p) + h(v_0) - h(v_k).$$

Ce résultat est vrai pour tout chemin de v_0 à v_k, et donc si le chemin est plus court pour w, il l'est aussi pour $\hat{w}$ (et vice-versa).
Pour les circuits, on a $v_0 = v_k$ donc les circuits ont les mêmes poids avec w et $\hat{w}$.

N'importe quelle fonction h nous donne ainsi la propriété 1. Comment choisir h pour obtenir la propriété 2? On veut avoir, pour tout $(u, v) \in E$, $w(u, v) + h(u) - h(v) \geq 0$.
Idée: utiliser des PCC à origine unique dans un graphe augmenté, puis l'inégalité triangulaire assurera le résultat. On définit $G' = (V', E')$, avec $V' = V \cup \{s\}$ (s est un nouveau sommet), $E' = E \cup \{(s, v) : v \in V\}$ et $w(s, v) = 0$ pour tout $v \in V$ (on rajoute un arc de poids nul de s vers chaque sommet). Sur notre exemple:



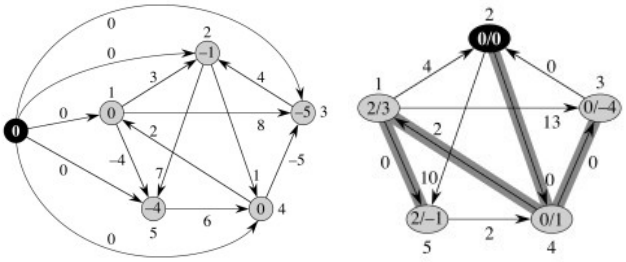
Aucune arête ne va vers s, donc G' a les mêmes circuits que G (incluant les circuits de poids négatif). On définit alors $h(v) = \delta(s, v)$ pour tout $v \in V$.

C'est ensuite facile de voir qu'on a la propriété 2: pour tout $(u, v) \in E$, par l'inégalité triangulaire, $\delta(s, v) \leq \delta(s, u) + w(u, v)$, d'où $h(v) \leq h(u) + w(u, v)$ et $\hat{w}(u, v) = w(u, v) + h(u) - h(v) \geq 0$.

On est maintenant prêt à écrire l'algorithme de Johnson:
Johnson(G, w):
1. Calculer G' (comme expliqué précédemment);
2. Si (Bellman-Ford(G', s) == False), le graphe contient un circuit de poids négatif;
3. Sinon,
4. pour chaque $v \in V$, $h(v) = \delta(s, v)$ (obtenu via Bellman-Ford);
5. pour chaque $(u, v) \in E$, $\hat{w}(u, v) = w(u, v) + h(u) - h(v)$;
6. soit $D = (d_{uv})$ une matrice $n \times n$;
7. pour chaque $u \in V$,
8. Dijkstra$(G, \hat{w}, u)$ calcule $\hat{\delta}(u, v)$ pour tout $v \in V$;
9. pour chaque sommet $v \in V$, $d_{uv} = \hat{\delta}(u, v) + h(v) - h(u)$;
10. return D;

Tout d'abord, le graphe G' est construit, puis on lance Bellman-Ford à partir de s pour voir s'il y a des circuits de poids négatifs, et s'il n'y en a pas cela retourne les $\delta(s, v)$ dont on a besoin pour calculer $h(v)$ (ligne 4). La nouvelle fonction de poids est définie ligne 5, puis on lance Dijkstra à partir de chaque sommet (lignes 7-8). On calcule les distances finales ligne 9, et le résultat est stocké dans la matrice D.

Exemple: sur la droite on indique les valeurs de $\hat{\delta}(x, v)/\delta(x, v)$ (et on part de $x = 2$ pour Dijkstra pour l'exemple).



Analyse du temps d'exécution: $\Theta(n)$ pour calculer G', $O(nm)$ pour Bellman-Ford, $\Theta(m)$ pour calculer $\hat{w}$, $\Theta(n^2)$ pour calculer D, et enfin le terme dominant est les n appels à Dijkstra, en $O(nm \log n)$ avec un tas binaire (identique à Dijkstra itéré, mais autorise les poids négatifs!). Ici encore, on peut améliorer la performance avec les tas de Fibonacci.

Programmation dynamique et multiplication de matrices. Grâce à la sous-structure optimale, on peut utiliser de la programmation dynamique: on peut étendre des solutions optimales en solutions plus longues. La première approche consiste à étendre les chemins de longueur maximum $\ell - 1$ en chemins de longueur ℓ, en passant par un sommet intermédiaire k. Très proche d'un algorithme de multiplication de matrices.

Chemin p de i vers j: $i \rightsquigarrow k \to j$, où p' est $i \rightsquigarrow k$, de longueur $\ell - 1$ (et on essaie tous les sommets intermédiaire k).

On calcule les matrices $D^{(\ell)}$ des PCC de longueur ℓ, pour $0 \leq \ell \leq n - 1$. $D^{(0)}$ a uniquement des 0 sur la diagonale, et on a

$$d_{ij}^{(\ell)} = \min\left(d_{ij}^{(\ell-1)}, \min_{1 \leq k \leq n}\{d_{ik}^{(\ell-1)} + w_{kj}\}\right) = \min_{1 \leq k \leq n}\{d_{ik}^{(\ell-1)} + w_{kj}\}$$

(car $w_{jj} = 0$).

En remplaçant le min par un + et le + par un $\times$, on obtient l'algorithme de multiplication de matrices, et on en fait $n-1$: $D^{(1)} = D^{(0)} \cdot W = W$, puis $D^{(\ell)} = D^{(\ell-1)} \cdot W = W^\ell$. On fait donc le calcul en $n \times O(n^3)$.

On veut en fait calculer une matrice $D^{(\ell)}$ avec $\ell \geq n-1$, et vu que les calculs sont associatifs, on peut le faire en $\log(n-1)$ produits pour accélérer les calculs, en mettant au carré à chaque fois. On obtient ainsi du $O(n^3 \log n)$.

Floyd-Warshall. Deuxième approche de programmation dynamique, où l'on n'étend pas simplement les chemins de longueur $\ell - 1$, ce qui permet d'obtenir une complexité en $O(n^3)$.

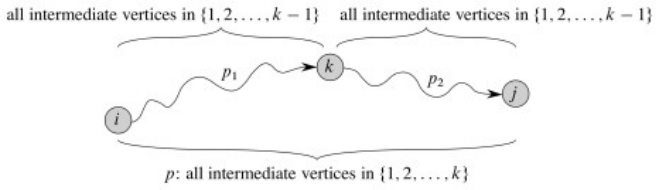
G représenté par la matrice des poids W (w_{ij} vaut 0 si $i = j$, $+\infty$ si $(i,j) \notin E$, et $w(i,j)$ sinon). L'idée est alors de formuler le problème ainsi:

- Trouver les PCC de tout i à tout j ne passant par aucun autre sommet (arêtes directes);
- Trouver les PCC de tout i à tout j soit direct, soit passant par le sommet 1;
- Trouver les PCC de tout i à tout j soit direct, soit passant par les sommets 1 et 2;
- $\cdots$
- A la fin, solutions passant par tous les sommets.

A chaque étape, on peut réutiliser ce qu'on vient de calculer, en regardant si on améliore le chemin en passant par le sommet k par rapport à la solution qui essaie de passer par 1, 2, ..., $k-1$.

Idée: un PCC de i à j n'utilisant que les sommets $\{1, \ldots, k\}$ est soit

- le PCC qui n'utilise que les sommets $\{1, \ldots, k-1\}$; ou bien
- un chemin p composé d'un PCC de i à k n'utilisant que $\{1, \ldots, k-1\}$, suivi d'un PCC de k à j n'utilisant que $\{1, \ldots, k-1\}$.



L'algorithme est donc le suivant:

FloydWarshall(W):
1. $D^{(0)} = W$;
2. for $k = 1$ to n
3. for $i = 1$ to n
4. for $j = 1$ to n
5. $d_{ij}^{(k)} = \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)})$;
6. return $D^{(n)}$;

Pour la construction des PCC, on utilise une matrice de prédécesseurs (plus simple à gérer que retrouver à partir des poids).

Exemple: $D^{(0)}$ est la matrice W. Si l'on essaie de passer par 1 (calcul de $D^{(1)}$), seuls de nouveaux chemins en partant de 4 sont trouvés (ligne 4), ce qui rajoute des 1 dans la matrice $\Pi^{(1)}$. Et ainsi de suite.

$$D^{(0)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(0)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & \text{NIL} & 4 & \text{NIL} & \text{NIL} \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(1)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(2)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(3)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(3)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(4)} = \begin{pmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(4)} = \begin{pmatrix} \text{NIL} & 1 & 4 & 2 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(5)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(5)} = \begin{pmatrix} \text{NIL} & 3 & 4 & 5 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

Le code est compact, sans structure de données élaborée, et il donne une complexité en $\Theta(n^3)$, avec une petite constante cachée. Algorithme très intéressant, y compris pour les graphes de taille modérée.

2.5 Couplages

Soit $G = (V, E)$ graphe non orienté, $M \subseteq E$ est un *couplage* si les arêtes de M n'ont pas d'extrémités en commun.
• Sommet M-*saturé*: incident à une arête de M;
• Sommet M-*insaturé*: incident à aucune arête de M;
• Couplage *maximal*: maximal pour $\subseteq$;
• Couplage *maximum*: maximal pour `card`;
• Couplage *parfait*: couplage qui sature tous les sommets;
• **Notation :** $\alpha'(G)$ = card max d'un couplage de G.

Tout couplage maximum est clairement maximal, mais est-ce que tout couplage maximal est maximum? → Non, prendre une arête sur deux, en ne commençant pas par la première, sur un chemin avec un nb impair d'arêtes 

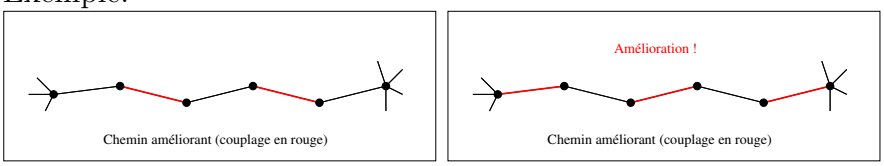
Est-ce que tout graphe admet un couplage parfait? et tout graphe avec un nombre pair de sommets? → Non, prendre par exemple les étoiles $K_{1,n}$.

2.5.1 Théorème de Berge

Définitions : soit M couplage d'un graphe $G = (V, E)$,
• Chemin M-*alternant* : alterne entre arête de M et arête de $E \setminus M$;
• Chemin M-*améliorant* : M-alternant avec extrémités M-insaturées.

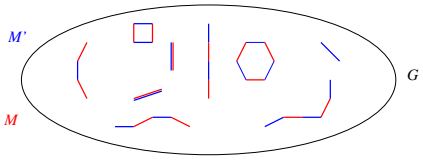
Théorème 8 (Berge 1957)**.** *Soit $G = (V, E)$ graphe non orienté, un couplage est maximum ssi il n'admet pas de chemin améliorant.*

Exemple:



Remarque: chemin alternant $\Longrightarrow$ simple.

Preuve:
$\boxed{\Rightarrow}$ Contraposée (sens facile): soit M un couplage admettant un chemin M-améliorant P. Echanger les arêtes de M avec celles de $E \setminus M$ sur P donne un nouveau couplage (car ne crée aucun conflit sur les sommets intermédiaires et aux extremités) de card $|M| + 1$.
$\boxed{\Leftarrow}$ Contraposée: soit M un couplage qui n'est *pas de card maximum*, montrons qu'il existe un chemin M-améliorant. Considérer M' un couplage *de card maximum* et focaliser son attention sur les arêtes de M et M'. Plus précisément, regarder $F = M \Delta M' = (M \cup M') \setminus (M \cap M')$ (c'est la *différence symétrique* entre deux graphes avec le même ensemble de sommets = graphe avec les arêtes qui apparaissent exactement dans l'un des graphes). Quelles formes voit-on apparaître ? → chaque sommet a au plus 1 arête incidente dans M et au plus 1 arête incidente dans M', le degré de chaque sommet est au plus 2.



Bestiaire des composantes connexes possibles dans le sous-graphe $(V, M\Delta M')$:
• cycles alternant arête de M et M' ($\Rightarrow$ cycle pair),
• chemins (non fermé) alternant arête de M et M',
• sommets isolés.

Comme $|M'| > |M|$ par hypothèse, il existe au moins une composante connexe C de $M\Delta M'$, où nb arêtes de $M' >$ nb arêtes de M. D'après le bestiaire, c'est forcément un chemin (non fermé) alternant entre M' et M, commençant et terminant par une arête de M'. Regardons ce chemin dans le graphe initial: c'est un chemin M-améliorant (extrémités M-insaturées car choix du chemin $\Rightarrow$ pas d'arête incidente dans $M \setminus M'$, et M' couplage $\Rightarrow$ pas d'arête incidente dans $M' \cap M$) !

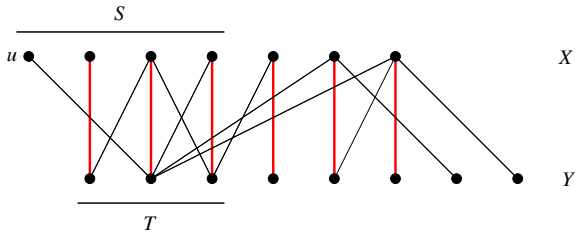
2.5.2 Théorème de Hall

Théorème 9 (Hall 1935)**.** *Soit $G = (V, E)$ graphe biparti entre X et Y, alors il existe un couplage saturant X ssi $\forall S \subseteq X$, $|N(S)| \geq |S|$.*

Théorème inefficace algorithmiquement (critère exponentiel) mais bon certificat de non-existence de couplage saturant X.

Preuve.
$\boxed{\Rightarrow}$ Sens facile: soit M couplage saturant X, $N(S) \supseteq \{y \in Y | x \in S,\ (x, y) \in M\}$ de card $|S|$ car en bijection avec S via M.

$\boxed{\Leftarrow}$ Contraposée: supposons qu'il n'existe pas de couplage saturant X.
Considérer M un couplage maximum, et $u \in X$ insaturé puisque M ne sature pas X. On va essayer de construire $S \subseteq X$ tel que $|N(S)| < |S|$ à partir de M et u.
Posons $S = \{x \in X \mid x$ atteignable depuis u par un chemin M-alternant$\}$
$T = \{y \in Y \mid y$ atteignable depuis u par un chemin M-alternant$\}$

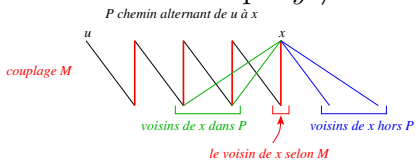


construction d'un $|S| > |N(S)|$ via un couplage maximum ne saturant pas X

▷ Montrons que M est un couplage entre T et $S \setminus \{u\}$. Les chemins M-alternants depuis u arrivent en Y via des arêtes $\notin M$ et retournent en X via des arêtes $\in M$. Ainsi, tous les sommets de $S \setminus \{u\}$ sont atteints par une arête de M venant de T. De plus, pas de chemins augmentants (couplage maximum), donc chaque sommet de T est saturé (le chemin ne se termine pas par une arête $\notin M$): si $y \in T$ est atteint, il y a une arête $(y, x) \in M$ avec $x \in S$. Il y a donc bijection et $|T| = |S \setminus \{u\}|$.

▷ De par le couplage entre T et $S \setminus \{u\}$, on a $T \subseteq N(S)$. En fait, montrons que $T = N(S)$. Si $y \in Y \setminus T$ a un voisin $x \in S$, alors l'arête (x, y) ne peut pas être dans M: u

n'est pas saturé et les autres sommets de S sont déjà couplés aux sommets de T. Ainsi, en rajoutant (x, y) à un chemin qui atteint x, on atteint y via un chemin alternant, ce qui contredit le fait que $y \notin T$: les voisins de tout $x \in S$ sont forcément dans T.



Avec $T = N(S)$, on a montré que $|N(S)| = |T| = |S| - 1 < |S|$ pour ce choix de S, ce qui conclut la preuve.

Si les deux parties ont le même cardinal ($|X| = |Y|$), le théorème de Hall est équivalent au théorème du mariage, prouvé en 1917 par Frobenius: s'il y a n hommes et n femmes, et si chaque homme (resp. femme) est compatible avec k femmes (resp. hommes), alors il existe un couplage parfait.

Corollaire: pour $k > 0$, tout graphe biparti régulier de degré k admet un couplage parfait.

Preuve: en comptant les arêtes incidentes aux sommets de X et celles incidentes aux sommets de Y, on obtient $k|X| = k|Y|$, soit $|X| = |Y|$. Il suffit donc de vérifier la condition de Hall car un couplage saturant X saturera également Y.

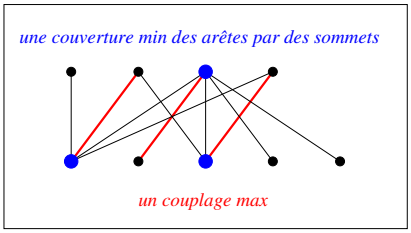
Soit $S \subseteq X$, et soit m le nombre d'arêtes de S vers $N(S)$. On a $m = k|S|$. Toutes ces arêtes arrivent dans $N(S)$, donc $m \leq k|N(S)|$, et donc $N(S) \geq S$ pour $k > 0$, et cela est vrai pour tout S (d'où la condition de Hall vérifiée).

2.5.3 Théorème de König/Egerváry

Si G n'admet pas de couplage parfait, le théorème de Berge nous permet de montrer qu'un couplage M est maximum s'il n'y a pas de chemin M-améliorant. Mais explorer tous les chemins M-alternant peut prendre du temps. Structure explicite dans G qui empêche un couplage plus grand que M?

Idée: établir une relation min-max: théorème de la forme $\max valeur(\texttt{machin}) = \min valeur(\texttt{truc})$. Ainsi, si on calcule machin et truc et si on obtient la même valeur, on a prouvé l'optimalité des deux.

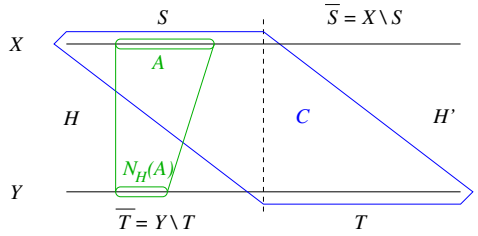
Pour les couplages, on considère le problème de couverture des arêtes par des sommets (choisir le nombre minimum de sommets pour couvrir toutes les arêtes):



Théorème 10 (König/Egerváry 1931)**.** *Dans un graphe biparti, max |couplage| = min |couverture| (i.e., $\alpha'(G) = \beta(G)$).*

Ne tient pas pour un graphe quelconque, comme par exemple sur un cycle impair: avec 5 sommets, le couplage max est de taille 2, mais il faut au moins 3 sommets pour couvrir les 5 arêtes.

Preuve de König/Egerváry:
$\boxed{\leq}$ Sens facile: étant donné un couplage de taille M, il faut utiliser un sommet distinct pour couvrir chaque arête du couplage, donc toute couverture vérifie $|C| \geq |M|$.
$\boxed{\geq}$ Pour montrer l'égalité, on part d'une couverture de taille min C, et on construit un couplage de taille $|C|$. Soit $S = C \cap X$ et $T = C \cap Y$.



Où y a-t-il des arêtes sur le schéma ? $\rightarrow$ pas entre $\overline{S}$ et $\overline{T}$ (car sinon pas couvert par C).
Notons H le sous-graphe induit par $S \cup \overline{T}$ et H' le sous-graphe induit par $\overline{S} \cup T$.
Comme $|C| = |S| + |T|$, pour montrer le résultat, il suffit de trouver un couplage dans H de card $|S|$ et un autre dans H' de card $|T|$. Regardons sur H (le même raisonnement s'appliquera à H').
Utilisation de Hall : $\forall A \subseteq S$, $|N_H(A)| \geq |A|$? où $N_H(A)$ voisinage de A dans H. [Par l'absurde] Supposons $\exists A \subseteq S$, $|N_H(A)| < |A|$. Considérer $C' = T \cup (S \setminus A) \cup N_H(A)$ (remplacer A par $N_H(A)$). Cela reste une couverture (regarder où sont les arêtes du biparti), mais de cardinal $< |C|$ (contradiction avec C minimum). Critère de Hall vérifié $\Rightarrow \exists$ couplage saturant S dans H. Idem pour H' avec T.

2.5.4 Couplage dans des graphes bipartis

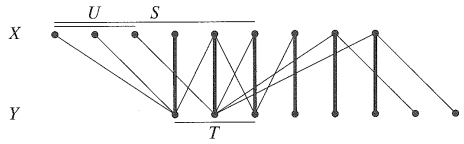
L'algorithme par **augmentation de chemin** est utilisé pour trouver un couplage maximum dans un graphe biparti. Il fournit au passage une preuve algorithmique du théorème de König/Egerváry.

Etant donné un couplage M dans un graphe biparti X, Y, on cherche un chemin M-améliorant pour chaque sommet de X qui est M-insaturé. Les chemins améliorant sont de longueur impaire, donc l'une des extrémités est dans X (pas la peine de chercher depuis Y).

En partant d'un couplage de taille 0, $\alpha'(G)$ applications de l'algorithme produisent un couplage maximum.

Augmenting Path Algorithm. Entrée: graphe biparti, couplage M, ensemble U de sommets non saturés dans X.
$S \subseteq X$ et $T \subseteq Y$ sont les sommets atteints lors de la recherche de chemins M-améliorants.
On marque les sommets de S déjà explorés, et on mémorise le prédécesseur.
Initialisation: $S = U$ et $T = \emptyset$.
Itération: Si tous les sommets de S sont marqués, $T \cup (X \setminus S)$ est une couverture minimum et M est un couplage maximum.
Sinon, on choisit $x \in S$ qui n'est pas marqué. Pour explorer depuis x, on considère chaque $y \in N(x)$ tel que $xy \notin M$. Si y est M-insaturé, on a trouvé un chemin M-améliorant de U vers y et on retourne le chemin. Sinon, y est couplé à un $w \in X$ (i.e., $yw \in M$). Dans

ce cas, on rajoute y à T (atteint depuis x) et on rajoute w à S (atteint depuis y). Après avoir exploré toutes les arêtes partant de x, on marque x et on itère.



A noter, en explorant depuis x, on peut retomber sur un $y \in T$ déjà atteint, ce qui va changer le chemin M-améliorant trouvé, mais ne changera pas l'existence d'un chemin M-améliorant.

Théorème 11. *En répétant l'algorithme d'amélioration de chemins sur un graphe biparti, on obtient un couplage et une couverture par sommets de même taille.*

Preuve: Montrons que l'algorithme renvoie soit un chemin M-améliorant, soit une couverture par sommets de taille $|M|$. Dans le premier cas, on a terminé (on itère l'algorithme). Sinon, tous les sommets de S sont marqués et l'algorithme affirme que $Q = T \cup (X \setminus S)$ est une couverture de taille $|M|$. Montrons que Q est une couverture, et qu'elle est de taille $|M|$.

Pour montrer que c'est une couverture, on montre qu'il n'y a pas d'arêtes entre S et $Y \setminus T$.

• Un chemin M-améliorant partant de U ne revient dans X que par des arêtes de M, donc tout sommet $x \in S \setminus U$ est couplé à un sommet de T, et il n'y a pas d'arête de M de S vers $Y \setminus T$ (sommets de U M-insaturés).

• Lorsqu'un chemin atteint $x \in S$, il peut se prolonger par n'importe quelle arête $\notin M$, et l'exploration de x rajoute tous les voisins de x dans T. Ainsi, toutes les arêtes vont de S vers T et sont couvertes par T.

• Les autres arêtes sont couvertes par $X \setminus S$.

Montrons maintenant que la couverture est de taille $|M|$.

• Seuls des sommets M-saturés sont mis dans T, et couplés à des sommets de S.

• $U \subseteq S$, donc tous les sommets de $X \setminus S$ sont également M-saturés, et les arêtes qui en partent ne peuvent pas aller vers T. On a donc au moins $|T| + |X \setminus S|$ arêtes dans le couplage. Vu qu'il n'y a pas de couplage plus grand que cette couverture par sommets (sinon ce n'est pas une couverture), on a l'égalité:

$$|M| = |T| + |X \setminus S| = |Q|.$$

Complexité de cet algorithme. On a $\alpha'(G) \leq n/2$, donc on va trouver un couplage maximum au bout d'au plus $n/2$ appels à l'algorithme. Un appel effectue une exploration d'un sommet de X au plus une fois (avant de marquer le sommet), et donc chaque arête est explorée au plus une fois. Le temps d'exploration d'une arête étant constant, l'algorithme est en $O(nm)$.

2.5.5 Couplage dans des graphes bipartis pondérés

Généralisation des résultats précédents sur des graphes pondérés: on cherche un couplage de poids maximum. On peut considérer un graphe complet avec des poids 0 sur les arêtes manquantes. Tous les poids sont positifs ou nuls.

On suppose $|X| = |Y| = n$, et on cherche donc un couplage parfait de poids maximum. On va aussi résoudre un problème dual de couverture.

Exemple. n fermes $X = \{x_1, \ldots, x_n\}$ et n usines $Y = \{y_1, \ldots, y_n\}$ pour transformer le maïs. Chaque ferme peut envoyer sa production à une seule usine. Si la ferme x_i envoie sa production à l'usine y_j, elle obtient un bénéfice de $w_{i,j}$ (le poids sur l'arête $x_i y_j$). Pour obtenir un profit maximum, il faut choisir un couplage parfait de poids maximum.
Le problème dual est le suivant. Le gouvernement trouve que trop de maïs est produit, et paye la compagnie pour ne plus produire. Il payera u_i si la ferme x_i n'est plus utilisée, et v_j si l'usine y_j n'est plus utilisée. Si $u_i + v_j < w_{i,j}$, la compagnie préfère continuer à produire. Pour arrêter la production, le gouvernement doit donc proposer des gains tels que $u_i + v_j \geq w_{i,j}$ pour tout i, j, et le coût sera minimisé si l'on minimise $\sum u_i + \sum v_j$.

Definitions. On peut représenter les poids sous forme d'une matrice $n \times n$, et le problème d'allocation (*assignment problem*), équivalent au problème de couplage de poids maximum, consiste à trouver une **transversale** à la matrice de poids maximum: n positions, une dans chaque ligne et une dans chaque colonne. Le poids d'un couplage, que l'on cherche à maximiser, est noté $w(M)$.
Une couverture (pondérée) est un choix d'étiquettes $u_1, \ldots, u_n$ et $v_1, \ldots, v_n$ telles que $u_i + v_j \geq w_{i,j}$ pour tout i, j. Le coût d'une couverture est $c(u, v) = \sum u_i + \sum v_j$. Le problème consiste à trouver une couverture de coût minimum.

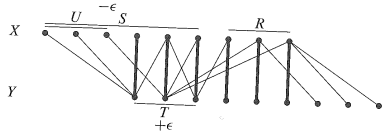
On a la dualité entre les deux problèmes: pour un couplage parfait M et une couverture $c(u, v)$, on a $c(u, v) \geq w(M)$. De plus, $c(u, v) = w(M)$ ssi M ne contient que des arêtes telles que $u_i + v_j = w_{i,j}$. Dans ce cas, M et (u, v) sont optimaux.

Preuve: M sature tous les sommets, et en sommant les contraintes on obtient $c(u, v) \geq w(M)$ pour toute couverture. De plus, s'il y a égalité, il doit y avoir égalité sur chaque contrainte. Finalement, comme $c(u, v) \geq w(M)$ pour toute couverture et tout couplage, l'égalité implique qu'il n'y a pas de couplage de poids plus grand que $c(u, v)$, et qu'il n'y a pas de couverture de coût plus petit que $w(M)$.

Sous-graphe des égalités et surcoût. On définit le sous-graphe $G_{u,v}$ pour une couverture (u, v) par le sous-graphe de G qui ne contient que les arêtes telles que $u_i + v_j = w_{i,j}$. Dans une couverture, le surcoût pour (i, j) est $u_i + v_j - w_{i,j}$.

Hungarian algorithm. Kuhn[1955], Munkres[1957]. Si $G_{u,v}$ a un couplage parfait, alors il est optimal d'après le résultat de dualité et on a fini. Sinon, on cherche un couplage maximum M et une couverture Q de même taille, par exemple avec l'algorithme par augmentation de chemins vu précédemment.

Soit $R = Q \cap X$, et $T = Q \cap Y$. Le couplage possède $|R|$ arêtes de R vers $Y \setminus T$, et $|T|$ arêtes de T vers $X \setminus R$.



Pour obtenir un couplage plus grand dans le sous-graphe des égalités, on veut modifier (u, v) pour rajouter une arête de $X \setminus R$ vers $Y \setminus T$ tout en maintenant l'égalité sur les arêtes de M.

• Toutes les arêtes entre $X \setminus R$ et $Y \setminus T$ ne sont pas dans $G_{u,v}$ (sinon elles seraient couvertes par Q), et donc elles ont un surcoût strictement positif. Soit ε le surcoût minimum parmi toutes ces arêtes: $\varepsilon = \min\{u_i + v_j - w_{i,j} \mid x_i \in X \setminus R \text{ et } y_j \in Y \setminus T\}$.

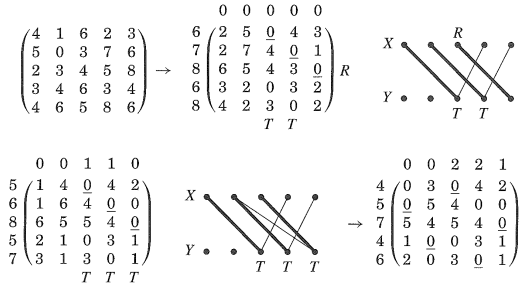
• On diminue u_i de ε pour tout $x_i \in X \setminus R$, ce qui maintient la condition pour toutes ces arêtes, tout en introduisant une de ces arêtes dans le nouveau sous-graphe des égalités.

• Afin de maintenir la propriété de couverture sur les arêtes de $X \setminus R$ vers T, on incrémente également v_j par ε pour $y_j \in T$.

La procédure est répétée, jusqu'à obtention d'un couplage parfait. C'est l'algorithme hongrois, nommé ainsi par Kuhn en honneur des travaux de König/Egerváry sur lesquels il se base.

Initialement, on fixe $u_i = \max_{1 \le j \le n} w_{i,j}$ et $v_j = 0$.

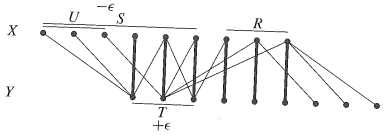
Exemple. On fait tourner l'algorithme avec une représentation par matrices. Première matrice: les poids, puis matrices des surcoûts. Les entrées soulignées dénotent le couplage choisi. Les 0 correspondent à des arêtes dans le sous-graphe des égalités. Première itération: on a réduit le coût de la couverture, mais on n'a pas changé le couplage. Après la deuxième itération, on obtient un couplage parfait de poids 31, qui est égal au poids de la couverture. On aurait convergé immédiatement si on avait pris comme couverture les 3 dernières colonnes à la première itération.



Théorème 12. *L'algorithme hongrois trouve un couplage de poids maximum et une couverture de coût minimum.*

Preuve: Au démarrage, l'algorithme utilise une couverture, et il ne peut se terminer que lorsqu'il obtient un couplage parfait dans $G_{u,v}$, ce qui garantit l'égalité entre le couplage et la couverture. Soit (u, v) la couverture courante, et supposons qu'il n'a pas de couplage parfait. Soit (u', v') la nouvelle couverture. On a $\varepsilon > 0$ car c'est le minimum d'un ensemble fini non vide de nombres strictement positifs.

Vérifions que (u', v') est une couverture. Pour les arêtes de $X \setminus R$ à T ou de R à $Y \setminus T$, on a $u'_i + v'_j = u_i + v_j$. Si $x_i \in R$ et $y_j \in T$, $u'_i + v'_j = u_i + v_j + \varepsilon$ et donc le poids est toujours couvert. Finalement, si $x_i \in X \setminus R$ et $y_j \in Y \setminus T$, alors $u'_i + v'_j = u_i + v_j - \varepsilon$, mais le choix de ε est tel que ce soit au moins $w_{i,j}$.



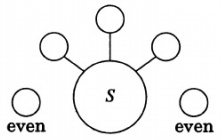
La terminaison a lieu lorsqu'un couplage parfait est trouvé dans $G_{u,v}$, donc il suffit de montrer qu'il termine. En supposant les poids entiers (s'ils sont rationnels, on peut tous les multiplier pour obtenir des entiers), chaque surcoût est entier et à chaque itération on réduit le surcoût d'une valeur entière, il y aura donc un nombre fini d'itérations.

2.5.6 Couplage dans des graphes généraux

Commençons par quelques définitions, étant donné un graphe G non orienté.
• **Sous-graphe couvrant**: sous-graphe qui inclut tous les sommets de G (pas nécessairement connexe, pas nécessairement un arbre).
• **k-facteur**: sous-graphe couvrant régulier de degré k.
• **Composante impaire**: composante connexe avec un nombre impair de sommets; $o(H)$ est le nombre de composantes impaires de H (*odd components*).

Un couplage parfait est un 1-facteur, sauf que le couplage n'est représenté que par les arêtes du graphe. Un graphe 3-régulier qui a un couplage parfait peut être décomposé en un 1-facteur et un 2-facteur.

Theorème de Tutte des 1-facteurs. Tutte a trouvé une condition nécessaire et suffisante pour laquelle un graphe admet un 1-facteur. Soit $S \subseteq V(G)$. Alors, toute composante impaire de $G \setminus S$ doit avoir un sommet relié à un sommet à l'extérieur, qui ne peut être que dans S. Tous ces sommets doivent être distincts, et donc $o(G \setminus S) \leq |S|$.



Condition de Tutte: pour tout $S \subseteq V(G)$, $o(G \setminus S) \leq |S|$.

Théorème 13 (Tutte 1947)**.** *Un graphe a un* 1*-facteur ssi* $o(G \setminus S) \leq |S|$ *pour tout* $S \subseteq V(G)$.

Preuve de Lovász [1975] (plusieurs preuves sont connues), basée sur la différence symétrique et les extrémités. Condition nécessaire: déjà énoncée ci-dessus (les composantes impaires de $G \setminus S$ doivent avoir un sommet couplé à des sommets distincts de S).

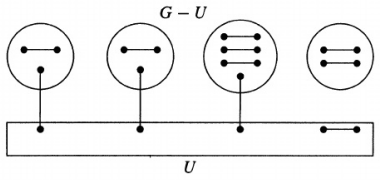
Pour la condition suffisante, on remarque que la condition de Tutte est préservée si l'on rajoute une arête à un graphe G: si $G' = G + e$, joindre 2 composantes de même parité crée une composante paire, et composante impaire + composante paire devient une composante impaire, donc le nombre de composantes impaires ne peut pas augmenter. Ainsi, si $S \subseteq V(G)$, alors $o(G' \setminus S) \leq o(G \setminus S) \leq |S|$. Aussi, si $G' = G + e$ n'a pas de 1-facteur, alors G n'en a pas non plus.

Ainsi, le théorème est vrai à moins qu'il existe un graphe G qui satisfait la condition de Tutte, qui n'a pas de 1-facteur, et auquel l'ajout d'une arête produit un graphe avec un 1-facteur. Soit G un tel graphe. On montre que G possède un 1-facteur, obtenant ainsi une contradiction.

Soit U l'ensemble des sommets de degré $|G| - 1$: ils forment une clique.

Cas 1: $G \setminus U$ est constitué de graphes complets disjoints. Alors on peut coupler les sommets dans chaque composante comme on le veut, avec un sommet non couplé dans

les composantes impaires. Chaque sommet de U est voisin avec tous les sommets de G, et $o(G \setminus U) \leq |U|$, donc on peut coupler les sommets restants avec ceux de U. Pour compléter le 1-facteur, il reste à coupler les sommets restant de U, et donc montrer qu'il reste un nombre pair de sommets. On a déjà couplé un nombre pair de sommets, donc il suffit de montrer que $|G|$ est pair. Cela découle de la condition de Tutte avec $S = \emptyset$: un graphe avec un nombre impair de sommets a forcément une composante impaire.

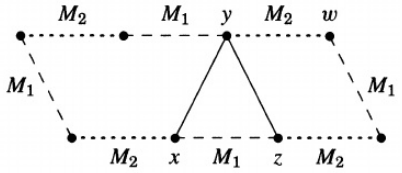


Cas 2: $G \setminus U$ contient 2 sommets qui sont à distance 2 (sinon on serait dans le cas 1): $x - y - z$ sommets qui ne sont pas dans U. De plus, y n'est pas dans U, donc de degré $< |G| - 1$, et il existe un sommet $w \in G \setminus U$ qui n'est pas connecté à y. De par le choix de G, si l'on rajoute une arête à G, on crée un 1-facteur; soit M_1 le 1-facteur dans $G + xz$, et M_2 le 1-facteur dans $G + yw$. Montrons que $M_1 \triangle M_2 \cup \{xy, yz\}$ contient un 1-facteur qui évite xz et yw, c'est donc un 1-facteur dans G.

Soit $F = M_1 \triangle M_2$. On a $xz \in M_1 \setminus M_2$ et $yw \in M_2 \setminus M_1$, donc xz et yw sont dans F. De plus, chaque sommet est de degré 1 dans M_1 et M_2, donc les sommets de F sont de degré 0 ou 2 et il n'y a que des cycles (de longueur paire, alternant entre M_1 et M_2), et des sommets isolés. Soit C le cycle qui contient xz.

• Si C ne contient pas également yw, alors on obtient un 1-facteur avec les arêtes de M_2 dans C et toutes les arêtes de M_1 hors de C.

• Sinon, C contient xz et yw, et l'idée est d'utiliser yx ou yz pour les éviter et avoir un 1-facteur dans G. Dans la portion $y \to w \rightsquigarrow \{x, z\}$, on utilise les arêtes de M_1 (ce qui n'inclut pas yw), et on rajoute zy si on arrive en z comme sur la figure, ou xy sinon (cas symétrique en inversant x et z). Dans le reste du cycle, on utilise les arêtes de M_2, ce qui n'inclut pas xz. On combine ces arêtes avec celles de M_1 ou M_2 en dehors de C, ce qui nous donne un 1-facteur dans G et conclut la preuve.



Remarques. • Vérifications faciles d'existence ou de non-existence: existence = exhiber un 1-facteur; non-existence = exhiber un ensemble qui, si on le supprime, laisse trop de composantes impaires.

• En comptant les sommets modulo 2, $|S| + o(G \setminus S)$ est de même parité que $|G|$. Ainsi, si $|G|$ est pair et G n'a pas de 1-facteur, alors il existe un S tel que $o(G \setminus S)$ dépasse $|S|$ d'au moins 2. On note par la suite $d(S) = o(G \setminus S) - |S|$.

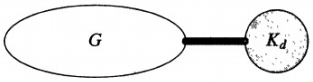
Formule de Berge-Tutte [1958]. Le plus grand nombre de sommets saturés par un couplage dans G est $\min_{S \subseteq V(G)}\{|G| - d(S)\}$.

Preuve: Soit $S \subseteq V(G)$. Au plus $|S|$ arêtes peuvent coupler un sommet de S à

un sommet dans une composante impaire de $G \setminus S$, donc chaque couplage a au moins $o(G \setminus S) - |S| = d(S)$ sommets non saturés. On veut montrer que cette borne est atteinte.

On définit le *join* entre deux graphes, $G \vee H$, par l'union disjointe des deux graphes augmentée de toutes les arêtes allant d'un sommet de G à un sommet de H, par exemple, avec un chemin de longueur 4 et une clique de taille 3:

Soit $d = \max\{d(S) : S \subseteq V(G)\}$. Comme $d(\emptyset) = 0$, on a $d \geq 0$. Soit $G' = G \vee K_d$ (join entre G et la clique de taille d).



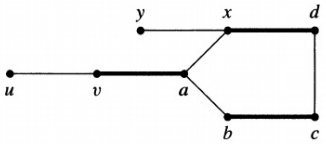
Comme $d(S)$ a la même parité que $|G|$ pour tout S, G' a un nombre pair de sommets. Regardons si G' satisfait la condition de Tutte. On a $o(G' \setminus S') \leq |S'|$ pour $S' = \emptyset$ car $o(G') = 0$. Si S' est non vide mais ne contient pas tout K_d, alors $G' \setminus S'$ a une seule composante, et $1 \leq |S'|$. Finalement, si $K_d \subseteq S'$, soit $S = S' \setminus V(K_d)$. On a alors $G' \setminus S' = G \setminus S$, et donc $o(G' \setminus S') = o(G \setminus S) \leq |S| + d = |S'|$ (par définition de d).

G' satisfait donc la condition de Tutte, et on obtient un couplage de taille $|G| - d$ en supprimant les d sommets que l'on a rajoutés dans G', qui sont couplés à au plus d sommets de G.

Ainsi, on peut prouver facilement qu'un couplage maximal est maximum (de taille max), en trouvant un ensemble S dont la suppression laisse le bon nombre de composantes impaires.

Algorithme d'Edmond (Blossom algorithm). Basé sur le théorème de Berge: M est un couplage maximum dans G ssi G n'a pas de chemin M-améliorant. On a vu comment rechercher de tels chemins dans un graphe biparti (algorithme par augmentation de chemin). La clé est d'arriver à chercher un chemin M-améliorant en un temps raisonnable, car on n'aura que $n/2$ recherches au maximum. Dans un graphe biparti, on explore à partir de chaque sommet une seule fois: en partant de u (sommet M-insaturé), on ne peut atteindre un x dans la même partie que u que via l'unique arête qui couple x, et on construit ainsi un arbre d'exploration.

Cette propriété ne tient plus dans des graphes généraux s'il y a des cycles de taille impaire: des chemins peuvent atteindre x depuis u à la fois via une arête de M et une arête $\notin M$. Exemple ci-dessous: si x est atteint via ax, on ne va pas forcément trouver le chemin M-améliorant $uvabcdxy$.



L'idée proposée par Edmond est basée sur le fait que si x peut être atteint par une arête saturée et une insaturée, alors x est dans un cycle de taille impaire. Les chemins ne peuvent diverger que lorsqu'on explore les arêtes insaturées (une seule arête saturée par sommet!). A partir de la divergence, le chemin qui arrive en x via M a une longueur

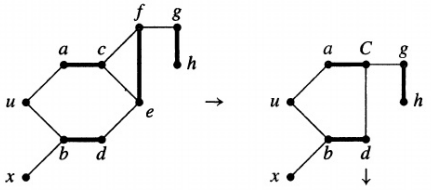
paire, et l'autre a une longueur impaire, d'où le cycle de longueur impaire.

Une **fleur** est l'union de deux chemins M-alternant qui atteignent x (tout le graphe sauf y). La **tige**, de longueur paire, est la partie commune aux deux chemins (uva), et le **blossom**, c'est la fleur à proprement dit, à savoir le cycle de longueur impaire (la fleur moins la tige, $abcdx$).

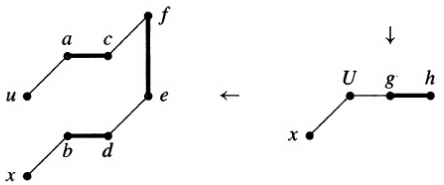
Dans un blossom, on peut tourner d'un côté ou de l'autre pour atteindre tout sommet soit via une arête saturée, soit via une arête insaturée. On peut donc poursuivre la recherche à partir de n'importe quelle arête non saturée qui part du blossom et va vers un sommet pas encore atteint. Dans l'exemple, il y a l'arête xy qui nous donne un chemin M-améliorant. De plus, on note qu'il n'y a que des arêtes non saturées qui partent d'un blossom, excepté la tige (car tous les sommets sont saturés). On contracte ainsi le blossom en un seul super-sommet situé au niveau de la tige, et on continue l'exploration. Si un chemin M-alternant est trouvé, on dé-contracte les super-sommets pour trouver le chemin (en tournant du bon côté dans le cycle).

On garde la terminologie de l'algorithme par augmentation de chemin: T (resp. S) est l'ensemble des sommets atteints via des arêtes non saturées (resp. saturées). Les blossoms contractés se trouvent dans S.

Exemple. Dans le graphe ci-dessous, on effectue la recherche à partir de u. On atteint a et b, que l'on prolonge en c et d: $S = \{u, c, d\}$. En explorant depuis c, on trouve e et f, et comme $ef \in M$, on a un blossom cef que l'on contracte en C: $S = \{u, C, d\}$ (graphe de droite).



On poursuit l'exploration en partant de C: on atteint g et d par les arêtes non saturées. g se prolonge en h (et on met h dans S), et comme d est déjà dans S, on a trouvé un autre blossom $ubdCa$, contracté en U (en bas à droite); on a alors $S = \{U, h\}$. L'exploration depuis h ne donne rien, par contre depuis U on trouve x, qui n'est pas saturé. Il reste à décompresser les blossoms, ce qui donne le chemin M-améliorant sur la gauche: on remplace U par le cycle correspondant, et x était atteint via bx, par le chemin $uaCdb$ (pour arriver en b par une arête de M). On décompresse également C, et d était atteint via ed, par le chemin cfe, d'où finalement le chemin $uacfedbx$.



Sketch de l'algorithme. En entrée, un graphe G, un couplage M, et un sommet non saturé u.
L'idée consiste à explorer les chemins M-alternants depuis u, en se rappelant du *père* de chaque sommet, et en contractant les blossoms. On maintient les ensembles S et T. Si l'on atteint un sommet non saturé, on a trouvé un chemin M-améliorant.

Initialisation: $S = \{u\}$ et $T = \emptyset$.
Itération: Si tous les sommets de S sont marqués, il n'y a pas de chemin M-améliorant depuis u.
Sinon, on choisit $v \in S$ qui n'est pas marqué. Pour explorer depuis v, on considère chaque $y \in N(v)$ tel que $y \notin T$.
• Si y est M-insaturé, on a trouvé un chemin M-améliorant de u vers y et on retourne le chemin (en décontractant les blossoms si besoin).
• Si $y \in S$, on a trouvé un blossom: suspendre l'exploration depuis v, contracter le blossom en remplaçant ses sommets dans S et T par un unique sommet dans S, et continuer la recherche depuis ce nouveau super-sommet.
• Sinon, y est couplé à un w (i.e., $yw \in M$). Dans ce cas, on rajoute y à T (atteint depuis v) et on rajoute w à S (atteint depuis y).
Après avoir exploré toutes les arêtes partant de v, on marque v et on itère.

On ne peut pas explorer simultanément depuis tous les sommets non saturés comme dans la version bipartie, car l'appartenance aux blossoms dépend du choix du sommet de départ. Par contre, si aucun chemin M-améliorant n'est trouvé depuis u, on peut supprimer u et continuer la recherche.

Le premier algorithme proposé par Edmond était en $O(n^4)$, et a établi l'appartenance du problème à la classe P. Avec des structures de données appropriées pour les blossoms, on peut réduire la complexité en $O(n^3)$, et le meilleur algorithme connu est en $O(n^{1/2}m)$.

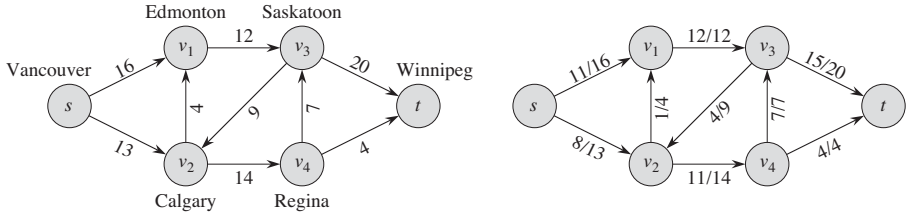
2.6 Flots

De nombreux problèmes doivent modéliser des flots dans des réseaux, en particulier pour maximiser le flot. Par exemple, un flot de liquide dans des tuyaux, des matériaux à transporter, des réseaux de communication. Les algorithmes de flots ont également des applications dans des problèmes qui ne ressemblent pas aux flots, comme l'ordonnancement, ou même les couplages!

2.6.1 Réseaux de transport et le problème de flot maximum

Réseau de transport: graphe orienté $G = (V, E)$ pondéré: chaque arête (u, v) a une capacité $c(u, v) \geq 0$. On a les propriétés suivantes:
• si $(u, v) \notin E$, alors $c(u, v) = 0$;
• si $(u, v) \in E$, alors $(v, u) \notin E$;
• le sommet s est la source du réseau;
• le sommet t est le puits du réseau (target);
• chaque sommet se trouve sur un chemin $s \rightsquigarrow t$.

Exemple à gauche ci-dessous: réseau routier de transport depuis l'usine vers l'entrepôt.



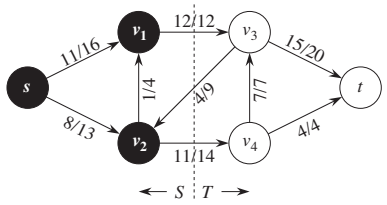
Un flot est alors défini comme une fonction $f : V \times V \to \mathbb{R}$ (on associe un réel aux arêtes), qui satisfait:
• La contrainte de capacité: pour tout $u, v \in V$, $0 \leq f(u, v) \leq c(u, v)$, i.e., la capacité ne peut pas être dépassée. On a donc $f(u, v) = 0$ si $(u, v) \notin E$.
• La contrainte de conservation du flot: pour tout $u \in V \setminus \{s, t\}$, $\sum_{v \in V} f(v, u) = \sum_{v \in V} f(u, v)$, i.e., le flot entrant est égal au flot sortant, sauf pour la source et le puits. Exemple ci-dessus à droite.

La valeur du flot, que l'on note traditionnellement $|f|$, est le flot sortant de la source moins le flot entrant dans la source (i.e., le flot qui sort effectivement de la source): $|f| = \sum_{v \in V} f(s, v) - \sum_{v \in V} f(v, s)$.

Problème du flot maximum. Etant donné G, s, t, c, trouver un flot f de valeur maximum.

2.6.2 Coupes et flots

On a déjà croisé pas mal de fois les coupes: une coupe partitionne V en deux ensembles S et $T = V \setminus S$. Une coupe d'un réseau de transport est telle que $s \in S$ et $t \in T$.



Le **flot net** à travers la coupe est $f(S,T) = \sum_{u\in S}\sum_{v\in T} f(u,v) - \sum_{u\in S}\sum_{v\in T} f(v,u)$.
La **capacité** de la coupe est $c(S,T) = \sum_{u\in S}\sum_{v\in T} c(u,v)$.
Une **coupe minimum** est une coupe dont la capacité est minimale (rapportée à toutes les coupes du réseau).
Lemme: Pour toute coupe (S,T), $f(S,T) = |f|$, i.e., le flot net à travers toute coupe est égal à la valeur du flot. L'intuition est que quel que soit l'endroit où l'on coupe les tuyaux d'un réseau, on verra le même flot sortir. Sinon, la loi de conservation ne serait pas respectée quelque part.
Corollaire: La valeur de tout flot est majorée par la capacité de toute coupe. Encore intuitif: si c'était faux, on pourrait faire passer plus de flot dans les tuyaux que ce qu'ils peuvent contenir.

Preuve du lemme: loi de conservation pour tout noeud u autre que s et t: $\sum_{v\in V} f(u,v) - \sum_{v\in V} f(v,u) = 0$. On rajoute ces termes pour $u \in S \setminus \{s\}$ à la définition de $|f|$, ce qui donne, en regroupant les termes $f(u,v)$ et $f(v,u)$, $|f| = \sum_{v\in V}\sum_{u\in S} f(u,v) - \sum_{v\in V}\sum_{u\in S} f(v,u)$. On sépare V en $S \cup T$, et du coup les parties avec $u, v \in S$ s'annulent, il reste $|f| = \sum_{v\in T}\sum_{u\in S} f(u,v) - \sum_{v\in T}\sum_{u\in S} f(v,u) = f(S,T)$, ce qui conclut la preuve. On voit au passage que $|f| \leq \sum_{v\in T}\sum_{u\in S} f(u,v) \leq \sum_{v\in T}\sum_{u\in S} c(u,v) = c(S,T)$, d'où le corollaire.

2.6.3 Méthode de Ford-Fulkerson

On parle de méthode et non d'algorithme, car il y a plein de façons de la réaliser.
FF-Méthode(G,s,t):
1. Initialiser le flot f à 0;
2. Tant qu'il existe un chemin améliorant p dans le réseau résiduel G_f,
3. Améliorer le flot f le long du chemin p;
4. Retourner f;

L'idée consiste simplement à chercher des chemins dont la capacité totale n'est pas utilisée, et à augmenter le flot le long de ce chemin. On continue tant qu'on trouve de tels chemins.

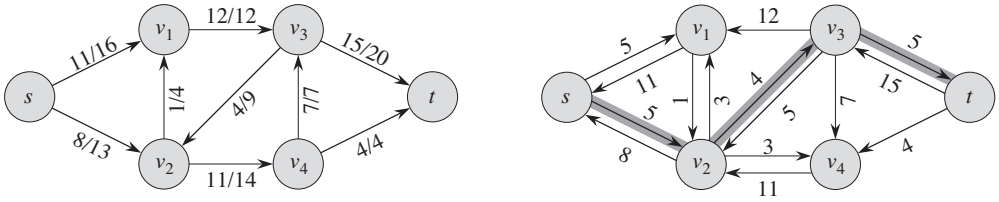
Paradoxe: il arrive que l'on améliore le flot total en diminuant le flot le long de certaines arêtes (mauvaise direction, ou bien va dans une partie du réseau qui ne peut pas bien le traiter).

Réseau résiduel. La capacité résiduelle d'une arête détermine combien de flot additionnel on peut faire passer de u vers v.
• Si $(u,v) \in E$, on peut encore utiliser $c(u,v) - f(u,v)$.
• Si $(v,u) \in E$, on peut annuler jusqu'à $f(v,u)$ qui va de v vers u, ce qui est équivalent à faire passer le flot de u vers v.
• Sinon, on ne peut rien faire passer de u vers v.

On définit ainsi la capacité résiduelle de u vers v:

$$c_f(u,v) = \begin{cases} c(u,v) - f(u,v) \text{ si } (u,v) \in E \\ f(v,u) \text{ si } (v,u) \in E \\ 0 \text{ sinon} \end{cases}$$

Le réseau résiduel est $G_f = (V, E_f)$, où $E_f = \{(u,v) \in V \times V : c_f(u,v) > 0\}$. Ainsi, chaque arête correspond à $(u,v) \in E$ ou $(v,u) \in E$, et $|E_f| \leq 2|E|$. Il peut contenir des arêtes anti-parallèles, mais autrement c'est un réseau de transport avec des capacités! Exemple avec le réseau résiduel sur la droite:



Entre s et v_1, on peut faire passer 5 de plus, ou bien retirer 11, d'où les deux arêtes correspondantes. Lorsque $f(u,v) = c(u,v)$, on ne peut que retirer.

Augmentation de flots et chemins améliorants. On commence par définir l'augmentation d'un flot f dans G par un flot f' dans G_f, noté $f \uparrow f'$ (fonction de $V \times V$ dans $\mathbb{R}$):

$$(f \uparrow f')(u,v) = \begin{cases} f(u,v) + f'(u,v) - f'(v,u) \text{ si } (u,v) \in E, \\ 0 \text{ sinon} \end{cases}$$

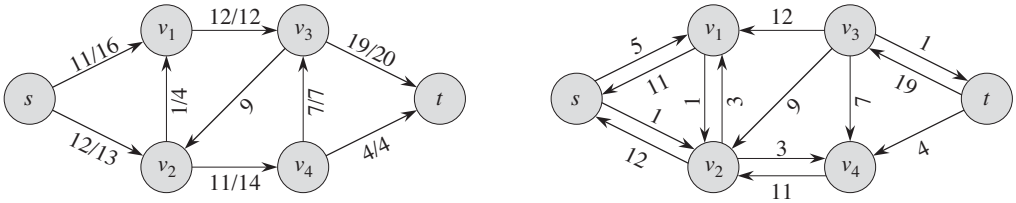
On augmente le flot de $f'(u,v)$ sur l'arête (u,v), mais on le diminue de $f'(v,u)$ car pousser le flot en direction opposée à la direction de l'arête revient à diminuer le flot.

Lemme: Etant donné un réseau G, un flot f dans G, le réseau résiduel G_f, et un flot f' dans G_f, alors $f \uparrow f'$ est un flot dans G de valeur $|f \uparrow f'| = |f| + |f'|$.

La preuve est un peu technique mais assez intuitive, on se sert des définitions de flot, des contraintes de capacité et de la conservation des flots. On montre alors que les contraintes sont satisfaites, puis on calcule la valeur du flot $f \uparrow f'$.

Un **chemin améliorant** est un chemin simple p de s à t dans G_f. Comme toutes les capacités dans G_f sont strictement positives, on peut faire passer du flot le long de ce chemin, en suivant la loi du maillon le plus faible: $c_f(p) = \min\{c_f(u,v) : (u,v) \in p\}$.

Dans le réseau résiduel ci-dessus, le chemin en gras est donc un chemin améliorant, et sa capacité est 4. On rajoute ce flot sur le chemin (et donc on soustrait 4 sur l'arc inverse $v_3 \to v_2$), obtenant le flot ci-dessous à gauche. Le nouveau réseau résiduel est à droite.



S'il n'y a plus de chemin améliorant, on affirme que le flot obtenu dans G est maximum!

Théorème max-flow min-cut. On a l'équivalence entre:
1. f est un flot maximum;
2. G_f n'a pas de chemin améliorant;
3. $|f| = c(S,T)$ pour une coupe (S,T).

Ainsi, si on ne peut pas trouver de chemin améliorant ou si on a un flot égal à la capacité d'une coupe, alors on a trouvé un flot maximum. On peut aussi prédire la valeur d'un flot maximum: il sera égal à la capacité de la coupe minimum (en terme de capacité): **max flow** est équivalent à **min cut**! Encore un problème min/max!

Preuve: (1) $\Rightarrow$ (2): Si f est un flot maximum et il y a un chemin améliorant, alors on peut augmenter f par f_p, où le flot f_p vaut $c_f(p)$ sur les arcs de p et 0 ailleurs. On obtient le flot $|f \uparrow f_p|$, de valeur strictement plus grande que $|f|$, d'où la contradiction.
(2) $\Rightarrow$ (3): Supposons que G_f n'a pas de chemin améliorant: pas de chemin de s vers t. On définit S par l'ensemble des sommets $v \in V$ tels qu'il existe un chemin de s vers v dans G_f, ce qui nous définit une coupe $(S, V \setminus S)$. Soit $u \in S$ et $v \in T$. Si $(u,v) \in E$, alors $f(u,v) = c(u,v)$ car sinon $(u,v) \in E_f$ et v serait dans S. Si $(v,u) \in E$, alors $f(v,u) = 0$ car sinon $c_f(u,v) = f(v,u)$ serait positif et encore on aurait $(u,v) \in E_f$. Sinon, $f(u,v) = f(v,u) = 0$. Ainsi, en sommant sur tous ces couples (u,v), on obtient

$$f(S,T) = \sum_{u \in S} \sum_{v \in T} f(u,v) - \sum_{u \in S} \sum_{v \in T} f(v,u) = \sum_{u \in S} \sum_{v \in T} c(u,v) - 0 = c(S,T),$$

et donc $|f| = f(S,T) = c(S,T)$ pour cette coupe.
(3) $\Rightarrow$ (1): On a $|f| \leq c(S,T)$ pour toute coupe (S,T). Comme $|f| = c(S,T)$ pour une coupe, le flot est maximum.

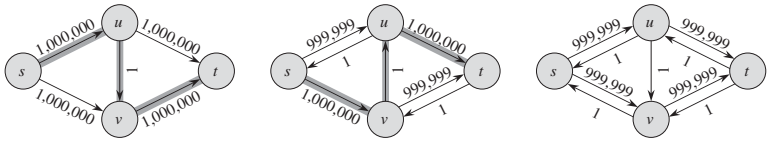
Algorithme de Ford-Fulkerson. L'idée consiste à augmenter le flot sur un chemin améliorant tant qu'il y en a.
Ford-Fulkerson(G,s,t):
1. Pour chaque $(u,v) \in E$, $f(u,v) := 0$;
2. Tant qu'il existe un chemin p dans G_f,
3. $\quad c_f(p) := \min\{c_f(u,v) : (u,v) \in p\}$;
4. $\quad$ Pour chaque arête $(u,v) \in p$,
5. $\quad\quad$ si $(u,v) \in E$, alors $f(u,v) := f(u,v) + c_f(p)$;
6. $\quad\quad$ sinon $f(v,u) := f(v,u) - c_f(p)$;

En ligne 5, on augmente le flot, ou on le réduit en ligne 6 si le chemin prend un arc inverse.

Analyse: le temps d'exécution dépend de la méthode utilisée pour chercher les chemins, et aussi des capacités du réseau. C'est mieux d'utiliser des poids entiers si possible.
Initialisation ligne 1 en $O(E)$.
Recherche de chemin ligne 2 en $O(V+E)$ avec BFS ou DFS par exemple, soit $O(E)$ car le graphe est connexe.
Intérieur de la boucle "Tant que": complexité inférieure à la recherche de chemin. La complexité totale est donc $O(E)\times$ nombre de passages dans la boucle.

Dans le pire des cas, chaque chemin améliorant augmente $|f|$ de 1, et donc si le flot maximum est f^*, on peut avoir besoin de f^* itérations, ce qui donne un coût de $O(Ef^*)$.

Exemple classique où ce pire cas est atteint:



... en alternant entre les 2 chemins de capacité 1, il faudra $2,000,000$ itérations pour converger!

Algorithme d'Edmonds-Karp. Vient à la rescousse en contrôlant l'ordre dans lequel les chemins sont explorés. Dans l'exemple précédent, si on prend les plus courts chemins (en terme de nombres d'arêtes), on n'a pas de problèmes car on trouvera $s-u-t$ et $s-v-t$.

Edmonds-Karp = Ford-Fulkerson avec une recherche BFS sur G_f.

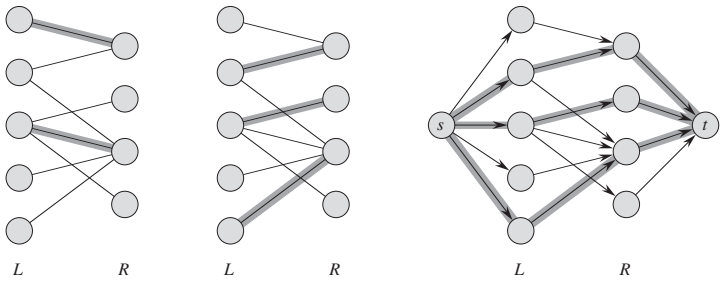
Théorème: L'algorithme d'Edmonds-Karp réalise au maximum $O(VE)$ augmentations de flots, soit une complexité en $O(VE^2)$, avec des capacités dans $\mathbb{R}^+$.

Preuve: borner les distances aux sommets dans G_f. $\delta_f(u,v)$: longueur d'un plus court chemin de u à v dans G_f (où chaque arc a un poids unitaire).
On montre d'abord que $\delta_f(s,v)$ augmente de façon monotone avec chaque augmentation de flot pour tout $v \in V\setminus\{s,t\}$: on obtient une contradiction si une distance a été diminuée (propriétés de plus court chemin, inégalité triangulaire).
Pour borner le nombre d'itérations, on définit les arcs critiques qui sont tels que $c_f(p) = c_f(u,v)$. Après avoir fait une augmentation de flot, tous les arcs critiques disparaissent du réseau résiduel. De plus, au moins un arc doit être critique. On montre que chacun des $|E|$ arcs ne peut pas devenir critique plus de $|V|/2$ fois.
Considérons l'arc $(u,v) \in E$. Lorsqu'il est critique, il est sur un plus court chemin et donc $\delta_f(s,v) = \delta_f(s,u)+1$. Après augmentation, (u,v) disparaît du réseau résiduel, et ne peut ré-apparaître que si le flot est diminué, i.e., (v,u) est sur un chemin améliorant. Alors, on aurait un flot f' et $\delta_{f'}(s,u) = \delta_{f'}(s,v)+1$. Augmentation monotone des flots: $\delta_f(s,v) \leq \delta_{f'}(s,v)$, d'où $\delta_{f'}(s,u) \geq \delta_f(s,u)+2$: augmentation de 2 entre deux sélections critiques, et la distance est bornée par $|V|-2$ (on a $u \neq t$), donc après la première fois où (u,v) devient critique, il ne peut pas devenir critique plus de $(|V|-2)/2 = |V|/2-1$ fois, soit un total d'au plus $|V|/2$ fois. Chaque chemin améliorant ayant au moins un arc critique, on a au plus $O(VE)$ améliorations, d'où la complexité en $O(VE^2)$.

2.6.4 Couplage dans des graphes bipartis

Problème déjà rencontré: trouver un couplage maximum dans un graphe biparti.

Etant donné $G=(V,E)$, avec $V = L \cup R$, on définit le réseau de transport $G' = (V', E')$ avec $V' = V \cup \{s,t\}$, et $E' = E$ augmenté des arcs de s vers tout $u \in L$, et de tout $v \in R$ vers t. On oriente également les arêtes de E de L vers R. Finalement, $c(u,v) = 1$ pour tout $(u,v) \in E'$.

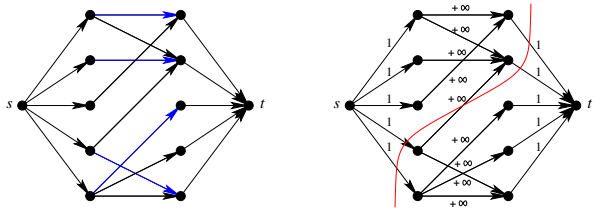


Algorithme: Ford-Fulkerson (pas besoin d'utiliser Edmonds-Karp car toutes les capacités sont unitaires). On a la correction car un flot maximum doit utiliser le nombre maximum d'arcs (de capacité 1) à travers la coupe (L, R).

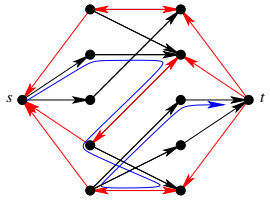
Complexité: Ford-Fulkerson en $O(Ef^*)$. Ici on est sur E' mais $E' = O(E)$. Et on peut borner f^* par $\min(|L|, |R|) = O(V)$ car les arcs ont une capacité unitaire, et on ne peut pas avoir 2 qui sort d'un sommet de L ou 2 qui rentre dans un sommet de R. D'où une complexité totale en $O(VE)$.

Note: A chaque itération, Ford-Fulkerson va sélectionner un flot entier, et n'utilisera jamais des fractions de capacité, ce qui permet de garantir qu'on a bien l'équivalence entre flot et couplage.

Corollaire: König-Egervàry. On retrouve $\max |couplage| = \min |couverture|$ via $\max |flot| = \min |coupe|$: la coupe min (S, T) nous identifie une couverture min: aucune arête de $L \cap S$ vers $R \cap T$. Et la capacité de la coupe est égale à la taille de la couverture: $|(L \cap T) \cup (R \cap S)|$. Idem si on a une couverture, on peut construire une coupe de même taille!



Lien avec Berge. Chemin améliorant pour le flot dans le réseau = chemin améliorant pour le couplage dans le graphe biparti.



2.6.5 Théorème de Menger

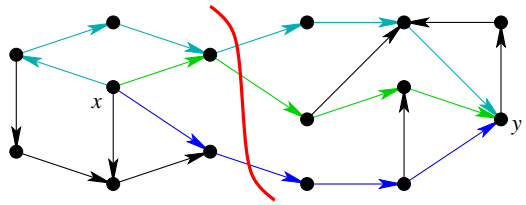
Application des flots à la connectivité dans les graphes.

Version arcs/arêtes: Soit G graphe orienté ou pas, et deux sommets x,y, alors :

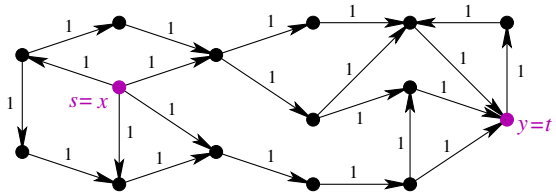
nb max de chemins arcs/arêtes disjoints de x à y
=
nb min d'arcs/arêtes à enlever pour séparer x de y

Noté $\kappa(G, x, y)$, ce nombre et les ensembles associés sont calculables en $O(mn)$.

Exemple: Un ensemble max de chemins arcs-disjoints et un séparateur min.



Modélisation par flot max entre source x et puits y:



2.6.6 Algorithmes push-relabel

Approche différente de la méthode Ford-Fulkerson, qui est à la base des algorithmes les plus rapides connus à ce jour ($O(V^2E)$ au lieu de $O(VE^2)$, que l'on peut encore améliorer en $O(V^3)$). Approche plus locale: on regarde un sommet et ses voisins dans le réseau résiduel.

On maintient un **préflot**, qui n'a pas toutes les propriétés des flots: on a la contrainte de capacité, mais la conservation de flot est un peu relâchée: $\sum_{v\in V} f(v,u) - \sum_{v\in V} f(u,v) \geq 0$ au lieu d'avoir $= 0$. Cela signifie que le flot entrant peut excéder le flot sortant (vrai pour $u \in V \setminus \{s\}$, imaginer un réservoir associé à chaque sommet qui garde l'excès).

On note $e(u) = \sum_{v\in V} f(v,u) - \sum_{v\in V} f(u,v)$ l'**excédent** de flot sur u, et le sommet $u \in V \setminus \{s,t\}$ **déborde** si $e(u) > 0$. Le préflot est un flot si aucun sommet déborde.

Intuition. Imaginer encore des tuyaux dans lesquels on fait couler de l'eau. La méthode Ford-Fulkerson cherchait un nouveau passage de liquide (chemin améliorant). Ici, on place les sommets à des hauteurs différentes pour faire couler le liquide: on peut pousser du liquide vers le bas. La source est à une hauteur $|V|$ et le puits à une hauteur 0, et les autres sommets, initialement à 0, sont remontés avec le temps (on peut alors avoir éventuellement du liquide qui remonte). Initialement, on pousse le liquide depuis la source, soit la capacité de la coupe $(\{s\}, V \setminus \{s\})$.

Lorsqu'un sommet déborde, on le remonte pour pouvoir le vider à son tour s'il n'a pas de voisins plus bas dans lesquels il peut pousser du flot. Une fois qu'on ne peut plus rien pousser, on vide l'excédent en remontant vers la source et en transformant le préflot en flot. On va montrer que le flot obtenu est par ailleurs un flot maximum.

Opérations de base. Les deux opérations de base sont donc le *push* (pousser le flot d'un sommet qui déborde), et le *relabel* (changer la hauteur d'un sommet). On définit une fonction de **hauteur** associée à un préflot f par $h : V \to \mathbb{N}$ par $h(s) = |V|$, $h(t) = 0$, et pour tout arc dans le réseau résiduel $(u, v) \in E_f$, $h(u) \leq h(v)+1$. Ainsi, si $h(u) > h(v)+1$, alors (u, v) n'est pas un arc du réseau résiduel.

Opération **Push**(u, v): s'applique si u déborde, $c_f(u, v) > 0$, et $h(u) = h(v) + 1$. L'opération met à jour le préflot f et les excédents en u et v: on pousse $\Delta_f(u, v) = \min(e(u), c_f(u, v))$ de u vers v.
Push(u, v):
1. $\Delta_f(u, v) = \min(e(u), c_f(u, v))$;
2. Si $(u, v) \in E$ alors $f(u, v) = f(u, v) + \Delta_f(u, v)$;
3. Sinon $f(v, u) = f(v, u) - \Delta_f(u, v)$;
4. $e(u) = e(u) - \Delta_f(u, v)$; $e(v) = e(v) + \Delta_f(u, v)$;

Si l'arc du réseau résiduel est un arc du réseau de transport, on rajoute le flot. Sinon, on diminue le flot sur (v, u) qui est un arc du réseau de transport. f reste un préflot après l'opération push. On suppose que $c_f(u, v)$ peut être calculé en temps constant étant donné c et f.

On ne se sert pas du fait que $h(u) = h(v)+1$, mais on ne pousse le flot que d'une case vers le bas. De par la fonction de hauteur, il n'y a de toute façon pas d'arc résiduel entre u et v si la différence est supérieure à 1. Le push est **saturant** si $\Delta_f(u, v) = c_f(u, v)$, i.e., $c_f(u, v) = 0$ après le push, et non saturant sinon.

Après un push non saturant de u vers v, le sommet u ne déborde plus. En effet, on a $\Delta_f(u, v) = e(u)$ et $e(u)$ devient 0 après le push.

La deuxième opération est **Relabel**(u): elle s'applique si u déborde, et pour tout $v \in V$ tel que $(u, v) \in E_f$, $h(u) \leq h(v)$. Alors on monte u.
Relabel(u):
1. $h(u) = 1 + \min\{h(v) : (u, v) \in E_f\}$;

On effectue un relabel uniquement lorsque E_f contient au moins un arc qui quitte u, donc le minimum est sur un ensemble non vide. Ceci peut être déduit du fait que $e(u) > 0$, car alors il y a forcément un v tel que $f(v, u) > 0$, et donc $c_f(u, v) > 0$ et $(u, v) \in E_f$. L'opération relabel donne la plus grande hauteur autorisée par les contraintes sur les fonctions de hauteur.

Algorithme générique. On initialise le préflot en mettant $h(v) = e(v) = 0$ pour tout $v \in V$, et $f(u, v) = 0$ pour tout $(u, v) \in E$. Ensuite, on monte la source et on pousse le flot depuis la source: $h(s) = |V|$, et pour tout v voisin de s, $f(s, v) = c(s, v)$, $e(v) = c(s, v)$, et $e(s) = e(s) - c(s, v)$.

L'algorithme générique effectue cette initialisation, puis tant qu'on peut appliquer une opération push ou relabel, il en effectue une.

Lemme: si le sommet u déborde, on peut toujours lui appliquer une opération push ou une opération relabel. Preuve: fonction de hauteur donc $h(u) \leq h(v) + 1$ pour tout arc résiduel. Si le push ne s'applique pas, alors on doit avoir $h(u) < h(v) + 1$, et donc $h(u) \leq h(v)$ et le relabel s'applique.

Correction de la méthode push-relabel. On montre ici que si l'algorithme générique termine, alors on obtient un flot maximum. On verra plus tard que l'algorithme termine (analyse de complexité).

Lemme: les hauteurs des sommets ne diminuent jamais, et le relabel augmente la hauteur d'au moins 1 (regarder la fonction relabel, qui est la seule à toucher à la hauteur; $h(u)$ est strictement inférieur à $h(v)+1$ pour tout v...).

Lemme: l'application de l'algorithme générique conserve une fonction de hauteur valide.

Preuve par récurrence: vrai initialement, puis on vérifie que cela reste vrai après un push ou un relabel.

Après un relabel, on regarde les arcs du réseau résiduel arrivant en u, et ceux sortant de u. Arc sortant (u,v): $h(u) \leq h(v)+1$ après le relabel par choix de $h(u)$ (choix du v de hauteur min). Arc entrant (w,u) $h(w) \leq h(u)+1$ reste vrai après avoir augmenté $h(u)$! Après un push, il se peut que cela rajoute (v,u) au réseau résiduel. Dans ce cas, $h(v) = h(u)-1 < h(u)+1$ donc h reste une fonction de hauteur. Si au contraire (u,v) disparaît de E_f, cela enlève une contrainte sur la fonction h.

Lemme: Il n'y a pas de chemin de s vers t dans le réseau résiduel.

Preuve par contradiction: chemin simple $p =< s = v_0, v_1, \ldots, v_k = t >$ de s vers t, et $k < |V|$. Alors, $(v_i, v_{i+1}) \in E_f$ et donc $h(v_i) \leq h(v_{i+1}) + 1$ pour $0 \leq i \leq k-1$. En sommant les inégalités, on obtient $h(s) \leq h(t) + k$, d'où $h(s) < |V|$, ce qui contredit le fait que $h(s) = |V|$.

Théorème: correction de l'algorithme générique push-relabel: si l'algorithme termine, alors le préflot est un flot maximum pour G.

Preuve: on utilise l'invariant suivant: f est toujours un préflot. C'est facile à vérifier pour l'initialisation. Relabel ne touche pas au flot. Et push conserve la propriété de flot car on garde toujours un excédent ≥ 0.

Si l'algorithme termine, il n'y a plus de sommet qui déborde (sinon on pourrait faire push ou relabel), et donc $e(u) = 0$ pour tout $u \in V \setminus \{s,t\}$, ce qui signifie que f est un flot. De plus, il n'y a pas de chemin de s à t dans G_f (le réseau résiduel), et de par le théorème max-flow min-cut, f est un flot maximum.

Analyse de la méthode push-relabel. On compte ici le nombre d'opérations pour montrer qu'il est borné et donc que la méthode termine. On compte le nombre de relabel, de push saturant, et de push non saturant.

Lemme: Pour chaque sommet x qui déborde, il y a un chemin simple de x vers s dans G_f.

Lemme: On a toujours $h(u) \leq 2|V| - 1$ pour tout $u \in V$, ce qui borne le nombre d'opérations relabel en $O(V^2)$.

Lemme: Le nombre de push saturant est en $O(VE)$.

Lemme: Le nombre de push non saturant est en $O(V^2E)$.

On peut donc borner le nombre d'opérations de l'algorithme générique par $O(V^2E)$, et ainsi montrer que l'algorithme termine. Il suffit de trouver des méthodes efficaces pour implémenter ces opérations et décider quelle opération exécuter.

2.6.7 Algorithme relabel-to-front

Utilisation de la méthode push-relabel pour obtenir un algorithme en $O(V^3)$. L'idée consiste à maintenir une liste de sommets dans le réseau. On choisit le premier sommet de la liste, et on le décharge complètement: on fait des push et relabel jusqu'à ce que son excédent soit nul (i.e., le sommet ne déborde plus). Lorsqu'on fait un relabel, on remet le sommet au début de la liste (d'où le nom relabel-to-front).

Arêtes admissibles. Etant donné un réseau de transport $G = (V, E)$ avec source s et destination t, un préflot f dans G, et une fonction de hauteur h, on dit que (u, v) est une **arête admissible** si $c_f(u, v) > 0$ et $h(u) = h(v) + 1$. $E_{f,h}$ est l'ensemble des arêtes admissibles et $G_{f,h} = (V, E_{f,h})$ est le réseau admissible. Il contient les arêtes sur lesquelles on peut pousser du flot.

Lemme: le réseau admissible est acyclique.
Preuve: par contradiction, s'il y a un cycle $< v_0, v_1, \ldots, v_k = v_0 >$, avec $k > 0$, on somme sur la propriété des hauteurs sur chaque arête, obtenant $0 = k$.

Lemme: si u déborde et (u, v) est admissible, alors on peut faire Push(u, v). L'opération ne peut pas créer d'arêtes admissibles, mais (u, v) peut ne plus être admissible.
Preuve: Push s'applique par définition. Nouvelle arête potentiellement créée dans le réseau résiduel: uniquement (v, u), mais comme $h(v) = h(u) - 1$, l'arête ne peut pas devenir admissible. Push saturant: $c_f(u, v) = 0$ après le push et l'arête (u, v) n'est plus admissible.

Lemme: si u déborde et s'il n'y a pas d'arête admissible quittant u, alors on peut faire Relabel(u), et après cette opération, il y a au moins une arête admissible quittant u, et aucune arrivant en u.
Preuve: Vu dans l'algo générique, si u déborde, on peut faire push ou relabel, et vu qu'il n'y a pas d'arête admissible, on ne peut pas faire push donc on peut faire relabel. Après le relabel, $h(u) = 1 + \min\{h(v) : (u, v) \in E_f\}$, et donc si v réalise le minimum, (u, v) devient admissible. Finalement, supposons qu'il existe v tel que (v, u) soit admissible. Alors $h(v) = h(u) + 1$ après le relabel, et donc $h(v) > h(u) + 1$ avant le relabel. Cela signifie que (v, u) n'est pas dans le réseau résiduel (définition de la hauteur), qui ne change pas suite au relabel. (v, u) n'est donc pas admissible.

Décharge et listes de voisins. Pour décharger un sommet, on regarde les sommets voisins et on essaie d'y pousser du flot. On conserve une liste (simplement chaînée) des voisins $N(u)$ de u, i.e., les sommets v tels que $(u, v) \in E$ ou $(v, u) \in E$. $N(u).tete$ est la tête de liste, et $v.voisinsuivant$ est le sommet suivant dans la liste (NIL pour le dernier sommet de la liste). Initialement, $u.courant = N(u).tete$, et c'est le voisin que l'on considère.

Opération de décharge: push vers les voisins, et relabel si nécessaire (pour avoir une arête quittant u qui devienne admissible).
Decharge(u):
1. Tant que $e(u) > 0$
2. $\quad v = u.courant$
3. $\quad$ Si $v = NIL$ alors Relabel(u), $u.courant = N(u).tete$;
4. $\quad$ Sinon, si $c_f(u, v) > 0$ et $h(u) = h(v) + 1$, alors Push(u, v);

5. Sinon, $u.courant = v.voisinsuivant$;

Ligne 3, on a fini de parcourir la liste des voisins et l'excédent est toujours > 0, donc on relabel et on ré-initialise $u.courant$ en tête de liste.
Montrer que le relabel est possible = montrer que toutes les arêtes quittant u ne sont pas admissibles (détails dans le Cormen).

Ligne 4, par définition on peut faire un Push(u, v).

Ligne 5, (u, v) n'est pas admissible, donc on avance au voisin suivant $v.voisinsuivant$ dans la liste $N(u)$.

Terminaison lorsque $e(u) = 0$, et seul un Push peut produire ce résultat. La dernière opération de Decharge est donc forcément un push.

L'algorithme relabel-to-front. Liste L avec tous les sommets de $V \setminus \{s, t\}$, triés dans un ordre topologique selon le réseau admissible. $u.next$ pointe vers le sommet suivant u dans L. On a aussi toutes les listes de voisins $N(u)$.
Relabel-to-front(G, s, t):
1. Initialise-Preflot(G, s);
2. Initialise L (sommets dans un ordre quelconque);
3. Pour chaque $u \in V \setminus \{s, t\}$, $u.courant = N(u).tete$;
4. $u = L.tete$;
5. Tant que $u \neq NIL$
6. $\quad oldh = h(u)$;
7. $\quad$ Decharge(u);
8. $\quad$ Si $h(u) > oldh$, positionner u au début de L;
9. $\quad u = u.next$;

Boucle de la ligne 5: parcourt de la liste L et décharge les sommets. Si u a subit un relabel (sa hauteur a augmenté), il est remis au début de la liste.

Pour montrer que Relabel-to-front calcule un flot maximum, il suffit de montrer que c'est un algorithme générique push-relabel. Etude de Decharge: on fait des push et relabel quand c'est possible. Il reste à montrer que lorsque l'algorithme termine, il n'y a plus d'opérations push ou relabel possibles.

Invariant: A chaque test en ligne 5, L est triée dans un ordre topologique selon le réseau admissible, et aucun sommet avant u dans la liste ne déborde.

Vrai initialement ($E_{f,h} = \emptyset$ car hauteurs 0 ou $|V| \geq 2$, et u est la tête de liste).

Chaque itération maintient le tri topologique: seul le relabel peut créer des arcs admissibles, et uniquement des arcs sortant de u dans ce cas. Du coup, mettre u en début de liste maintient le tri topologique. Pour les sommets débordant, regardons u' qui sera choisi à l'itération suivant u. Si u a eu un relabel, aucun autre sommet avant u', donc pas de sommets qui débordent avant u'. Sinon, aucun sommet avant u n'a reçu du flot: le tri topologique impose que la décharge se fait vers des sommets après u dans la liste.

A la fin, par invariant, aucun sommet ne déborde, donc aucune opération ne peut être appliquée.

Analyse. L'algorithme est en $O(V^3)$. Comme pour l'algorithme générique, il y a au plus $O(V^2)$ opérations relabel, et à chaque phase il y a au plus $|V|$ appels à Decharge.

Sans compter le travail fait dans Decharge, la complexité est au plus $O(V^3)$.

Pour terminer l'analyse, il faut borner le travail fait pendant toutes les opérations Decharge au cours de l'algorithme (analyse amortie). On compte le travail des relabel, des push non saturant, et les mises à jour des pointeurs. Essayez de compter les opérations pour vous en convaincre (ou regardez le Cormen!).

3 Algorithmique des mots

Dans cette dernière partie du cours, on s'intéresse à la recherche d'un motif x de longueur m dans un texte t de longueur n. Il n'y a pas de notes de cours pour ce chapitre, mais il est fortement inspiré du livre "Elements d'algorithmique" de Beauquier, Berstel et Chrétienne, disponible librement à l'adresse `http://www-igm.univ-mlv.fr/~berstel/Elements/Elements.html`. En complément, vous pouvez également vous référer au Cormen. Finalement, la page `http://www-igm.univ-mlv.fr/~lecroq/string/index.html` répertorie de nombreux algorithmes de recherche de motifs, avec le code et des animations Java.