Algorithmique des graphes

Équipe pédagogique :

Rumen Andonov <randonov@irisa.fr> : CM et TD (G1, G2 en Info,

Pablo Alvarez <pablo-arturo.alvarez@univ-rennes1.fr> : TD (G3 en Info)

Université de Rennes 1 et INRIA Rennes Bretagne-Atlantique

Contenu du cours

1. Généralités sur les graphes : exemples de graphes comme modèles de situations concrètes, et questions associées pour découverte des notions (chemins, PCC, fermeture transitive, descendance, clique, CFC, arbres, etc.).
2. Représentation des graphes, structures de données associées : plusieurs représentations (matrice d'adjacence, liste des prédécesseurs, liste des successeurs), équivalence et passage de l'une à l'autre.
3. Notions de complexité des algorithmes (ordre de grandeur des fonctions).
4. Parcours en profondeur « *Depth-first search (DFS)* » et en largeur « *Breadth-first search (BFS)* ». *Directed Acyclic Graph (DAG)*.
5. Composantes fortement connexes.
6. Les problèmes du plus court chemin (PCC) :
   - algorithme de Dijkstra. *Binary heap* (tas binaire) ;
   - algorithme de Bellman-Ford. Découverte des circuits négatifs ;
   - PCC dans un DAG (ordre topologique).
7. Algorithmes gourmands pour l'ACM (*Minimum spanning tree*) : algs de Prim et de Kruskal.
8. Le problème du flot maximum dans les réseaux de transport.
9. Chemins, flots et programmes linéaires dans les graphes.

Bibliographie

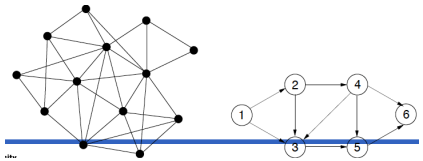
Ce cours est fondé essentiellement sur les références suivantes.

- *Algorithms*, S. Dasgupta, C. H. Papadimitriou, et U. V. Vazirani, McGraw-Hill 2006 (*the undergraduate Algorithms course at Berkeley and U.C. San Diego*)
- *Graphes et algorithmes*, Michel Gondran et Michel Minoux, Eyrolles, 1995

Généralités, notions de base, exemples d'applications

Définitions formelles et notations : graphe, graphe orienté et non orienté

Un *graphe* est un doublet $G = (V, E)$, où $V = \{v_1, v_2, \ldots, v_n\}$ est l'ensemble des sommets/noeuds et E est un ensemble de couples $(u, v) \in V \times V$. Si tous les couples $(u, v) \in E$ sont symétriques, le graphe est dit *non orienté*. Sinon, le graphe est *orienté*. Un couple symétrique / non ordonné est dit une *arête*. Un couple non symétrique / ordonné est dit un *arc*.



Deux graphes : non orienté et orienté.

5/152

Exemples de graphes

Obtention d'un diplôme Master d'informatique (Exam. décembre 2008)

Pour obtenir un Master d'informatique, il est nécessaire d'avoir passé un certain nombre de modules. Chaque module demande un certain nombre de prérequis. Un master simplifié pourrait se composer (prérequis entre parenthèses) de :

- Système (Programmation) ;
- AGR1 (Programmation, Math Discrètes) ;
- Sécurité réseau (Système, Initiation réseau) ;
- Initiation réseau (Programmation, AGR1) ;
- Systèmes répartis (AGR1, Système, Initiation réseau).

1. Modéliser les prérequis à l'aide d'un graphe.
2. Quel est le nombre minimum de semestres pour obtenir ce master ? On suppose bien sur que vous passez tous les examens avec succès, que chaque module dure un semestre et est enseigné chaque semestre. Le nombre de modules suivis par un étudiant pendant un semestre n'est pas limité et il n'y a pas de problème d'emploi du temps. Indiquer l'algorithme utilisé et justifiez votre choix.
3. Un étudiant décide d'étudier un module dès qu'il a obtenu les modules prérequis. Quel est le nombre maximum de modules qu'il devra suivre simultanément en appliquant cette stratégie ? Quel algorithme permet de le calculer ? Justifiez votre choix.

6/152

Exemple de graphes : suite

Construction d'un pavillon

La construction d'un pavillon demande la réalisation d'un certain nombre de tâches. La liste des tâches à effectuer, leur durée et les contraintes d'antériorités à respecter sont données dans le table ci-dessus. Le travail commençant à la date 0, on cherche un planning des opérations qui permet de minimiser la durée totale.

Code tâche	libellé	durée (semaines)	antériorité
A	Travaux de maçonnerie	7	-
B	Charpente de la toiture	3	A
C	Toiture	1	B
D	Installation électrique	8	A
E	Façade	2	D,C
F	Fenêtres	1	D,C
G	Aménagement du jardin	1	D,C
H	Travaux de plafonnage	3	F
J	Mise en peinture	2	H
K	Emménagement	1	E,G,J

FIGURE 1 – Liste des tâches, durée et contraintes.

7/152

Définitions formelles et notations : chemins, circuit, chaîne, cycle

- Un *chemin* P dans un graphe orienté G est une suite d'arcs dont les extrémités droites/gauches coïncident de la façon suivante : $P = ((u_0, u_1), (u_1, u_2), (u_2, u_3), \ldots, (u_{k-1}, u_k))$. La longueur du P est le nombre d'arcs (ici k).
- Si $u_0 = u_k$, le chemin est appelé *circuit*.
- Un *chemin élémentaire* est défini par une suite de sommets sans répétition (sauf pour le premier et le dernier sommet).
- Si le graphe est non orienté, on utilise *chaîne* et *cycle* à la place de chemin et circuit.
- Un graphe sans cycle/circuit est appelé *acyclique / sans circuit*.
- Un graphe non orienté tel que chaque couple de sommets est connecté par une chaîne est dit *connexe*.
- Un graphe orienté tel que chaque couple de sommets (u, v) est connecté par un chemin dans les deux sens (c.-à-d. de u à v et de v à u) est dit *fortement connexe*. On dit aussi que les sommets u et v sont mutuellement accessibles.
- Un graphe, non orienté, connexe et acyclique est dit *arbre*.

8/152

Modéliser l'accessibilité au centre-ville de l'Utopia (examen 2011/12)

Un ingénieur du service technique de la ville Utopia propose, pour la zone centrale, le plan sens unique représenté à la figure (2). Avant d'adopter le plan, il convient d'examiner s'il autorise la circulation des automobiles, c'est à dire s'il permet d'aller de n'importe quel point à n'importe quel autre.

1. Enumérer les algorithmes vus en cours pouvant résoudre ce problème. Donner la complexité de ces algorithmes pour un graphe quelconque $G = (V, E)$.
2. Appliquer l'algorithme de votre choix sur le graphe de la figure. Le calcul à chaque itération sera clairement indiqué.
3. Vous constaterez facilement que le plan n'est pas acceptable. Quelles sont les zones où les sommets sont mutuellement accessibles ? Quelle est la dénomination de ces zones dans la terminologie spécifique pour l'algorithme en considération ?
4. Trouver le nombre minimum de rues tel que l'inversion du sens de circulation dans chacune de ces rues rend ce plan acceptable. Combien de solutions avez vous trouvées ? Quelles sont ces solutions ?

9/152

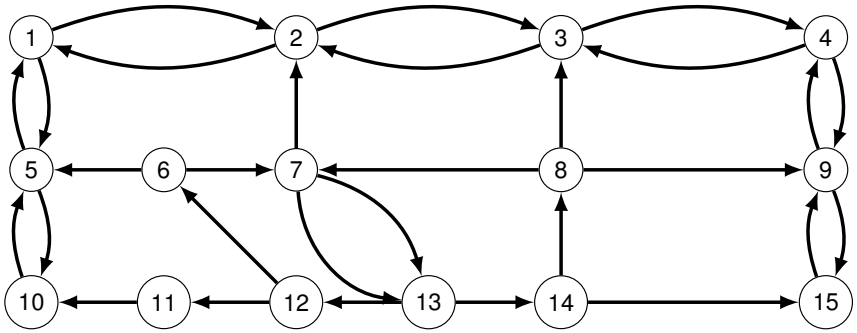


FIGURE 2 – Le plan de sens uniques du centre-ville.

10/152

Soit le graphe orienté $G = (V, E)$.

- Pour un arc $(x, y) \in E$, x est l'origine et y est l'extrémité de l'arc.
  On dit aussi que y est successeur de x, ou que x est prédécesseur de y.
- $\Gamma^+(x) = \{y \in V \mid (x, y) \in E\}$ est l'ensemble des successeurs de x.
- $\Gamma^{-1}(x) = \{z \in V | (z, x) \in E\}$ est l'ensemble des prédécesseurs de x.
- $d^+(x) = |\Gamma(x)|$ est le degré extérieur de x.
- $d^-(x) = |\Gamma^{-1}(x)|$ est le degré intérieur de x.
- $d(x) = d^+(x) + d^-(x)$ est le degré de x.
- Un chemin P allant de x à y, dont on ne précise pas les sommets intermédiaires, sera noté : $P = x \rightsquigarrow y$. y est alors un descendant de x et x un ascendant de y.

11/152

Soit le graphe $G = (V, E)$ où $|V| = n$ et $|E| = m$. On dit parfois que G est d'ordre n.

- Deux sommets sont *indépendants* s'ils ne sont pas connectés, c'est-à-dire pas *adjacents*.
- Il existe 2 cas extrêmes pour l'ensemble de ses arêtes : soit le graphe n'a aucune arête : on parle alors de *stable*. Soit toutes les arêtes possibles pouvant relier les sommets 2 à 2 sont présentes : le graphe est dit alors *complet*.
- On dit aussi qu'ensemble de sommets est *indépendant (ou stable)* s'il n'y a pas deux de ses sommets adjacents.
- Une clique est un sous-graphe complet.
- Un stable est un sous-graphe sans arête.

Pour un graphe général, il est souvent intéressant de rechercher de tels sous-graphes.

12/152

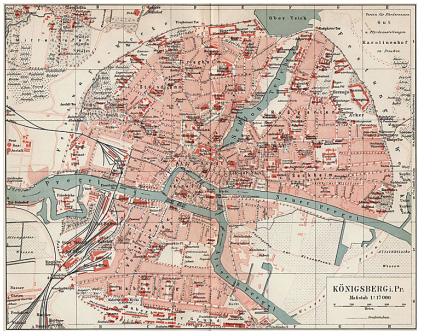
Exemple de graphes : suite

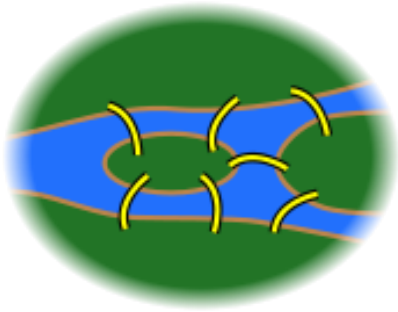
Planning d'examen

Les cinq étudiants : Dupont, Dupond, Durand, Duval et Duduche doivent passer certaines épreuves parmi les suivants : Français, Anglais, Dessin, Couture, Mécanique et Solfège. L'examen se déroulant par écrit, on désire que tous les étudiants qui doivent subir une même épreuve le fassent simultanément. Chaque étudiant ne peut se présenter qu'à une épreuve au plus chaque jour. Ci-dessous la liste des épreuves que doit passer chaque étudiant : Dupont : Français, Anglais, Mécanique ; Dupond : Dessin, Couture ; Durand : Anglais, Solfège ; Duval : Dessin, Couture, Mécanique ; Duduche : Dessin, Solfège.

1. Quel est le nombre maximal d'épreuves que l'on peut organiser le même jour ?
2. Quel est le nombre minimal de jours nécessaires à l'organisation de toutes les épreuves ?

Exemple des graphes III : les 7 ponts de Königsberg





La ville de Königsberg sur le Pregel et ses 7 ponts. Déterminer s'il existe ou non une promenade dans les rues de Königsberg permettant, à partir d'un point de départ au choix, de passer une et une seule fois par chaque pont, et de revenir à son point de départ. Ce problème (résolu par Leonhard Euler en 1736) est considéré comme l'origine de la théorie des graphes.

Vocabulaire et modélisation

Peut-on dessiner sans lever le crayon et en ne passant qu'une seule fois sur chaque arête les graphes suivants ?
La réponse de cette question est liée à l'existence d'une chaîne eulérienne/cycle eulérien).

- On appelle chaîne eulérienne (resp. cycle eulérien) une chaîne (resp. un cycle) qui emprunte une fois et une seule chaque arête du graphe.
- **Théorème d'Euler :** Un graphe connexe a une chaîne eulérienne si et seulement si tous ses sommets ont un degré pair ou tous ses sommets ont un degré pair sauf deux sommets.
  On peut démontrer que :
  - si le graphe n'a pas de sommet impair, alors il a un cycle eulérien.
  - un graphe ayant plus de deux sommets impairs ne possède pas de chaîne eulérienne (donc non pour le graphe de la ville de Königsberg).
  - si le graphe a deux sommets impairs, ce sont les extrémités de la chaîne eulérienne.

Graphe de de Bruijn

- Un graphe de de Bruijn est un graphe orienté qui permet de représenter les chevauchements de longueur $n-1$ entre tous les mots de longueur n sur un alphabet donné.
  Le graphe de de Bruijn $B(k,n)$ d'ordre n sur un alphabet A à k lettres est construit comme suit. Les sommets de $B(k,n)$ sont étiquetés par les k^n mots de longueur n sur A. Si u et v sont deux sommets, il y a un arc de u à v s'il existe deux lettres a et b, et un mot x, tels que $u = ax$ et $v = xb$. La présence d'un arc signifie donc un chevauchement maximal entre deux mots de même longueur.
- Le graphe $B(2,3)$ ci-contre est construit sur un alphabet binaire A=$\{0,1\}$ pour des mots de longueur $n=3$.

Exemple de graphes : suite

Application 2 : Le problème de mariage (couplage)

Soit $G = (V, E)$ un graphe. Rappel :

- Un couplage M est un ensemble d'arêtes deux à deux non adjacentes.
- Un ensemble stable S est un ensemble de sommets 2 à 2 non adjacents.
- Une couverture C est un ensemble de sommets tel que chaque arête est incidente avec au moins un sommet de C.

Soit un graphe biparti $G(U \cup V, E)$. L'ensemble des arêtes représente les couples compatibles. Il s'agit de trouver le couplage maximal.

- Montrer que le problème du couplage maximal se réduit au problème du flot maximal.

Exemple :

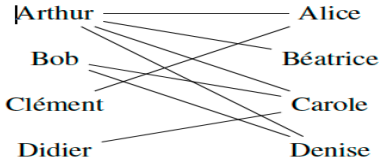
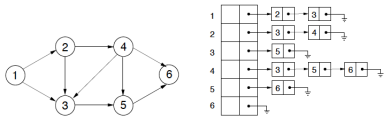


FIGURE 3 – Tentons à marier ces personnes en satisfaisant les contraintes des couples compatibles.

Représentations des graphes

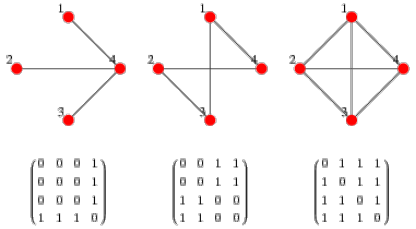
Soit le graphe $G = (V, E)$ où $|V| = n$ et $|E| = m$. G peut être représenté en mémoire soit par des listes d'adjacence, soit par une matrice d'adjacence.

- Un graphe et sa représentation par des listes de successeurs.



- La matrice d'adjacence d'un graphe $G = (V, E)$ où $|V| = n$, est une matrice carrée de taille n et définie par :

$$A(u,v) = \begin{cases} 1 & \text{si } (u,v) \in E \\ 0 & \text{sinon} \end{cases}$$

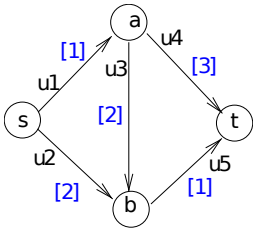


Représentations des graphes : matrice d'incidence

Soit le graphe $G = (V, E)$ où $|V| = n$ et $|E| = m$.

- La matrice d'incidence sommet/arc d'un graphe orienté est une matrice $A = [a_{i,j}]$ $i = 1, 2, \ldots, |V|$ et $j = 1, 2, \ldots, |E|$ telle que :

$$a_{i,j} = \begin{cases} +1 & \text{si l'arc } u_j \text{ est sortant pour le sommet } i \\ -1 & \text{si l'arc } u_j \text{ est entrant pour le sommet } i \\ 0 & \text{sinon} \end{cases}$$



$A =$

	$u1$	$u2$	$u3$	$u4$	$u5$
s	$+1$	$+1$	0	0	0
t	0	0	0	-1	-1
a	-1	0	$+1$	$+1$	0
b	0	-1	-1	0	$+1$

Remarque : les valeurs en bleu sont les poids (longueurs) des arcs.

Le problème du flot maximum

En effet :

- Soit $R = (X, U, C)$ un graphe connexe orienté (réseau). A $\forall$ arc u on affecte une valeur (*capacité*) c_u qui est une borne supérieure du flux sur l'arc.
- Soit deux sommets particuliers $s \in X$ (source) et $t \in X$ (puits). Considérons le graphe $G^0 = (X, U^0)$ où $U^0 = U \cup \{t, s\}$. L'arc (t, s) est appelé l'*arc de retour* du flot (numéroté 0). Notons $M = |U|$.
- On dit que $[\phi_1, \phi_2, \ldots \phi_M]^T$ est un *flot de s à t* ssi la loi de conservation du flot est vraie en tout $\forall u \in X \backslash \{s, t\}$, i.e.

$$\sum_{(u,v) \in \Gamma^+(u)} \phi_{(u,v)} = \sum_{(v,u) \in \Gamma^-(u)} \phi_{(v,u)} \quad (1)$$

- La valeur du flot est notée par ϕ_0. Elle est définie par

$$\sum_{v \in \Gamma^+(s)} \phi_{(s,v)} = \sum_{u \in \Gamma^-(t)} \phi_{(u,t)} = \phi_0 \text{ (valeur du flot)} \quad (2)$$

- But : Trouver dans G^0 un *flot compatible* $\phi' = [\phi_0, \phi_1, \phi_2, \ldots \phi_M]$ (c.-à-d.

$$0 \leq \phi_u \leq c_u, \forall u = \in U \quad (3)$$

et tel que ϕ_0 soit *maximale*.

- Appliquer ce modèle pour le problème des cases admissibles.

Exemple de graphes : suite

Application 2 : Application 1 : Problème des cases admissibles

Soit le tableau T de taille $m \times n$, dont certaines cases sont dites “admissibles” (les autres sont “non admissibles”. On se donne également $m+n$ entiers positifs ou nuls $l_1, \ldots, l_m, c_1 \ldots, c_n$. Il s’agit d’affecter des nombres entiers aux cases admissibles (et à celles-ci seulement) de telle sorte que :

- la somme des nombres affectés aux cases d’une ligne i soit inférieure ou égale à l_i ($i = 1, 2, \ldots, m$).
- la somme des nombres affectés aux cases d’une colonne j soit inférieure ou égale à c_j ($j = 1, 2, \ldots, n$).
- la somme des nombres affectés aux cases du tableau soit maximum.

1. Formuler le problème des cases admissibles comme un problème de flots maximum en décrivant le graphe approprié.

Problème des cases admissibles (suite)

Considérons l’instance suivante :

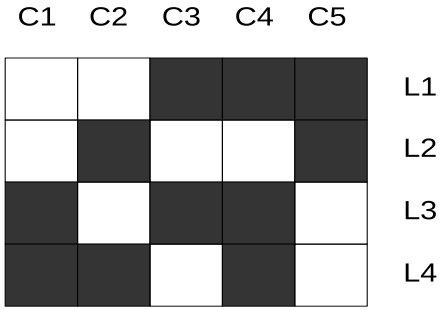


FIGURE 4 – Exemple du problème de cases admissibles. Les cases interdites sont en gris.

- Taille du tableau : $m = 4$ et $n = 5$.
- Valeurs des lignes : $l_1 = 9, l_2 = 10, l_3 = 15, l_4 = 2$.
- Valeurs des colonnes : $c_1 = 7$, $c_2 = 5$, $c_3 = 9$, $c_4 = 4$, $c_5 = 8$.
- Cases interdites : t_{13}, t_{14}, t_{15}, t_{22}, t_{25}, t_{31}, t_{33}, t_{34}, t_{41}, t_{42}, t_{44}.

Exemple de graphes : suite I

Le problème du chou, de la chèvre et du loup

Un passeur, disposant d’une barque, doit faire traverser une rivière à un chou, une chèvre et un loup. Outre le passeur, la barque peut contenir au plus une des trois unités qui doivent traverser. D’autre part, pour des raisons de sécurité, le passeur ne doit jamais laisser seuls la chèvre et le chou, ou le loup et la chère. (Par contre, le loup ne mangera pas le chou... et réciproquement). Comment le passeur doit-il s’y prendre ?

Modélisation

Il s’agit d’un système à états possédant un état initial, un état final, et des transitions autorisées. Donner le graphe des transitions du problème problème du chou, de la chèvre et du loup.

Suggestion

Dans cet exemple, un état est une configuration possible sur la rive de départ.

Exemple de graphes : suite II

Blocage mutuel dans un partage de ressources

3 philosophes chinois A, B, C possèdent, à eux trois, trois baguettes R, S, T. Pour manger, chacun a besoin de 2 baguettes. Lorsqu’un des philosophes désire manger, il requiert deuxbaguettes et ne les libère que lorsqu’il a fini son repas 1. Par contre, tant qu’il n ?a pas obtenu les deux baguettes, il ne peut rien faire qu’attendre : même s’il a eu la chance d’obtenir une baguette, il ne la rend pas tant qu’il n’a pas pu aller au bout de son repas.
Considérons la situation suivante : les 3 philosophes ont envie de manger exactement au même moment, et chacun se précipite sur les baguettes... mais chacun n’en obtient qu’une.Pour fixer les idées :
A possède R et requiert S ; B possède S et requiert T ; C possède T et requiert R
Les philosophes ne peuvent pas sortir de ce blocage mutuel ! !

Modélisation

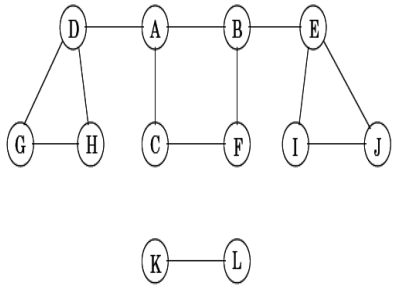
Cette situation peut se produire dans les systèmes informatiques complexes, où plusieurs entités (processeurs, périphériques,etc.) se partagent des ressources.

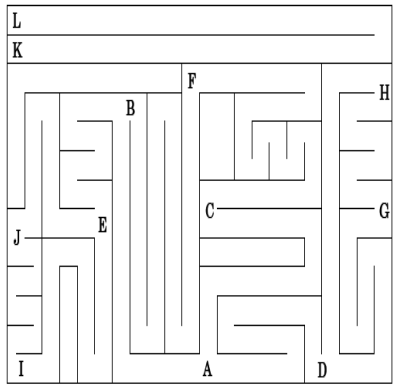
- Proposer une modélisation adéquate.
- A quoi correspond la situation d’interblockage ?

Exploration des graphes : Parcours en profondeur et parcours en largeur

Exploration des graphes

- L'exploration d'un graphe (c.-à-d. la visite de tous les sommets joignables à partir d'un sommet de départ) est une opération fréquente et importante.
- elle ressemble à l'exploration d'un labyrinthe
- pour explorer un labyrinthe il faut :
  - une craie : pour noter les carrefours/couloirs déjà visités
  - un fil : pour pouvoir retourner sur ses pas (backtrack)





Parcours en profondeur : 1^e version itérative

Algorithm 1 DFS(G,s) (utilise une **pile** P et suit la stratégie **LIFO** (Last In First Out)).

Require: G=(V,E), start vertex s.
Ensure: v.visited = true for all vertices $v \in V$ reachable from s.

1: **Initialisation :** $\forall v \in V$: v.visited $\leftarrow$ false ; clock $\leftarrow 1$; previsit(s) ; P $\leftarrow$ [s] ;
2: $u \leftarrow P.head()$ (u reçoit le sommet de la pile P)
3: **while** ($u \neq nil$) **do**
4: **if** ($\exists(u,v) \in E \mid v.visited = false$) **then**
5: { previsit(v) ; P.put(v) } { v est inséré dans P }
6: **else**
7: { postvisit(u) ; delete(u,P) } { le sommet u est supprimé de P }
8: **end if**
9: $u \leftarrow P.head()$ { u reçoit le sommet de la pile P }
10: **end while**

where :

- previsit(v)={v.visited $\leftarrow$ true ; v.in $\leftarrow$ clock ; clock $\leftarrow$ clock+1}
- postvisit(v)={v.out $\leftarrow$ clock ; clock$\leftarrow$ clock+1}

Parcours enigme

Algorithm 2 Engm(G,s) (utilise une **file** F et suit la stratégie **FIFO**)First In First Out)).

Require: G=(V,E), start vertex s.
Ensure: to be discovered

1: **Initialisation :** $\forall v \in V$: v.visited $\leftarrow$ false ; clock$\leftarrow 1$; previsit(s) ; F $\leftarrow$ [s] ;
2: $u \leftarrow P.head()$ (u reçoit le sommet de la file F)
3: **while** ($u \neq nil$) **do**
4: **if** ($\exists(u,v) \in E \mid v.visited = false$) **then**
5: { previsit(v) ; P.put(v) } {v est inséré dans P}
6: **else**
7: { postvisit(u) ; delete(u,P) } {le sommet u est supprimé de P}
8: **end if**
9: $u \leftarrow P.head()$ {u reçoit le sommet de la pile P}
10: **end while**

where :

- previsit(v)={v.visited $\leftarrow$ true ; v.in $\leftarrow$ clock ; clock $\leftarrow$ clock+1}
- postvisit(v)={v.out $\leftarrow$ clock ; clock$\leftarrow$ clock+1}

Une application

- Appliquer l'algorithme (**??**) sur le graphe (5).
- Appliquer l'algorithme (2) sur le graphe (5).
- Commenter les différences constatées.

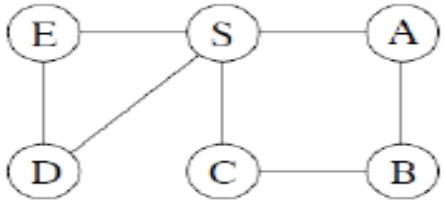


FIGURE 5 – Un graphe non orienté G.

29/152

Parcours en largeur d'abord et calcul des PCC de s aux autres sommets

- $G = (V, E) \mid \forall (u, v) \in E,\ l(u, v) = 1$ (la longueur de chaque arête vaut 1).
- La distance du sommet s au sommet v est la longueur du PCC($s \rightsquigarrow v$).
- Une version de l'algorithme Enigme est donnée par :

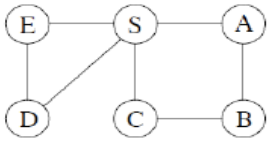
Algorithm 3 BFS(G,s) (2^e version du parcours en largeur d'abord).

Require: G=(V,E), $\forall (u, v) \in E$ l(u,v)=1, start vertex s.
Ensure: For all vertices u reachable from s, u.dist is set to the distance from s to u.

```
1: Initialisation : ∀v ∈ V : v.visited ← false and v.dist = ∞ ; F ← [s] ; s.dist ← 0 ;
   s.visited ← true ;
2: while (F ≠ ∅ ) do
3:    u ← eject(F) { u reçoit le sommet de la file F, le dernier est supprimé de F }
4:    while (∃(u, v) ∈ E | v.visited = false ) do
5:       v.visited ← true
6:       v.dist ← u.dist+1
7:       inject(F,v) { le sommet v est inséré dans F }
8:    end while
9: end while
```


30/152

Application



Order of visitation	Queue contents after processing node
	[S]
S	[A C D E]
A	[C D E B]
C	[D E B]
D	[E B]
E	[B]
B	[ ]

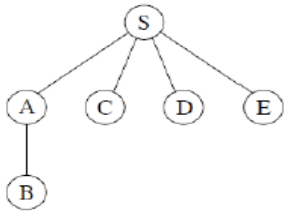


FIGURE 6 – Illustration du fonctionnement de l'algorithme 4.

Remarque : tous les chemins partant de la racine s sont des PCC (c.-à-d. que ceci est un arbre des PCC de s à tous les autres sommets).

31/152

BFS(G,s) : validation formelle

- **Proposition** : BFS(G,s) visite chaque sommet joignable à partir de s.
- *Preuve* : Par induction à la base des propriétés suivantes :
- Lors de l'exécution de BFS(G,s), pour chaque $d = 0, 1, 2, \ldots,$ *max_dist* il existe un moment tel que
  - pour chaque v éloigné de s à une distance $\leq d$, la valeur $v.dist$ a été correctement calculée.
  - pour tous les autres sommets $u, u.dist = \infty$.
  - la file F contient uniquement les sommets u éloigné de s à une distance égale à d.
- Analyse de complexité (similaire à l'analyse de DFS(G,s)) :
  - chaque sommet est inséré une fois dans F lors de sa première visite, et est éjecté de F lorsque tous ses voisins ont été visités (en total $2 \times |V|$ opérations insertions/éjections) .
  - Chaque arête (u,v)$\in$ E est examinée exactement deux fois (en cas d'un graphe non orienté) ; chaque arc est examiné une seule fois (en cas d'un graphe orienté). Donc, $O(|E|)$ opérations sur les arêtes/arcs.
  - en total $O(|V| + |E|)$.

32/152

Parcours en profondeur d'abord : version récursive

Exploration de la descendance d'un sommet donné v

Algorithm 4 Explore(G,v).

Require: G=(V,E), start vertex v∈ V.
Ensure: u.visited = true for all vertices u reachable from v.

```
1: v.visited ← true
2: previsit(v)
3: for all each edge (v,u)∈E do
4:     if not u.visited then Explore(G,u)
5: end for
6: postvisit(v)
```

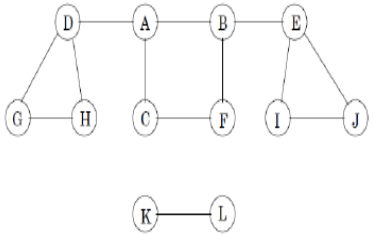
- Les procédures « previsit(v) » et « postvisit(v) » contiennent des opérations optionnelles à effectuer lors de la première visite du sommet v , et après l'avoir complètement explorer. Ces procédures peuvent être extrêmement utiles.

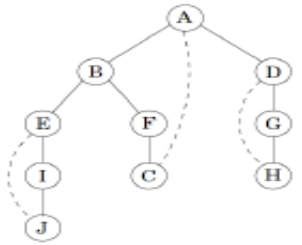
Explore(G,v) : validation formelle

- **Proposition** : Explore(G,v) visite chaque sommet joignable à partir de v.
- *Preuve* : Supposons que le sommet u est joignable à partir de v, mais n'a pas été visité par Explore(G,v). Notons P le chemin allant de v à u. Considérons le dernier sommet z visité par Explore(G,v) sur ce chemin et soit w le sommet qui suit immédiatement z sur P.

- D'après la définition d'Explore(G,v), w devrait être aussi visité ; contradiction.

Explore(G,v) : arbre de parcours et classification des arêtes





- Cette figure illustre un parcours possible Explore(G,A).
- Les lignes « normales » indiquent les arêtes réellement traversées par l'algorithme à la recherche des sommets non visités ((arêtes de liaison) ou « *tree edges* »). Elles représentent l'arbre de parcours.
- Les lignes en pointillé indiquent des arêtes qui mènent vers des sommets déjà visités. Ces arêtes ne sont pas traversées, mais simplement examinées à partir du sommet courant pour détecter si l'autre extrémité de l'arête a déjà été visitée ou non ((arêtes retour) ou « *back edges* »)

Parcours en profondeur d'abord : l'algorithme complet

- Puisque le graphe G peut avoir plusieurs composantes distinctes, plusieurs appels d'Explore(G,.) à partir de différents sommets de départ sont éventuellement nécessaires.

Algorithm 5 DFS(G) : algorithme **depth-first search**.

Require: G=(V,E).
Ensure: v.visited = true for all vertices v∈ V.

```
1: for all v∈V do
2:     v.visited ← false
3: end for
4: for all v∈V do
5:     if not v.visited then Explore(G,v)
6: end for
```

- Analyse de complexité :
  - $\forall v \in V$ Explore(G,v) est appelée une seule fois (grâce à l'étiquette v.visited).
  - Chaque arête (u,v)∈ E est examiné exactement deux fois : la première fois lors d'Explore(G,u) et une deuxième fois lors d'Explore(G,v).
  - Sous l'hypothèse que previsit et postvisit prennent un temps constant, DFS(G) nécessite un temps $O(|V|+|E|)$ (linéaire).

37/152

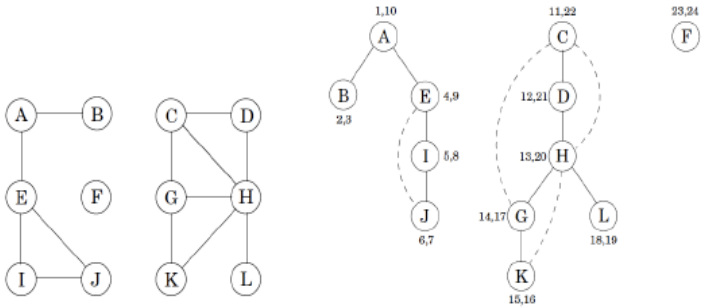
Calcul des étiquettes in/out (prev/post)

- Les valeurs in/out peuvent être calculées dans les procédures previsit et postvisit.
- On utilise pour ce faire le compteur « clock » initialisé à 1 et défini comme suit.
  - previsit(v)={v.in ← clock ; clock ← clock+1}
  - postvisit(v)={v.out ← clock ; clock ← clock+1}
- **Proposition** : Pour chaque couple de sommet u et v, les intervalles (u.in,u.out) et (v.in,v.out) sont soit complètement disjoints, soit l'un est entièrement inclus dans l'autre.

38/152

Parcours en profondeur d'abord : example

- En général, DFS(G) engendre une foret qui contient plusieurs arbres de parcours, une pour chaque composante connexe du graphe.



- Lors du parcours, chaque sommet v est muni de deux étiquettes (v.in,v.out). On utilise aussi les notations (prev[v],post[v]). v.in indique le moment de la première visite du sommet v. v.out indique le moment quand la procédure DFS a définitivement quitté le sommet v (il a été donc complètement exploré).

39/152

Parcours en profondeur dans les graphes orientés

- DFS(G) s'applique d'une façon similaire dans les graphes orientés. En raison de l'orientation des arcs, les arbres de parcours engendrés sont plus complexes (appelés aussi arborescences).

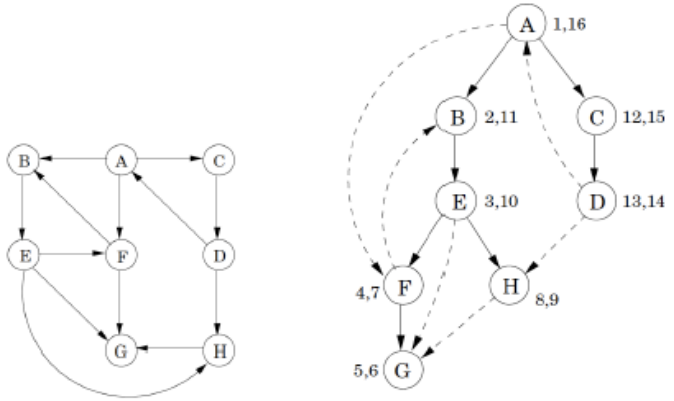
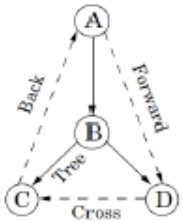


FIGURE 7 – Applquer le parcours en profondeur sur le graphe au-dessus.

40/152

Parcours en profondeur dans les graphes orientés : classification des arcs



- Les arcs « normaux » indiquent les arcs de liaison (« *tree edges* »). L'arc (u,v) est dit de liaison si v a été découvert la première fois lors de la traversé de l'arc (u,v). Ces arcs représentent l'arbre de parcours.
- Les arcs en pointillé peuvent être de trois types différents.
  - Les arcs retour (« *back edges* ») sont les arcs (u,v) reliant un sommet u à un ancêtre v. Les boucles sont considérées comme des arcs retour.
  - Les arcs avant (« *forward edges* ») sont les arcs (u,v) qui relient un sommet u à un descendant v (qui n'est pas son fils).
  - Les arcs couvrant (« *cross edges* ») sont tous les autres arcs. Ils peuvent relier deux sommets d'une même arborescence, tant que l'un des sommets n'est pas ancêtre de l'autre ; ils peuvent aussi relier deux sommets appartenant à des arborescences différentes.

Relations entre le type des arcs et les étiquettes pre/post

- Dans une forêt DFS, le sommet u est un ancêtre du sommet v ssi u est visité/découvert le premier et v est visité/découvert lors de l'appel explore(u) (c.-à-d. pre(u) < pre(v) < post(v) < post(u)).
- On peut alors donner la classification suivante :

pre/post *ordering for* (u, v)	*Edge type*
$[_u\ [_v\ ]_v\ ]_u$	Tree/forward
$[_v\ [_u\ ]_u\ ]_v$	Back
$[_v\ ]_v\ [_u\ ]_u$	Cross

Application de DFS : linéarisation d'un DAG (Directed acyclic graph)

Le parcours en profondeur peut être utilisé pour effectuer un tri topologique (ou linéarisation) d'un graphe orienté sans circuit. Le tri topologique d'un graphe orienté acyclique $G = (S, A)$ consiste à ordonner linéairement tous ses sommets de telle sorte que si G contient un arc entre s_i et s_j, s_i apparaisse avant s_j.
Les propositions suivantes sont utilisées pour effectuer un tri topologique.

- **Proposition** : Un graphe orienté G contient un circuit, ssi la forêt engendrée pars le parcours DFS(G) contient un arc retour.
- *Preuve* :
  - Evident si le parcours DFS(G) contient un arc retour.
  - Supposons maintenant que G contient un circuit $C : v_0 \to v_1 \to v_2 \to \ldots \to v_k \to v_0$. Soit v_i le premier sommet du C qui a été visité durant le parcours DFS. Tous les autres sommets de C sont des descendants de v_i dans l'arbre de parcours et alors $v_{i-1} \to v_i$ est un arc retour (si $i = 0$ alors $v_k \to v_0$).

Application de DFS : linéarisation d'un DAG (suite)

- **Proposition** : La numérotation post d'un DAG obtenue lors d'un parcours DFS satisfait l'inégalité $post(u) > post(v)$ pour chaque arc (u, v).
- *Preuve* : Les seuls arcs (u, v) pour lesquels $post(u) < post(v)$ dans l'arbre DFS sont les arc retour. Or, il n'y pas de tels arcs dans un DAG.

Algorithm 6 Algorithme pour linéariser un DAG.

Require: G=(V,E), DAG
Ensure: List vertices in linear order.
1: Perform a DFS search and obtain the number post(v) for any vertex $v \in V$.
2: List vertices in order of decreasing post(v) values.

- **Propriété** : Chaque DAG a au moins un sommet source (sommet sans prédécesseurs) et au moins un sommet puits (sommet sans successeurs)
- *Preuve* : Ce sont respectivement le sommet avec la plus grande, et la plus petit valeur de la numérotation post.

Algorithm 7 Autre algorithme pour linéariser un DAG.

1: Find a source, output it, and delete it from the graph.
2: Repeat until the graph is empty.

Application de DFS : linéarisation d'un DAG (suite)

Exercice Appliquer les algorithmes 16 et 17 sur le graphe de la figure 7.

Composantes fortement connexes

Application de DFS : Composantes fortement connexes

- Soit le graphe orienté $G = (V, E)$. G est dit *fortement connexe*, si pour chaque couple de sommets (u, v), il existe un chemin de u à v, noté $(u \rightsquigarrow v)$, et un chemin de v à u, noté $(v \rightsquigarrow u)$.
- Cette définition définit une relation binaire dans V.

$$uRv \equiv (u \in \Gamma_v^*) \wedge (v \in \Gamma_u^*) \tag{4}$$

  où Γ_v^* indique la descendance du sommet v.
- C'est une relation d'équivalence dont les classe s'appellent compostantes fortement connexes (c.f.c.).
- Elle partitionne V en ensembles disjoints (appelés c.f.c.).

Application de DFS : Composantes fortement connexes (suite)

- La détection des c.f.c. se fait facilement grâce aux propriétés du parcours DFS.
- **Propriété 1** : La procédure explore(G,u) ne se termine que si tous les sommets accessibles à partir de u soient visités/explorés (c.-à-d. après avoir exploré toute la descendance Γ_u^* du sommet u).
- Ainsi, si la procédure *explore*(G, v) est exécutée à partir d'un sommet v appartenant à une c.f.c. *puits* (une c.f.c. sans arcs sortants), alors elle visiterait exactement cette c.f.c. La dernière pourrait être alors enlevée du graphe G, et le processus continuerait sur le graphe résultant G'.
- Illustrer cette propriété avec les deux meta-sommets {D} et {G,H,I,K,J,L} du graphe de la figure (8).

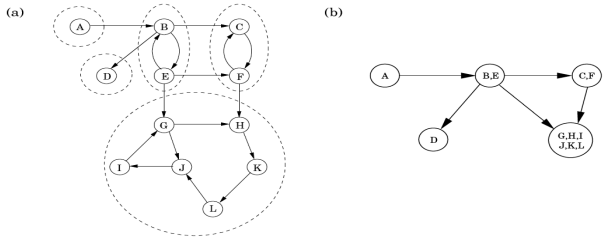


FIGURE 8 – a) Un graphe orienté G et ses c.f.c. b) Le graphe réduit $\tilde{G}$

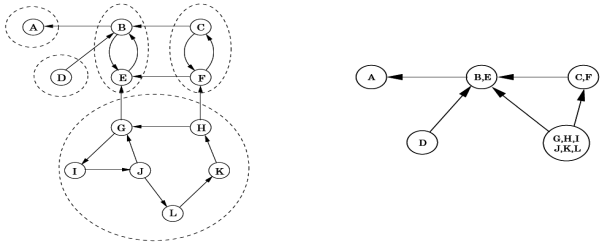


FIGURE 9 – a) G_R : le graphe renversé du graphe G et ses c.f.c. b) $\tilde{G_R}$: le graphe réduit de G_R

- Comment détecter une c.f.c. puits ?
- Trouver une c.f.c. puits est difficile, mais trouver une c.f.c. *source* (sans arcs entrants) est facile en raison de l'observation suivante.
- **Propriété 2** : Le sommet v avec la plus grande valeur $post(v)$ calculée lors d'un parcours DFS se trouve dans une c.f.c. source.
- Cette propriété est déduite du fait suivant.
- **Propriété 3** : Soit deux c.f.c. C et C', telles qu'il existe un arc d'un sommet de C vers un sommet de C', alors :

$$\max_{u \in C}\{post(u)\} > \max_{u \in C'}\{post(u)\} \tag{5}$$

- *Preuve* : Deux cas sont à considérer.
  - DFS commence à partir d'un sommet $u \in C$. Alors DFS visite Γ^*_u (tous les sommets appartenants à C et C') et $post(u) > post(v), \forall v \in C \cup C'$ (propriété 1).
  - DFS commence à partir d'un sommet $u \in C'$. Alors tous les sommets de C' sont visités avant de commencer la visite de C. Donc, $\max_{u \in C} post(u) > \max_{u \in C'} post(u)$.

- Grace à la propriété 3, nous savons comment trouver une c.f.c. source dans G.
- Déterminer les sommets appartenant aux c.f.c. nécessite que le parcours commence à partir d'une c.f.c. puits. Comment trouver une telle c.f.c. ?
- Idée : Renverser le graphe (c.-à-d. changer l'orientation de tous les arcs). Le graphe ainsi obtenu sera noté G_R.
- Alors les c.f.c. ne changent pas (facile à vérifier). Mais les sommets puits se transforment en sources et vice versa. Le graphe $\tilde{G}$ se renverse aussi (voir les graphes des figures (8) et (9).
- La linéarisation du DAG $\tilde{G_R}$ permet de trier topologiquement les sommets de DAG $\tilde{G}$ dans l'ordre des sommets-puits vers les sommets-sources.

Algorithm 8 Déterminer les composantes fortement connexes (CFC) d'un graphe orienté G.

Require: G=(V,E)
Ensure: Detect the strongly connected components (SCC) of G
1: Reverse all the edges in G, yielding digraph G_R.
2: Run DFS on G_R, obtaining the $post(v)$ numbers for all vertices v.
3: $k \leftarrow 1$.
4: Run EXPLORE(G,v) in G from the vertex v that has the highest $post(v)$ value in G_R, and has not yet been assigned to any SCC. Assign all vertices mapped out by the exploration into SCC k.
5: Set $k \leftarrow k+1$ and repeat from step 4, until all vertices have been assigned to SCCs.

- Complexité : linéaire.
- Appliquer l'algorithme 8 pour trouver les c.f.c. du graphe de fig. (8.a)

Calcul des Plus Courts Chemins (PCC) dans les graphes

Le cas des valeurs positives des arcs/arêtes

Calcul des PCC de s aux autres sommets dans le cas $\{l_e > 0 : e \in E\}$

Algorithm 9 Procédure Dijkstra_a (G,l,s).

Require: Graph G=(V,E) with positive edge weights $\{l_e > 0 : e \in E\}$, vertex $s \in V$
Ensure: $\forall u \in V$, $dist(u)$ is set to the shortest distance from s to u ; $prev(u)$ points to the previous of u on this shortest path

1: **Initialisation :** $\forall u \in V : dist(u) \leftarrow \infty$ and $prev(u) \leftarrow nil$; $dist(s) \leftarrow 0$;
2: $P \leftarrow \emptyset$, (P contient les sommets v, tq $dist(v)$ est calculée définitivement)
3: $T \leftarrow V$ (T contient des sommets v, tq $dist(v)$ n'est qu'une borne supérieure)
4: **while** T is not empty **do**
5: $u \leftarrow \underset{v \in (T)}{\operatorname{argmin}}\, dist(v)$
6: $P \leftarrow P \bigcup \{u\}$
7: $T \leftarrow T \backslash \{u\}$
8: **for all** edges $(u, v) \in E$ **do**
9: dist_update(u,v)
10: **end for**
11: **end while**

- où

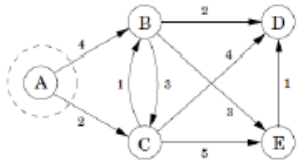
Algorithme de Dijkstra (suite)

Algorithm 10 Procédure dist_update(u,v).

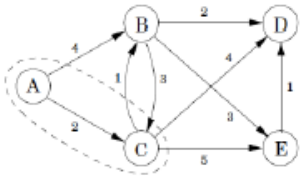
1: **if** $dist(v) > dist(u) + l(u, v)$ **then**
2: $dist(v) = dist(u) + l(u, v)$
3: $prev(v) \leftarrow u$
4: **end if**

Exemple :

Application de l'algorithme de Dijkstra sur un graphe



A: 0	D: ∞
B: 4	E: ∞
C: 2	



A: 0	D: 6
B: 3	E: 7
C: 2	

FIGURE 10 – A est le sommet de départ dans cet exemple. Les valeurs de dist sont données sont données dans la table associée.

Application de l'algorithme de Dijkstra sur un graphe

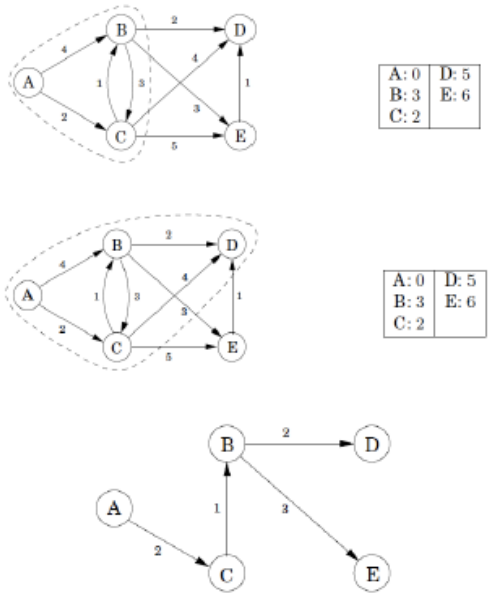


FIGURE 11 – A est le sommet de départ dans cet exemple. Les valeurs de dist sont données sont données dans la table associée. L'arbre des chemins les plus courts est aussi visualisé.

Justification de l'alg. de Dijkstra

- Soit $G = (V, E, l_e > 0 : e \in E)$. On cherche les PCC(s$\rightsquigarrow$u) : les chemins les plus courts du s aux $u \in V \backslash \{s\}$.
- Soit $dist^*(u)$: la longueur du PCC(s$\rightsquigarrow$u). Elle est calculée dans la variable $dist(u)$. A début $dist(u) = \infty$, à la fin du calcul, $dist(u) = dist^*(u)$.
- **Lemme :** Soit $u \in T$ tq $dist(u) = \min_{v \in T} dist(v)$ (c.-à-d.

$$u \leftarrow \underset{v \in (T)}{\operatorname{argmin}}\, dist(v)) \tag{6}$$

  Alors $dist^*(u) = dist(u)$.
- **Preuve :** Soit v tq $v = prev(u)$ sur le PCC(s$\rightsquigarrow$u). Puisque $l(v,u) > 0 \Rightarrow v \in P$ (sinon contradiction avec (6)). c.-à-d. une seule arête sépare u de l'ensemble P (voir Fig. (12)). Alors, $dist(u) = \min_{v \in P, v \in \Gamma^{-1}(u)} dist(v) + l(v,u) = dist^*(u)$.

Justification de l'alg. de Dijkstra (suite)

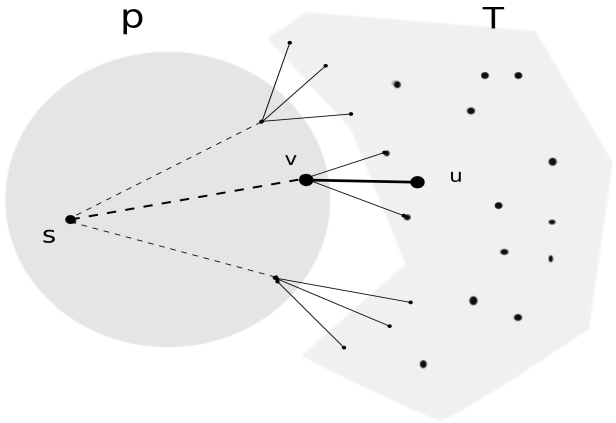


FIGURE 12 – P est élargi avec le sommet u tq $dist(u) = \min_{v \in P, v \in \Gamma^{-1}(u)} dist(v) + l(v,u) = dist^*(u)$.

On constate facilement que les étiquettes $dist(u)$ vérifient :

- si $u \in P$: $dist(u) = dist^*(u)$
- si $u \in T$: $dist(u) = \min_{v \in P, v \in \Gamma^{-1}(u)} dist(v) + l(v,u)$

Analyse de complexité

- Alg 1 nécessite $|V|$ opérations **argmin** plus $|E|$ opérations **update**. La complexité de ces opérations dépend du choix de la structure des données.
- Si on utilise **Array** : on obtient $O(|V|^2 + |E|)$.
- L'utilisation d'un tas binaire **Binary heap** (une file de priorité importante) améliore la complexité.
- **Définition :** Soit $\pi(j)$ la clé associée avec l'élément j de la file F. F est un tas binaire si :

$$\pi(j) \geq \pi(\lfloor \frac{j}{2} \rfloor) \quad \forall j \tag{7}$$

$$\iff \pi(j) \leq \pi(2j) \text{ and } \pi(j) \leq \pi(2j+1)\ \forall j$$

- On peut représenter le tas par un **arbre parfait partiellement ordonné** (c.-à-d.) :
  - $\forall$ niveau est rempli de gauche à droite ;
  - $\forall$ niveau est complètement rempli avant de commencer le niveau suivant ;
  - la condition 7) est satisfaite.
- Propriétés d'un tas de k éléments :
  - la racine contient le plus petit élément. L'opération d'« accès eu minimum » se fait en $\Theta(1)$;
  - l'opération « insertion » se fait en $\Theta(\log_2 k)$;
  - l'opération « suppression du minimum » se fait en $\Theta(\log_2 k)$.
- Le tas est facile à implémenter en tableau.

Tas binaire

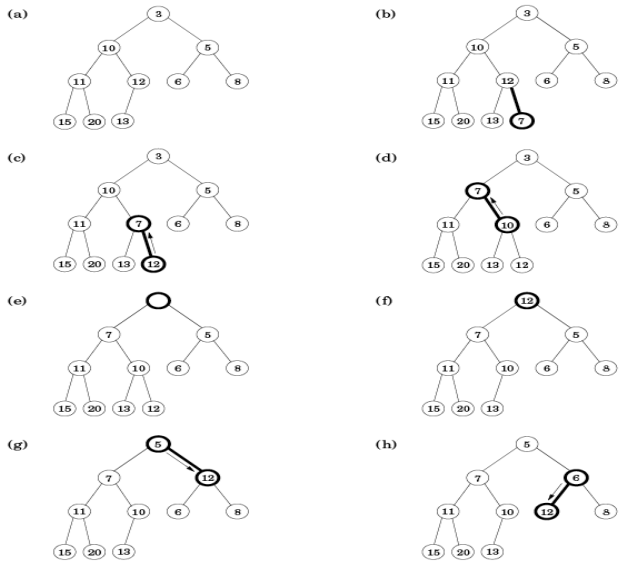


FIGURE 13 – (a) : un tas binaire avec 10 éléments. Seules les clés sont indiquées. (b)-(d) : illustration de l'opération « insertion ». (e)-(g) : illustration de l'opération « extraction et suppression du minimum ».

61/152

Version tas binaire de l'alg. de Dijkstra

Algorithm 11 Procédure Dijkstra_bis(G,l,s).

Require: Graphe G=(V,E) directed or undirected, with positive edge weights $\{l_e : e \in E\}$; vertex $s \in V$
Ensure: For all vertices $u \in V$, $dist(u)$ is set to the shortest distance from s to u ; $prev(u)$ points to the previous of u on this shortest path

1: $\forall u \in V$: initialize $dist(u) \leftarrow \infty$ and $prev(u) \leftarrow nil$
2: $dist(s) \leftarrow 0$
3: $P \leftarrow \{s\}$ {P contient les sommets v pour lesquels la valeur $dist(v)$ est calculée définitivement}
4: $T \leftarrow$ **makequeue(V)** {créer une file de priorité T avec $dist(v), v \in (T)$ comme clés}
5: **while** T is not empty **do**
6: $u \leftarrow$ **ExtractMin(T)**
7: $P \leftarrow P \bigcup \{u\}$
8: $T \leftarrow T \backslash \{u\}$
9: **for all** edges $(u,v) \in E$ **do**
10: dist_update_bis(T,(u,v))
11: **end for**
12: **end while**

62/152

Version tas binaire de l'alg. de Dijkstra (suite)

Algorithm 12 Procédure dist_update_bis(Q,(u,v)).

Require: Priority queue Q using *dist* as keys ; an edge $(u,v) \in E$
Ensure: Q is updated if the value of $dist(v)$ has been modified
if $dist(v) > dist(u) + l(u,v)$ **then**
$dist(v) \leftarrow dist(u) + l(u,v)$
$prev(v) \leftarrow u$
ChangeKey(Q,v) {Q is updated according to the new value of $dist(v)$ }
end if

Analyse de complexité de la version tas binaire de l'alg. de Dijkstra

- **makequeue(V)** nécessite $O(|V| \log |V|)$ opérations.
- la boucle **WHILE)** (line 6) nécessite $O(|V|)$ **ExtractMin** plus $O(|E|)$ **ChangeKey (updates)** : $O((|V|+|E|)\log|V|)$ opérations.
- En total : $O((|V|+|E|)\log|V|)$.

63/152

Calcul des chemins les plus courts dans les graphes

Présence de poids négatifs

64/152

Présence de poids négatifs

Les valeurs $dist(.)$ dans l'alg. de Dijkstra sont soit des valeurs exactes, soit des bornes supérieurs. La valeur $dist(u)$ équivaut $dist^*(u)$ à condition que tous les sommets intermédiaires du chemin s$\rightsquigarrow$u sont dans P. Les valeurs $dist(.)$ peuvent être vues comme une suite des mises à jours :

Algorithm 13 Procédure update($(u,v) \in E$).

$$dist(v) = \min\{dist(v), dist(u) + l(u,v)\}$$

Propriétés :

- $dist(v) = dist^*(v)$ si $dist(u) = dist^*(u)$ et si u est l'avant-dernier sommet ($u = prev(v)$) sur le PCC(s$\rightsquigarrow$v) ;
- la procédure update((u,v)) ne génère que des bornes supérieures à la valeur $dist^*(v)$.

Utilisation pour chercher le PCC(s$\rightsquigarrow$v) :

(a) $\mid PCC(s \rightsquigarrow v) \mid \leq \mid V \mid -1$;

(b) si update(.,.) est appliqué aux arêtes du PCC(s$\rightsquigarrow$v) $(s,u_1),(u_1,u_2),(u_2,u_3),\ldots(u,v))$ dans cet ordre, alors $dist(v) = dist^*(v)$.

65/152

Présence de poids négatifs : Algorithm de Bellman-Ford

Algorithm 14 Procédure Bellman-Ford (G,l,s).

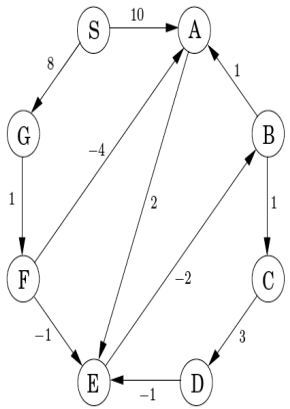
Require: Graphe G=(V,E) directed, edge weights $\{l_e : e \in E\}$ with no negative cycles ; vertex $s \in V$
Ensure: For all vertices $u \in V$ reachable from s, $dist(u)$ is set to the shortest distance from s to u ;

1: **for all** $u \in V$ **do**
2: $dist(u) \leftarrow \infty$
3: $prev(u) \leftarrow nil$
4: **end for**
5: $dist(s) \leftarrow 0$
6: $k \leftarrow 1$
7: **while** $(k \leq |V| - 1)$ **do**
8: **for all** edges $(u,v) \in E$ **do**
9: $dist(v) \leftarrow \min\{dist(v), dist(u) + l(u,v)\}$
10: **end for**
11: $k \leftarrow k + 1$
12: **end while**

- Complexité de l'algorithme (14) : $O(|V||E|)$

66/152

Illustration pour l'algorithme de Bellman-Ford



	Iteration							
Node	0	1	2	3	4	5	6	7
S	0	0	0	0	0	0	0	0
A	∞	10	10	5	5	5	5	5
B	∞	∞	∞	10	6	5	5	5
C	∞	∞	∞	∞	11	7	6	6
D	∞	∞	∞	∞	∞	14	10	9
E	∞	∞	12	8	7	7	7	7
F	∞	∞	9	9	9	9	9	9
G	∞	8	8	8	8	8	8	8

FIGURE 14 –

67/152

Algorithme de Bellman-Ford (variante)

Une autre approche est d'utiliser la fonction multivoque Γ^{-1} et l'observation que le (PCC(s$\rightsquigarrow$v)) passe par un des prédécesseurs $u \in \Gamma^{-1}(v)$. On doit donc avoir

$$dist(v) = \min_{u \in \Gamma^{-1}(v)} \{dist(u) + l(u,v)\} = \min\{dist(v), \min_{u \in \Gamma^{-1}(v)} \{dist(u) + l(u,v)\}\} \quad (8)$$

On obtient la variante suivante.

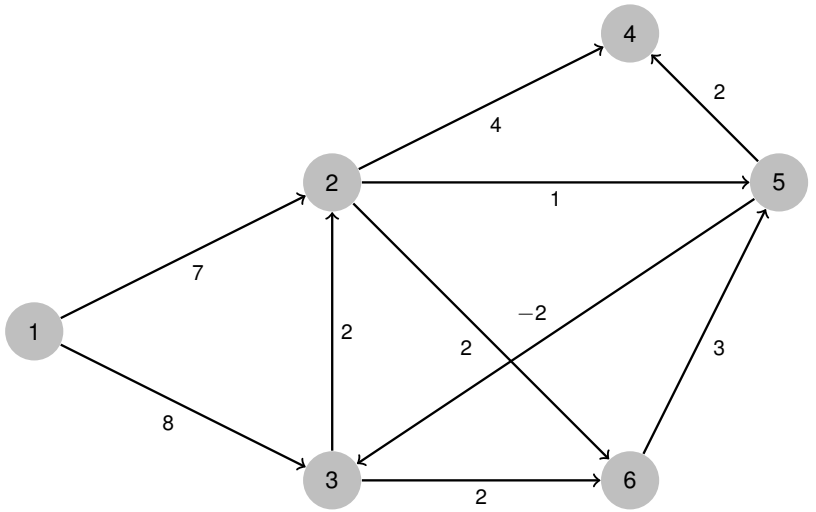
Algorithm 15 Procédure Bellman-Ford_bis (G,l,s).

Require: Graphe G=(V,E) directed, edge weights $\{l_e : e \in E\}$; vertex $s \in V$
Ensure: $dist^k(u)$ is set to the shortest distance from s to u over all paths with at most k arcs. It also detects the presence of negative cycle.

1: **Initialisation :** $\forall u \in V : dist^0(u) \leftarrow \infty$; $dist^0(s) \leftarrow 0$; stable $\leftarrow$ false ; $k \leftarrow 1$;
2: **while** ($(k \leq |V|)$ and (nonstable)) **do**
3: $dist^k(s) \leftarrow 0$
4: **for all** vertices $v \in V \setminus \{s\}$ **do**
5: $dist^k(v) = \min\{dist^{k-1}(v), \min_{u \in \Gamma^{-1}(v)} dist^{k-1}(u) + l(u,v)\}$
6: **end for**
7: stable $\leftarrow (dist^k(v) = dist^{k-1}(v)$, for all $v \in V$) ; $k \leftarrow k + 1$;
8: **end while**
9: **if** $(k = |V| + 1)$ **then** $\exists$ negative cycle
10: **end if**

68/152

- $dist^k(v)$ représente la valeur du $PCC(s \rightsquigarrow v)$ qui ne contient pas plus de k arcs.
- Dans l'algorithme (15) on a *le choix de l'ordre* à la ligne 4 et on a intérêt à définir un ordre astucieux sur les sommets.
- Par exemple si les l_e sont positifs et l'ordre est définit par l'alg de Dijkstra, alors l'alg. (15) converge en une seule étape.
- Si peu d'arcs ont une valeur négative, on a aussi intérêt à effectuer d'abord l'alg. de Dijkstra qui donnera une bonne initialisation des valeurs $dist(v)$ et on prendra alors comme ordre celui des $dist(v)$ croissants.



Calcul des chemins les plus courts/longues dans les graphes

L'approche programmation dynamique

- L'algorithme (15) résout un ensemble de sous-problèmes $\{dist^k(u), u \in V\}$ en commençant par les plus « petits » (c.-à-d. par ceux pour lesquels les valeurs de k sont petites), et en allant progressivement vers les plus « grands ».
- Pour les « plus petits » ($k = 0$), la solution est connue.
- C'est une méthode de résolution applicable aux problèmes qui satisfont le principe d'optimalité de Bellman (1955) : chaque sous-chemin d'un chemin optimal est lui-même optimal.
- C'est une approche très générique. La fonction min de la ligne (10) peut être remplacée par la fonction max et l'opérateur binaire « addition » par « multiplication ».

Algorithmes pour linéariser un DAG (graphe orienté sans circuit) (rappel)

Algorithm 16 Algorithme pour linéariser un DAG.

Propriétés importantes des DAG :

(a) Chaque DAG a au moins un sommet source (sommet sans prédécesseurs) et au moins un sommet puits (sommet sans successeurs)

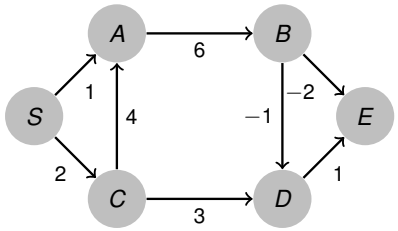
(b) Les sommets du graphe peuvent être numérotés dans l'ordre topologique (ils peuvent être placés sur une ligne de façon que tous les arcs soient orientés de gauche à droite).

Require: G=(V,E), DAG
Ensure: List vertices in linear order.
1: Perform a DFS search and obtain the number post(v) for any vertex $v \in V$.
2: List vertices in order of decreasing post(v) values.

Algorithm 17 Autre algorithme pour linéariser un DAG.

1: Find a source, output it, and delete it from the graph.
2: Repeat until the graph is empty.

Application de DFS : linéarisation d'un DAG (suite)



Exercice Appliquer les algorithmes 16 et 17 sur le graphe de la figure 7.

Application de l'ordre topologique/linéaire pour la détermination des chemins plus courts/longs dans un DAG

Algorithm 18 Procédure dag-shortest-paths(G,l,s).

Require: DAG G=(V,E), edge weights $\{l_e : e \in E\}$, vertex s$\in V$
Ensure: For all vertices u reachable from s, $dist(u)$ is set to the distance from s to u
1: **for all** $u \in V$ **do**
2: $dist(u) \leftarrow \infty$
3: **end for**
4: $dist(s) \leftarrow 0$
5: linearize G
6: **for all** $u \in V$, in linearized order **do**
7: **for all** edges $(u,v) \in E$ **do**
8: $dist(v) = \min\{dist(v), dist(u) + l(u,v)\}$
9: **end for**
10: **end for**

- Complexité de l'algorithme (22) : le coût de la linéarisation ($O(|V|+|E|)$) + le coût des boucles (6) et (7) ($\Sigma_u d^+ u$) = $O(|V|+|E|)$.

Application de l'ordre topologique/linéaire pour la détermination des chemins plus courts/longs dans un DAG

Algorithm 19 Procédure dag-shortest-paths(BIS)(G,l,s).

Require: DAG G=(V,E), edge weights $\{l_e : e \in E\}$, vertex s$\in V$
Ensure: For all vertices u reachable from s, $dist(u)$ is set to the distance from s to u
1: **for all** $u \in V$ **do**
2: $dist(u) \leftarrow \infty$
3: **end for**
4: $dist(s) \leftarrow 0$
5: linearize G
6: **for all** $v \in V \setminus \{s\}$, in linearized order **do**
7: **for all** edges $(u,v) \in E$ **do**
8: $dist(v) := \min\{dist(v), dist(u) + l(u,v)\}$
9: **end for**
10: **end for**

Cas des graphes sans circuit (DAG)

- Propriétés importantes des DAG :
  (a) Il existe au moins un sommet s tel que $\Gamma_s^{-1} = \emptyset$ (c.-à-d. s est sans prédécesseurs).
  (b) Les sommets du graphe peuvent être numérotés dans l'ordre topologique (ils peuvent être placés sur une ligne de façon que tous les arcs soient orientés de gauche à droite).
- Afin de générer un ordre topologique on peut, par exemple, utiliser la fonction rang (20).
- Pour introduire rapidement cette notion, nous supposerons que le graphe possède un seul sommet sans prédécesseurs (disons le sommet s).
- La fonction rang associe à chaque sommet $v \in V \backslash \{s\}$ le nombre $rank(v)$ qui correspond au nombre d'arcs dans un chemin de cardinalité maximum entre s et v.

Cas des graphes sans circuit (DAG)

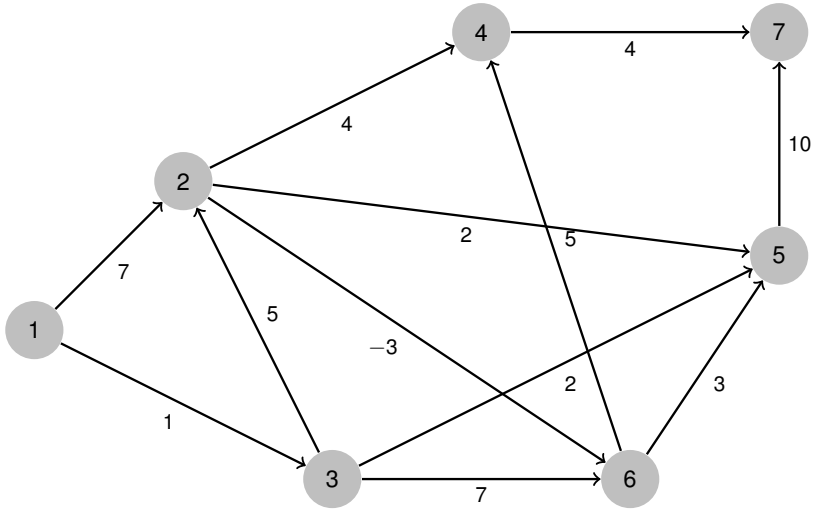
Algorithm 20 Procédure vertex-rank(G).

Require: DAG $G = (V, E)$, vertex s$\in V$ such that $d^-(s) = 0$.
Ensure: For all vertices u reachable from s, $rank(u)$ is set to the number of arcs in the longest (in cardinality) path from s to u

1: **Initialisation :** $\forall u \in V : d^-(u) \leftarrow |\Gamma^{-1}(u)|$; $k \leftarrow 0$; $S_0 \leftarrow \{s\}$;
2: **while** $|S_k| > 0$ **do**
3: $S_{k+1} \leftarrow \emptyset$
4: **for all** $u \in S_k$ **do**
5: $rank(u) \leftarrow k$
6: **for all** edges $(u, v) \in E$ **do**
7: $d^-(v) \leftarrow d^-(v) - 1$
8: **if** $(d^-(v) = 0)$ **then**
9: $S_{k+1} \leftarrow S_{k+1} \bigcup \{v\}$
10: **end if**
11: **end for**
12: **end for**
13: $k \leftarrow k + 1$
14: **end while**

- Complexité de l'algorithme (20) : $O(|V| + |E|)$.

Illustration pour l'algorithme **vertex-rank(G)**



Cas des graphes sans circuit (DAG) (suite)

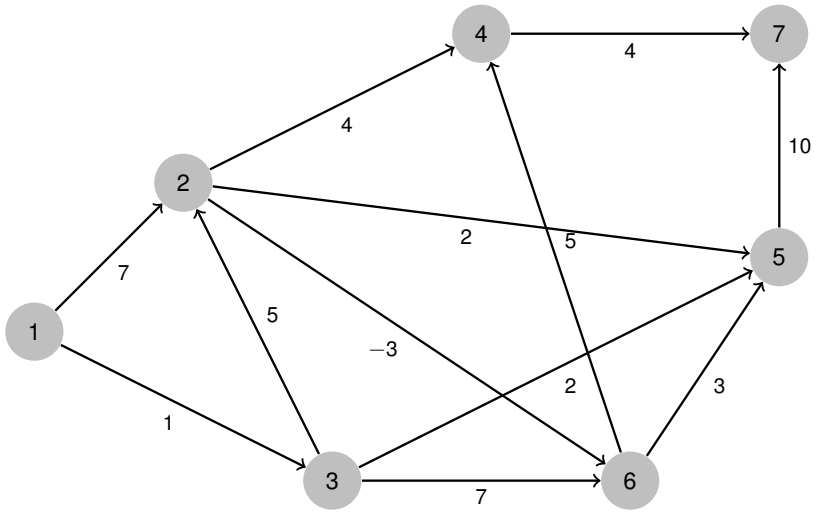
Algorithm 21 Procédure vertex-rank-shortest-paths(G,l,s).

Require: DAG $G = (V, E)$, edge weights $\{l_e : e \in E\}$, vertex s$\in V$, G is represented by vertex-rank levels
Ensure: For all vertices u reachable from s, $dist(u)$ is set to the distance from s to u

1: **for all** $u \in V$ **do**
2: $dist(u) \leftarrow \infty$
3: **end for**
4: $dist(s) \leftarrow 0$
5: **for all** vertices $v \in V \setminus \{s\}$ in rank order **do**
6: $dist(v) = \min_{u \in \Gamma^{-1}(v)} \{dist(u) + l(u, v)\}$
7: **end for**

- Complexité de l'algorithme (21) : $O(|V|)$ + le coût des boucles (5) et (6) ($\Sigma_u d^- u$) = $O(|V| + |E|)$.

Illustration pour **vertex-rank-shortest-paths(G,l,s)**



81/152

Application la fonction rang pour la détermination de l'ordre topologique/linéaire.

- La fonction rang permet de déterminer un ordre topologique/linéaire.
- Pour se faire, il suffit de placer les sommets sur une ligne dans l'ordre croissant de leurs rangs (les sommets de grands rangs placés à droite).
- Une version de l'algorithme (21) est donné par l'algorithme suivant.

Algorithm 22 Procédure dag-shortest-paths(G,l,s).

Require: DAG G=(V,E), edge weights $\{l_e : e \in E\}$, vertex s$\in V$
Ensure: For all vertices u reachable from s, $dist(u)$ is set to the distance from s to u

```
for all u ∈ V do
    dist(u) ← ∞
end for
dist(s) ← 0
linearize G
for all u ∈ V, in linearized order do
    for all edges (u, v) ∈ E do
        dist(v) = min{dist(v), dist(u) + l(u, v)}
    end for
end for
```


82/152

Application la fonction rang pour la détermination de l'ordre topologique/linéaire.

Une variante de l'algorithme (22) est la suivante :

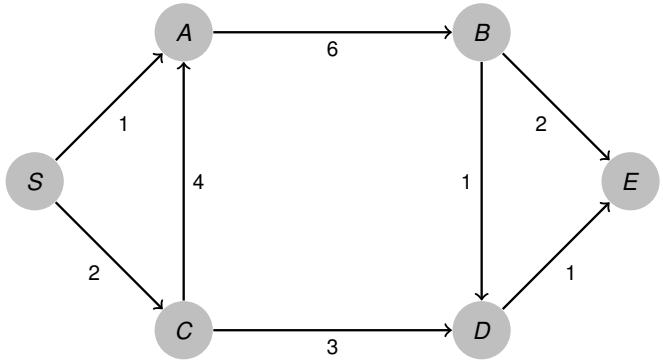
Algorithm 23 Procédure dag-shortest-paths(BIS)(G,l,s).

Require: DAG G=(V,E), edge weights $\{l_e : e \in E\}$, vertex s$\in V$
Ensure: For all vertices u reachable from s, $dist(u)$ is set to the distance from s to u

```
for all u ∈ V do
    dist(u) ← ∞
end for
dist(s) ← 0
linearize G
for all v ∈ V \ {s}, in linearized order do
    for all edges (u, v) ∈ E do
        dist(v) = min{dist(v), dist(u) + l(u, v)}
    end for
end for
```


83/152

Illustration pour **dag-shortest-paths(BIS)(G,l,s)**

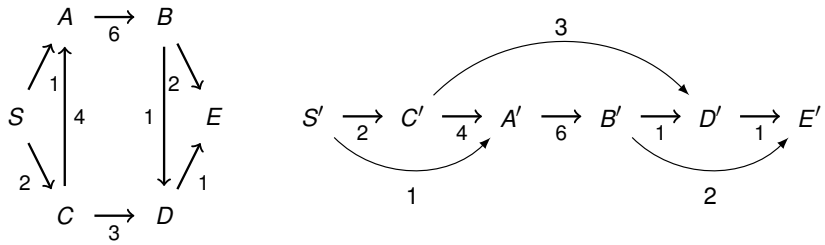


84/152

Les principes de la programmation dynamique : rappel

- Trouver le PCC est particulièrement simple dans le cas d'un graphe $G = (V, E)$ de type DAG (graphe orienté, sans circuit, et avec des poids sur les arêtes $\{l_e : e \in E\}$).
- Cela est dû à la propriété des DAG d'être linéarisés (c.-à-d. numéroter les sommets de façon que pour chaque arc $(u, v) \in E$ on a $num(u) < num(v)$).
- Exemple

FIGURE 15 – À gauche : un graphe G ; à droite : le même graphe après l'avoir trié/ordonné topologiquement/linéairement).



Les principes de la programmation dynamique : rappel

- Problème : Trouver le PCC d'un sommet $s \in V$ à tous les autres $v \in V \backslash \{s\}$.
- Observation : Le problème se simplifie si on connait les $PCC(s \rightsquigarrow u)$ pour $\forall u \in \Gamma^{-1}(v)$. Dans ce cas :

$$dist(v) = \min_{u \in \Gamma^{-1}(v)} \{dist(u) + l(u, v)\}$$

- Le calcul des valeurs *dist* dans l'ordre linéaire garantit que les valeurs nécessaires pour le calcul de $dist(v)$ sont déjà connues au moment de ce calcul.
- Cela permet de trouver les valeurs *dist* avec un seul parcours des sommets (l'algorithme 23 « dag-shortest-paths(BIS)(G,l,s) »).

Les principes de la programmation dynamique : rappel

- L'algorithme 23 illustre bien l'approche programmation dynamique qui consiste à résoudre un ensemble de sous-problèmes $\{dist(u), u \in V\}$ en commençant par les plus « petits » (les prédécesseurs) et en allant progressivement vers les plus « grands » (les successeurs).
- Cependant, le graphe DAG peut ne pas être explicitement donné, mais il est implicite.
- Les sommets représentent dans ce cas les sous-problèmes à résoudre, tandis que les arcs correspondent aux dépendances entre les problèmes ; si la solution du sous-problème B nécessite une réponse du sous-problème A, on met un arc (conceptuel) de A à B.
- A est considéré ainsi comme un sous-problème plus petit que B (un prédécesseur de B).

La plus longue sous-séquence croissante

Donné : Une séquence de nombres $S = a_1, a_2, \ldots, a_n$.

Def_1 : Une sous-séquence : chaque sous-ensemble des nombres $a_{i1}, a_{i2}, \ldots, a_{ik}$ tel que $1 \leq i1 \leq i2 \leq \ldots ik \leq n$.

Def_2 : Une sous-séquence croissante : telle que les nombres a_i croissent.

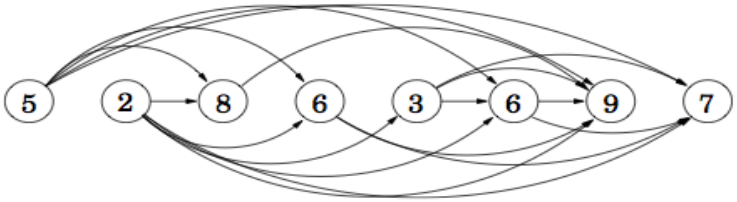
Problème : Trouver la plus longue sous-séquence croissante de S (PLSSC(S)).

Exemple : La plus longue sous-séquence croissante de $S = 5, 2, 8, 6, 3, 6, 9, 7$ est $2, 3, 6, 9$.

La plus longue sous-séquence croissante

- L'espace des solutions sera présenté par un DAG $G = (V, E)$ où à chaque nombre a_i on associera un sommet $i \in V$ et on ajoutera un arc (i, j) si $i < j$ et $a_i < a_j$.
- On établit ainsi une bijection (« *one-to-one correspondence* ») entre les chemins dans G et les sous-séquences croissantes dans S.
- L'objectif est de trouver le plus longue chemin (PLC) dans ce graphe.

Exemple : Le graphe DAG associé à la séquence $S = 5, 2, 8, 6, 3, 6, 9, 7$.



89/152

Algorithme pour la plus longue sous-séquence croissante

Algorithm 24 Algorithme *longest increasing subsequence*.

1: **for** $j = 1$ to n **do**
2: $\quad L(j) = 1 + \max\{L(i) \mid (i, j) \in E\}$
3: **end for**
4: return $\max_j L(j)$

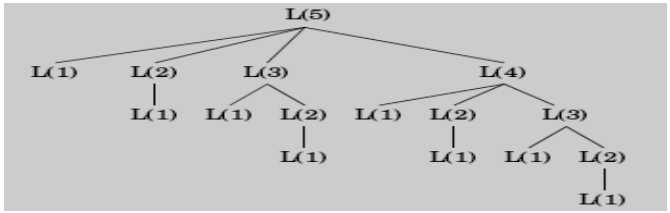
where $\max\{\} = 0$.

- L'idée de base : $L(j)$ est la longueur de la PLSSC aboutissant au sommet j (+1) (on cherche le nombre de sommets, pas des arêtes).
- $L(j)$ est la longueur maximale parmi les PLSSC des prédécesseurs du j (+1).
- On découvre ici l'approche programmation dynamique : l'ordre et les relations entre les problèmes indiquent que pour résoudre un problème donné $L(j)$, il faut que ses prédécesseurs (les sous-problèmes qui le précédent) soient résolus.
- Complexité de l'algorithme (24) : le coût de la boucle **for** de la ligne (1) ($O(|V|)$) + le coût de la ligne (2) $\Sigma_{j \in V} d^-(j) = O(|E|)$. En total : $O(|V| + |E|) = O(n^2)$.
- L'algorithme (24) ne trouve que la longueur de la PLSSC. Pour connaitre la séquence même, il faut sauvegarder les indices argmax de la ligne (2).

90/152

Attention aux appels récursifs naïfs ! ! ! !

- La ligne 2 de l'algorithme (24) suggère une alternative – pourquoi ne pas utiliser un algorithme récursif ?
- En fait, c'est une mauvaise idée puisque l'algorithme récursif s'exécute en temps exponentiel.
- Considérons par exemple le cas d'une séquence croissante. Elle contient tous les arcs $\{(i, j) \mid i < j\}$. La formule devient $L(j) = 1 + \max\{L(1), L(2), \ldots, L(j-1)\}$.
- Pour $L(5)$ on obtient alors :



- Et ce sous-problème sera résolu maintefois !. Pour $L(n)$ la taille de l'arbre sera exponentielle. Cela s'explique par les dépendances entre les problèmes $L(j)$ et $L(j-1)$; ils sont presque de la même taille.
- Pour améliorer l'efficacité il faut mémoriser les solutions des sous-problèmes résolus, et les calculer dans le bon ordre (cela évoque déjà la programmation dynamique).

91/152

La distance de Levenshtein (distance d'édition, de similarité)

- La distance de Levenshtein entre mots ou chaînes de caractères donne des indications sur le degré de ressemblance de ces chaînes.

Exemple : Deux alignements possibles des mots « SNOWY » et « SUNNY ».

S	-	N	O	W	Y		-	S	N	O	W	-	Y
S	U	N	N	-	Y		S	U	N	-	-	N	Y
coût = 3							coût = 5						

- Le symbole « _ » indique une omission (« *gap* »). On peut en utiliser tant que l'on veut.
- Le coût d'un alignement équivaut le nombre de colonnes où les caractères divergent.
- La distance de Levenshtein correspond au coût du meilleur alignement.
- Si A, B sont deux mots, la distance de Levenshtein est le nombre minimum de remplacements, ajouts et suppressions de lettres pour passer du mot A au mot B.

92/152

- Afin d'appliquer la programmation dynamique, il faut repérer les sous-problèmes.
- L'entrée consiste en deux séquences $x[1,2,\ldots,m]$ et $y[1,2,\ldots,m]$.
- Considérons la distance d'édition entre le préfixe $x[1,2,\ldots,i]$ de x et le préfixe $y[1,2,\ldots,j]$ de y. Notons ce problème $E(i,j)$.
- L'objective est de calculer $E(m,n)$
- Par exemple, le problème $E(7,5)$ compare les parties en rouge des deux séquences suivantes.

E	X	P	O	N	E	N	T	I	A	L
P	O	L	Y	N	O	M	I	A	L	

93/152

- Comment découper $E(i,j)$ en sous-problèmes ?
- Considérons la dernière colonne de l'alignement de $x[1,2,\ldots,i]$ avec $y[1,2,\ldots,j]$. Il y trois cas possibles.

- Dans le premier cas le coût est 1 + le coût d'aligner $x[1,2,\ldots,i-1]$ avec $y[1,2,\ldots,j]$ (c.-à-d. le sous-problème $E(i-1,j)$).
- Dans le deuxième cas le coût est 1 + le coût d'aligner $x[1,2,\ldots,i]$ avec $y[1,2,\ldots,j-1]$ (c.-à-d. le sous-problème $E(i,j-1)$).
- Dans le troisième cas le coût est le coût d'aligner $x[1,2,\ldots,i-1]$ avec $y[1,2,\ldots,j-1]$ (c.-à-d. le sous-problème $E(i-1,j-1)$) + 1 si $x[i] \neq y[j]$ et 0 si $x[i] = y[j]$.

94/152

- $E(i,j)$ est découpé donc en trois sous-problèmes : $E(i-1,j)$, $E(i,j-1)$, $E(i-1,j-1)$.
- Pour trouver la meilleure solution il suffit de prendre le min :

$$E(i,j) = \min\{1+E(i-1,j), 1+E(i,j-1), \text{diff(i,j)}+E(i-1,j-1)\}$$

où diff(i,j)=0, si $x[i] = y[j]$, 1 sinon.

- Pour se faire, on mémorisera les solutions des sous-problèmes $E(i,j)$ dans une table 2D. Cette table sera parcourue de façon que les sous-problèmes $E(i-1,j), E(i,j-1), E(i-1,j-1)$ sont calculés avant $E(i,j)$.
- Valeurs initiales : on pose $E(i,0) = i$ (puisque c'est la distance d'édition entre une séquence vide et les i premiers caractères de x). On pose aussi $E(0,j) = j$.
- On obtient ainsi l'algorithme suivant.

95/152

Algorithm 25 *Dynamic programming algorithm for edit distance.*

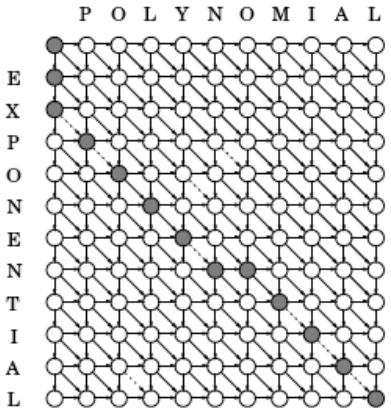
1: **for** $i = 0$ to m **do**
2: $\quad E(i,0) = i$
3: **end for**
4: **for** $j = 0$ to n **do**
5: $\quad E(0,j) = j$
6: **end for**
7: **for** $i = 0$ to m **do**
8: $\quad$ **for** $j = 0$ to n **do**
$\quad\quad E(i,j) = \min\{1+E(i-1,j), 1+E(i,j-1), \text{diff(i,j)}+E(i-1,j-1)\}$
9: $\quad$ **end for**
10: **end for**
11: return $E(m,n)$

96/152

La distance d'édition par la programmation dynamique

- Les sous-problèmes sont de la forme (i,j) et dans le graphe DAG associé il existe des dépendances (arcs) entre $(i-1,j),(i,j-1),(i-1,j-1)$ et (i,j).
- On peut fixer les longueurs des arcs de façon que les distances d'édition correspondent aux plus courts chemins.

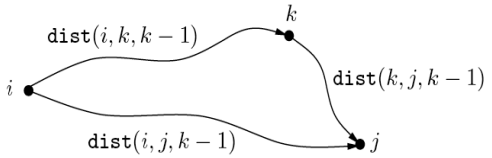
- On pose $l_e = 1$ pour tous les arcs sauf pour les arcs entre $(i-1,j-1)$ et (i,j) pour lesquels $x[i] = y[j]$. Dans ce dernier cas on pose $l_e = 0$ (ces arcs correspondent aux arcs diagonales de la figure).
- La distance d'édition équivaut la longueur du PCC entre $(0,0)$ et (m,n).



97/152

Recherche du plus courts chemin entre chaque couple de sommets

- Initialisation : $dist(i,j,0) = l(i,j)$ si $(i,j) \in E$, $dist(i,j,0) = \infty$ sinon.
- En utilisant le principe d'optimalité de Bellman, on obtient alors pour chaque nouveau sommet intermédiaire la récurrence suivante :
- $dist(i,j,k) \leftarrow \min\{dist(i,j,k-1), dist(i,k,k-1) + dist(k,j,k-1)\}$.



- On obtient à la fin la longueur du $PCC(i \rightsquigarrow j)$ pour chaque couple (i,j).

98/152

Algorithm 26 Procédure Floyd-Warshall(G,l,s).

Require: $G = (V,E)$ without negative cycles, edge weights $\{l_e : e \in E\}$
Ensure: $dist(i,j,n)$ = length of of the shortest path from i to j for all $i,j, \in V$

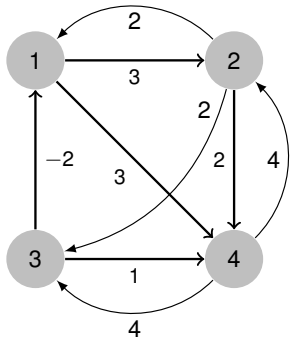
```
1: for i = 1 to n do
2:   for j = 1 to n do
3:     dist(i,j,0) ← ∞
4:   end for
5: end for
6: for all (i,j) ∈ E do
7:   dist(i,j,0) ← l(i,j)
8: end for
9: for k = 1 to n do
10:  for i = 1 to n do
11:    for j = 1 to n do
12:      dist(i,j,k) ← min{dist(i,j,k−1), dist(i,k,k−1) + dist(k,j,k−1)}
13:    end for
14:  end for
15: end for
```

Améliorations possibles : Ajoutez des testes complémentaires pour éviter le calcul si $l(i,k) = \infty$ et qui permettent de s'arrêter à la détection du premier circuit de longueur négative.

99/152

Algorithme de Floyd-Warshall : illustration

FIGURE 16 – Appliquer l'alg. de Floyd-Warshall au graphe suivant.



100/152

Arbres couvrants minimaux (ACM)

101/152

Définitions et propriétés principales

Définition : Un graphe, non-orienté, connexe et acyclique est dit arbre.

Propriétés :
- Un arbre avec n sommets possède $n-1$ arêtes.
- Si le graphe $G=(V,E)$, non-orienté et connexe, est tel que $|E|=|V|-1$, alors G est un arbre.
- Un graphe non-orienté est un arbre, si et seulement si, entre chaque couple de sommets il existe un seul chemin.

Problème : Donné un graphe $G=(V,E,w_e,e\in E)$, non-orienté, connexe. On cherche $T\subseteq E$ qui connecte tous les sommets et qui minimise le poids total $w(T)=\sum_{(u,v)\in T} w(u,v)$.
On voit facilement que T est acyclique, donc un arbre. T est appelé **Arbre Couvrant Minimal (ACM)** .

102/152

Extrait du sujet d'examen du décembre 2011

Exercice 2 : Des chemins forestiers
Les Eaux et Forêts ont décidé d'abattre 5 bosquets situés dans une forêt. Les distances mutuelles entre les bosquets sont données (en km) par la matrice suivante.

	a	b	c	d	e
a		10	1	2	5
b	10		6	4	3
c	1	6		8	9
d	2	4	8		7
e	5	3	9	7	

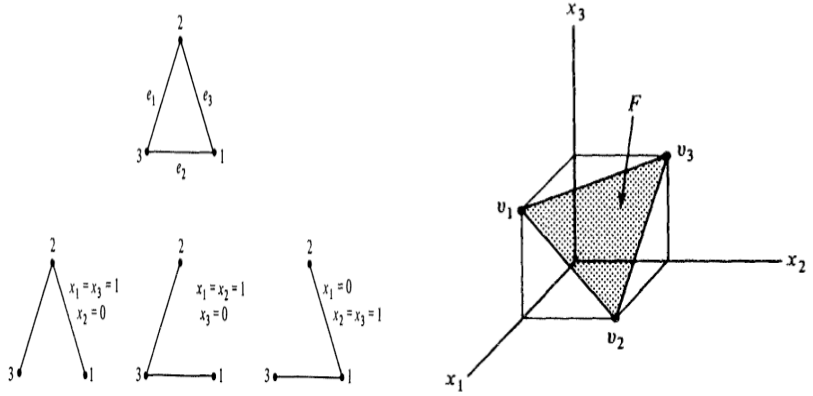
Pour mener à bien cette exploitation il est nécessaire de tracer un réseau de chemins de coût de construction minimum qui permette de circuler entre tous les bosquets. Les coûts de construction de ces chemins sont proportionnels à leurs longueurs.

- Enumérer les algorithmes vus en cours pouvant résoudre ce problème. Donner la complexité de ces algorithmes pour un graphe quelconque $G=(V,E)$.
- etc. (cf. le sujet d'examen du décembre 2011)

103/152

L'ACM vu comme un PL

On cherche l'arbre couvrant minimal (ACM) du graphe $G=(V,E)$ tel que $|V|=|E|=3$. Il y 3 arbres candidats d'être l'ACM pour G (ci-dessous à gauche). Notons $x_j=1$ ssi $e_j=1$. Les trois arbres peuvent alors etre vus comme des points en $3D$ et ils coincîdent avec les sommets v_1, v_2 et v_3 du polytope F (ci-dessous à droite) défini par les contraintes $x_1+x_2+x_3=2, 0\le x_i\le 1$. Ainsi, un problème purement combinatoire peut être résolu comme un PL.



104/152

Définitions et propriétés principales

Problème : Donné un graphe $G = (V, E, w_e, e \in E)$, non-orienté, connexe. On cherche $T \subseteq E$ qui connecte tous les sommets et qui minimise le poids total $w(T) = \sum_{(u,v)\in T} w(u,v)$. On voit facilement que T est acyclique, donc un arbre.
T est appelé **Arbre Couvrant Minimal (ACM)** .

Définitions :
- Soit T un ACM, et soit $X \subseteq T$. Une arête (u,v), telle que $X \cup \{(u,v)\} \subseteq T$, est appelée *arête sure* pour X.
- Une *coupure* $(S, V\backslash S)$ du graphe $G(V,E)$ est une partition des sommets V.
- Une arête *traverse la coupure* $(S, V\backslash S)$ si l'une de ses extrémités appartient à S et la seconde appartient à $V\backslash S$.
- Un ensemble d'arêtes X traverse la coupure $(S, V\backslash S)$ si $\exists$ au moins une arête $e \in X$ qui traverse cette coupure.
- Une arête est dite *minimale* pour la traversée de la coupure si son poids est minimal parmi toutes les arêtes qui traversent la coupure.

Théorème : *Soit T un ACM, et soit $X \subset T$. Soit $S \subset V$ tq X ne traverse pas la coupure $(S, V\backslash S)$ (on dira que la coupure respecte X), et soit (u,v) une arête minimale traversant $(S, V\backslash S)$. Alors (u,v) est une arête sure pour X.*

Algorithme générique pour la construction d'un ACM

Algorithm 27 function ACM_generique (G(V,E),w)

Require: Un graphe $G = (V, E)$ avec des poids $w_e, e \in E$.
Ensure: Un ACM défini par les arêtes de l'ensemble X.
  $X \leftarrow \emptyset$
  while X ne forme pas un arbre couvrant **do**
    trouver pour X une arête sure (u,v)
    $X \leftarrow X \cup \{(u,v)\}$
  end while
return X

Algorithme de Prim pour la construction d'un ACM (version tbl)

L'algorithme de Prim a pour propriété que les arêtes de l'ensemble X constituent toujours un arbre unique. Cet algorithme ressemble beaucoup à l'alg. de Dijkstra.

Algorithm 28 function ACM_Prim (G(V,E),w,r)

Require: Un graphe $G = (V, E)$ avec des poids $w_e, e \in E$. r est le sommet initial.
Ensure: Un ACM défini par les pointeurs π tq $\pi(v)$ désigne le père de v dans l'arbre.
1: **Initialisation :** $\forall u \in V$: $clef(u) \leftarrow \infty$ and $\pi(u) \leftarrow nil$; $S \leftarrow \emptyset$; $clef(r) \leftarrow 0$;
2: $F \leftarrow$ **maketbl(V)** {crée un tbl F avec $clef(v), v \in (V)$ comme clés}
3: **while** F is not empty **do**
4: $u \leftarrow$ **ExtractMin(F)** {l'élément u sera considéré hors F }
5: $S \leftarrow S \cup \{u\}$
6: **for all** edges $(u,v) \in E$ **do**
7: **if** $v \in F$ **and** $clef(v) > w(u,v)$ **then**
8: $clef(v) = w(u,v)$
9: $\pi(v) \leftarrow u$
10: **end if**
11: **end for**
12: **end while**
13: **return** π

- Complexité : Les lignes (3-4) en $O(|V|^2)$. La boucle for all (ligne 6) en $O(|E|)$. En total $O(|V|^2 + |E|) = O(|V|^2)$.

Illustration pour l'algorithme **ACM_Prim (G(V,E),w,r)**

Set S	A	B	C	D	E	F
$\{\}$	0/nil	∞/nil	∞/nil	∞/nil	∞/nil	∞/nil
A		$5/A$	$6/A$	$4/A$	∞/nil	∞/nil
A, D		$2/D$	$2/D$		∞/nil	$4/D$
A, D, B			$1/B$		∞/nil	$4/D$
A, D, B, C					$5/C$	$3/C$
A, D, B, C, F					$4/F$	

FIGURE 17 – Un graphe et son arbre couvrant minimal trouvé par l'algorithme de Prim

Le tas binaire "Binary Heap" : une file de priorité importante (Rappel)

Chaque élément d'un tas binaire est caractérisé par le couple (nom, valeur numérique) $(v, clef(v))$.

Propriétés d'un tas de k éléments :

- la racine contient le plus petit élément. L'opération d'accès eu minimum" se fait en $\Theta(1)$
- l'opération "insertion" se fait en $\Theta(\log_2 k)$
- l'opération "suppression du minimum " se fait en $\Theta(\log_2 k)$

109/152

Tas binaire (rappel)

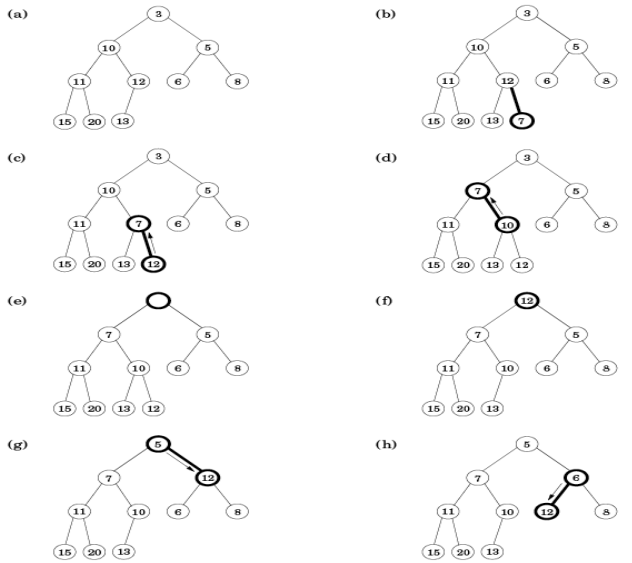


FIGURE 18 – (a) : un tas binaire avec 10 éléments. Seulement les clés sont indiquées. (b)-(d) : Illustration de l'opération "insertion". (e)-(g) : Illustration de l'opération "extraction et suppression du minimum".

110/152

Algorithme de Prim pour la construction d'un ACM (version tas binaire)

Algorithm 29 function ACM_Prim (G(V,E),w,r)

Require: Un graphe $G = (V, E)$ avec des poids $w_e, e \in E$. r est le sommet initial.
Ensure: Un ACM défini par les pointeurs π tq $\pi(v)$ désigne le père de v dans l'arbre.

1: **Initialisation :** $\forall u \in V$: $clef(u) \leftarrow \infty$ and $\pi(u) \leftarrow nil$; $clef(r) \leftarrow 0$;
2: $F \leftarrow$ **makequeue(V)** {crée une file de priorité F (binary heap) avec $clef(v), v \in (V)$ comme clés}
3: **while** F is not empty **do**
4: $u \leftarrow$ **ExtractMin(F)**
5: **for all** edges $(u, v) \in E$ **do**
6: **if** $clef(v) > w(u, v)$ **then**
7: $clef(v) = w(u, v)$
8: $\pi(v) \leftarrow u$
9: **end if**
10: **end for**
11: **end while**
12: **return** π

Complexité :

- Les lignes (2-3) en $O(|V| \log |V|)$.
- La boucle for all (ligne 5) en $O(|E|)$.
- La mise à jour des clefs (lignes 6-9) en $O(\log |V|)$.
- En total : $O(|V| \log |V| + |E| \log |V|) = O(|E| \log |V|)$.

111/152

Algorithme de Kruskal pour la construction d'un ACM

Algorithm 30 function ACM_Kruskal (G(V,E),w)

Require: Un graphe $G = (V, E)$ avec des poids $w_e, e \in E$.
Ensure: Un ACM défini par les arêtes de l'ensemble X.

for all $u \in V$ **do**
makeset(u)
end for
$X \leftarrow \emptyset$
trier les arêtes E par leurs poids
for all $(u, v) \in E$, dans l'ordre croissant des poids **do**
if find(u) $\neq$ find(v) **then**
$X \leftarrow X \cup \{(u, v)\}$
union(u,v)
end if
end for
return X

- Complexité : On fait $|V|$ makeset, $2|E|$ find et $|V| - 1$ unions.

112/152

Illustration pour l'algorithme **ACM_Kruskal (G(V,E),w)**

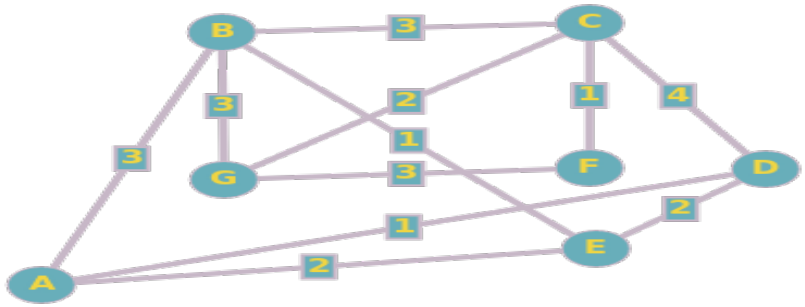


FIGURE 19 – Appliquer l'algorithme de Kruskal pour trouver l'arbre couvrant minimal

113/152

Opérations et structures de données pour les ensembles disjoints

La structure de données maintient à jour une collection $\{S_1, S_2, \ldots, S_k\}$ d'ensembles dynamiques disjoints. Chaque ensemble est un arbre orienté, dont la racine est le représentant de cet ensemble. Chaque noeud de l'arbre est muni d'un pointeur, ($\boldsymbol{\pi}$), vers son père, ainsi que d'un rang, (**rank**), qui équivaut la taille de sous-arbre enraciné à ce noeud. Cette structure permet de connaitre rapidement à quel ensemble appartient un élément donné, et de pouvoir réunir deux ensembles.

114/152

Opérations et structures de données pour les ensembles disjoints (suite)

Algorithm 31 procedure makeset (x)

Ensure: crée un nouvel ensemble dont le seul élément (et donc le représentant) est x.
$\pi(x) \leftarrow x$
$rank(x) \leftarrow 0$

Algorithm 32 function find(x)

Ensure: retourne un pointeur vers le représentant de l'ensemble (unique) contenant x.
while x$\neq \pi(x)$ **do**
$x \leftarrow \pi(x)$
end while
return x

Algorithm 33 function find_bis(x)

Ensure: effectue une compression du chemin lors de l'opération find
if x$\neq \pi(x)$ **then**
$\pi(x) \leftarrow$find($\pi(x)$)
end if
return $\pi(x)$

115/152

Opérations et structures de données pour les ensembles disjoints (suite)

Algorithm 34 procedure union (x,y)

Require: Deux ensembles dynamiques S_x et S_y représentés par leurs racines x et y. S_x et S_y sont supposés disjoints avant l'opération.
Ensure: Réunit les ensembles dynamiques S_x et S_y dans $S_x \cup S_y$. On fait pointer la racine du moindre rang sur celle de rang supérieur au moment de l'union. Comme les ensembles de la collection sont obligatoirement disjoints, on supprime les ensembles S_x et S_y.
$r_x \leftarrow$ find(x)
$r_y \leftarrow$ find(y)
if $r_x = r_y$ **then**
return
end if
if rank(r_x) $>$ rank(r_y) **then**
$\pi(r_y) \leftarrow r_x$
else
$\pi(r_x) \leftarrow r_y$
if rank(r_x) $=$ rank(r_y) **then**
rank(r_y) $\leftarrow$ rank(r_y) $+ 1$
end if
end if

116/152

Illustration pour les opérations sur les ensembles disjoints

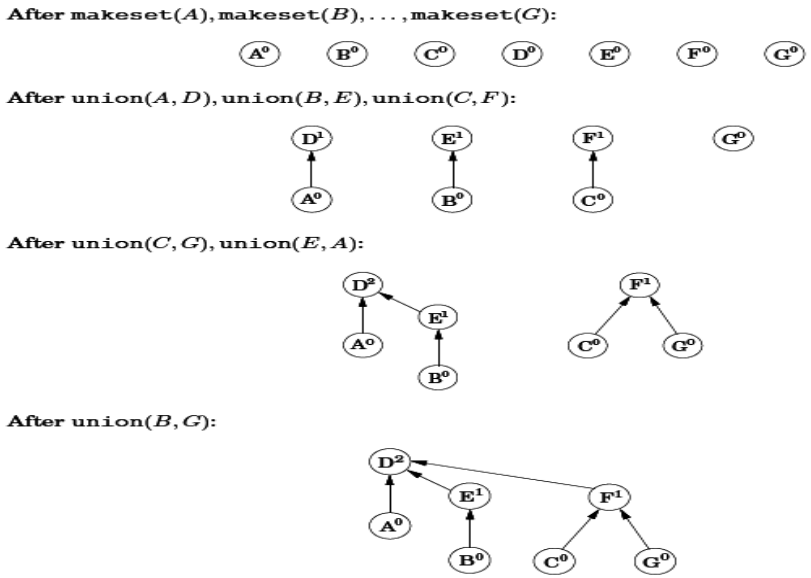


FIGURE 20 – Résultat des opérations **makeset** et **union** sur des ensembles disjoints

117/152

Analyse de complexité des opérations sur les ensembles disjoints

Propriétés des structures de données pour les ensembles disjoints :

- $\forall x \neq$ la racine, rank(x) $<$ rank(π(x))
- Chaque racine de rang k a au moins 2^k noeuds dans son arbre.
- Chaque ensemble de n éléments possède au plus $\frac{n}{2^k}$ noeuds de rang k.
- La hauteur de l'arbre représentant un ensemble de n éléments ne dépasse pas $\log n$.

Corolaire 1 La complexité des opérations find et union est en $O(\log|V|)$.
Corolaire 2 La complexité de l'algorithme de Kruskal est en $O(|E|(\log|E|+\log|V|)) = O(|E|\log|V|)$ puisque $\log|E| \approx \log|V|$
(la même que la complexité de l'algorithme de Prim).

118/152

Le problème du flot maximum dans les réseaux de transport : Algorithme de Ford-Fulkerson

119/152

Les réseaux de transport : définitions

- Soit $R = (X, U, C)$ un graphe connexe orienté (réseau). A $\forall$ arc u on affecte une valeur (*capacité*) c_u qui est une borne supérieure du flux sur l'arc.
- Soit deux sommets particuliers $s \in X$ (source) et $t \in X$ (puits). Considérons le graphe $G^0 = (X, U^0)$ où $U^0 = U \cup \{t, s\}$. L'arc (t, s) est appelé l'*arc de retour* du flot (numéroté 0). Notons $M = |U|$.
- On dit que $[\phi_1, \phi_2, \ldots \phi_M]^T$ est un *flot de s à t* ssi la loi de conservation du flot est vraie en tout $\forall u \in X \backslash \{s, t\}$, i.e.

$$\sum_{v \in \Gamma^+(u)} \phi_{(u,v)} = \sum_{v \in \Gamma^-(u)} \phi_{(v,u)} \tag{9}$$

- La valeur du flot est notée par ϕ_0. Elle est définie par

$$\sum_{v \in \Gamma^+(s)} \phi_{(s,v)} = \sum_{u \in \Gamma^-(t)} \phi_{(u,t)} = \phi_0 \text{ (valeur du flot)} \tag{10}$$

- But : Trouver dans G^0 un *flot compatible* $\phi' = [\phi_0, \phi_1, \phi_2, \ldots \phi_M]$ (c.-à-d. $0 \leq \phi_u \leq c_u \; \forall u =\in U$) et tel que ϕ_0 soit *maximale*.

120/152

Exemple de réseau

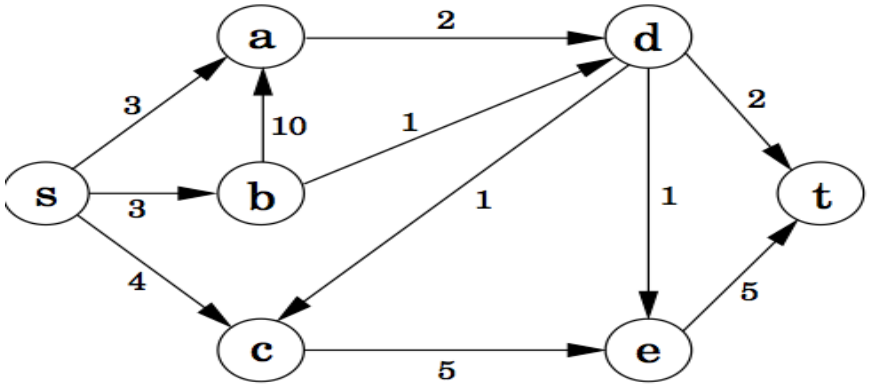


FIGURE 21 – Trouver le flot maximum entre les sommets s et t.

121/152

Définition d'une coupe et de sa capacité

- *Une coupe séparant s et t* (notée $s-t$) est une partition $(A, \bar{A})$ de X, telle que $s \in A$ et $t \in \bar{A} = X \setminus A$.
- *La capacité* $C(A, \bar{A})$ *de la coupe* $s-t$ est définie par

$$C(A, \bar{A}) = \sum_{e \in \omega^+(A)} c_e$$

où $\omega^+(A)$ représente le sous-ensemble d'arcs qui ont leur extrémité initiale dans A et leur extrémité terminale dans $\bar{A}$.

- **Lemme :** La valeur maximale d'un flot de s à t compatible avec c_u n'excède jamais la capacité d'une coupe séparant s et t.
- **Théorème :** La valeur maximale d'un flot de s à t est égale à la capacité d'une coupe de capacité minimale séparant s et t.
  Pour démontrer le théorème précédent on aura besoin de la notion *graphe d'écart* (aussi noté *graphe résiduel*).

122/152

Graphe d'écart (résiduel)

- Soit $\phi = [\phi_1, \phi_2, \ldots \phi_M]^T$ un flot entre s et t compatible avec c_u.
- Le graphe d'écart associé à ϕ est le graphe $\bar{G}(\phi) = [X, \bar{U}(\phi)]$ où $\bar{U}(\phi)$ est constitué de façon suivante :
  pour $\forall u = (i,j) \in U$ on associe au plus deux arcs de $\bar{G}(\phi)$
  - $u^+ = (i,j)$ si $\phi_u < c_u$ avec capacité (résiduelle) $c_u - \phi_u > 0$
  - $u^- = (j,i)$ si $\phi_u > 0$ avec capacité (résiduelle) $\phi_u > 0$

  Remarque : si $\phi_u = c_u$, on lui associe seulement u^-, si $\phi_u = 0$, on lui associe seulement u^+.
- Etant donné un flot ϕ, un *chemin améliorant* $(s \rightsquigarrow t)$ est un chemin simple de s à t dans le réseau résiduel $\bar{G}(\phi)$.
- Soit ε le minimum des capacités résiduelles d'un chemin améliorant π.
- Alors on peut obtenir un nouveau flot de valeur $\phi_0' = \phi_0 + \varepsilon$.

123/152

Algorithme de Ford et Fulkerson (1956)

- $k = 0$. Partir d'un flot initial ϕ^0 compatible avec les capacités.
- Soit ϕ^k le flot courant.
  Tant qu'il existe un chemin améliorant $(s \rightsquigarrow t)$ dans $\bar{G}(\phi^k)$ faire
  soit π^k un chemin améliorant $(s \rightsquigarrow t)$ dans $\bar{G}(\phi^k)$.
  Soit ε^k le minimum des capacités résiduelles du π^k.
  Définir le flot ϕ^{k+1} par

$$\phi_u^{k+1} = \phi_u^k + \varepsilon^k \text{ si } u^+ \in \pi^k \quad (11)$$

$$\phi_u^{k+1} = \phi_u^k - \varepsilon^k \text{ si } u^- \in \pi^k \quad (12)$$

$$\phi_0^{k+1} = \phi_0^k + \varepsilon^k \quad (13)$$

  $k \leftarrow k+1$
- fin tant que : le flot ϕ^k est maximum

124/152

Le relation entre le flot maximal et la coupe minimale

Théorème : Soit ϕ un flot compatible dans le réseau G. Les conditions suivantes sont équivalentes :

1. ϕ est un flot maximal dans G.
2. Le réseau résiduel $\bar{G}(\phi)$ ne contient aucun chemin améliorant.
3. La valeur du flot ϕ est égale à la capacité d'une coupe de capacité minimale séparant s et t.

Démonstration :

- $(1) \Rightarrow (2)$: Evident
- $(2) \Rightarrow (3)$: On définit $L = \{v \in X : \exists$ un chemin de s à t dans $\bar{G}(\phi)\}$.
  Soit $R = X \setminus L$. La partition (L, R) est une coupe : de façon triviale $s \in L$ et $t \in R$.
  Par hypothèse, pour chaque arc (u, v) tq $u \in L$ et $v \in R$ on a $\phi(u, v) = c(u, v)$.
  D'autre part, pour chaque arc (v, u) tq $v \in R$ et $u \in L$ on a $\phi(v, u) = 0$.
  Donc, $size(\phi) = C(L, R)$.
- $(3) \Rightarrow (1)$: Conséquence du lemme.

Illustration : Construction d'autoroutes

Avant d'établir un projet de construction d'autoroutes on désire étudier la capacité du réseau routier, représenté par le graphe ci-dessous, reliant la ville S à la ville T.
Pour cela, on a évalué le nombre maximal de véhicules que chaque route peut écouler par heure, compte tenu des ralentissements aux traversées des villes et villages, des arrêtes aux feux etc. ...Ces évaluations sont indiquées entre crochets, en centaines de véhicules par heure sur les arcs du graphe. Les temps de parcours entre villes sont tels que les automobilistes n'emprunteront que les chemins représentés par le graphe. On cherche le débit horaire total maximal de véhicules susceptible de s'écouler entre les villes S et T.

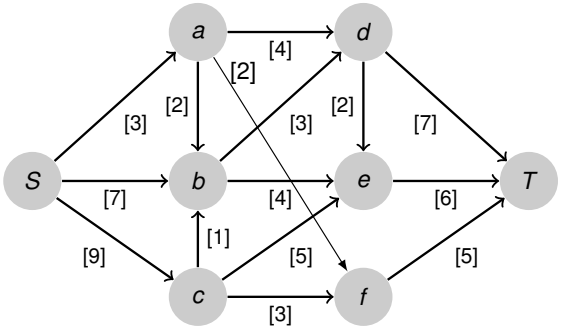


FIGURE 22 – Capacité d'un réseau routier.

Analyse de complexité : le cas où la valeur de chaque capacité égale à 1

- La recherche d'un chemin améliorant dans le graphe résiduel peut se faire en $O(|X| + |U|)$ (avec un des parcours vus en cours).
- Supposons que chaque sommet et est incident avec au moins un arc. Alors $M = |U|$, $M \geq \frac{|X|}{2}$ et $O(|X| + |U|) = O(M)$
- La mise à jour du flot nécessite $O(M)$ soustractions/additions.
- Considérons le cas où la valeur de chaque capacité vaut 1.
- Le temps total est majoré par $O(\min(d^+(s), d^-(t))M)$.

Chemins, flots et programmes linéaires dans les graphes

Programmes linéaires (PL) versus PL en Nombres Entiers (PLNE)

Exemple d'un PL : À l'approche des fêtes de Pâques, un artisan chocolatier décide de confectionner des oeufs en chocolat. En allant inspecter ses réserves, il constate qu'il lui reste 18 kg de cacao, 8 kg de noisettes et 14 kg de lait. Il a deux spécialités : l'oeuf Extra et l'oeuf Sublime. Un oeuf Extra nécessite 1 kg de cacao, 1 kg de noisettes et 2 kg de lait. Un oeuf Sublime nécessite 3 kg de cacao, 1 kg de noisettes et 1 kg de lait. Il fera un profit de 20 € en vendant un oeuf Extra, et de 30 € en vendant un oeuf Sublime. Combien d'oeufs Extra et Sublime doit-il fabriquer pour faire le plus grand bénéfice possible ?
un PL avec la solution : 3 oeufs Extra, 5 oeufs Sublime, benefice 210 €.

PLNE : si 18,5 kg de cacao à la place de 18 kg on obtien la solution 2,75 Extra et 5,25 Sublime avec une benefice=212,5 € . Ce qui n'est pas admissible !. Pour rendre le problème admissible on ajoute la conrainte : Extra, Sublime : **entier**≥ 0.

129/152

Programmes linéaires (PL)

L'intérêt des programmes linéaires (PL) réside dans ce qui suit :

- Ils modélisent convenalement un grand nombre de situations réelles.
- L'existance des solveurs efficaces pour la résolution (**IBM ILOG CPLEX, Gurobi, COIN, MINOS** etc.).
- L'existance des langages de modélisation comme **AMPL** (A Mathematical Programming Language) et **Pyomo** (Python-based, open-source optimization modeling language), **JuMP** (Julia for Mathematical Optimization).

Chaque programme linéaire peut être décrit sous la forme :

$$\textbf{\textit{Forme canonique}}\ \text{d'un PL :}\quad \begin{array}{lrcl} \text{Max} & c_1x_1 + \cdots + c_nx_n & & \\ \text{s. c.} & a_{11}x_1 + \cdots + a_{1n}x_n & \leq & b_1 \\ & \vdots \qquad\qquad\ \vdots & & \vdots \\ & a_{m1}x_1 + \cdots + a_{mn}x_n & \leq & b_m \\ & x_1 \geq 0,\ x_2 \geq 0, \ldots,\ x_n \geq 0 & & \end{array}$$

Un PL requiert toujours :

- une fonction objectif (**linéaire**)
- $m+n$ contraintes (**linéaires**) du problème.
- n variables $x_i, i = 1, \ldots, n$

130/152

Un PL sous forme matricielle

$$\begin{array}{lll} \text{Maximiser} & \sum_{j=1}^{n} c_j x_j & \\ \text{s. c.} & \sum_{j=1}^{n} a_{ij} x_j \leq b_j, & i = 1, \ldots, m \\ & x_j \geq 0 & j = 1, \ldots, n \end{array}$$

ou

$$\max\{cx \mid Ax \leq b,\ x \geq 0\}$$

où

$$A = \begin{pmatrix} a_{11} & \ldots & a_{1n} \\ \vdots & & \vdots \\ a_{m1} & \ldots & a_{mn} \end{pmatrix},\ x = \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix},\ b = \begin{pmatrix} b_1 \\ \vdots \\ b_m \end{pmatrix},$$

$$c = (c_1 \ldots c_n)$$

c désigne le vecteur **objectif**,
b – le vecteur **membre droit**,
et $A^{m \times n}$ – la **matrice des contraintes**.

131/152

Historique

La programmation linéaire est dans les fondements de la recherche opérationnelle (RO) ou aide à la décision : propose des modèles conceptuels pour analyser des situations complexes et permet aux décideurs de faire les choix les plus efficaces.

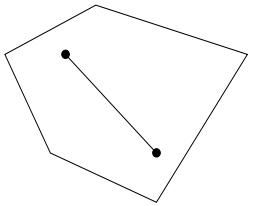
1940	P. Blackett prix Nobel de physique (1948)	dirige 1^{re} équipe de RO : implantation optimale de radars de surveillance
1939-45	L. Kantorovich	programmation linéaire
1947	G. Dantzig (le fondateur)	algorithme du simplexe " one of the top 10 algorithms of the century", CSE, 2 :1, 2000
2005	décès de Dantzig	

Aujourd'hui : développement considérable grâce aux solveurs très performants (Gurobi, CPLEX) et langages de modélisation AMPL, Jiulia, Pyomo.
En France : Société Française de Recherche Opérationnelle et d'Aide à la Décision (ROADEF).

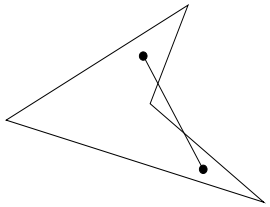
132/152

Espace des solutions admissibles

Soit un PL : $\max\{cx \mid Ax \leq b,\ x \geq 0\}$
L'espace des solutions *admissibles/réalisables/faisables* défini par l'intersection d'un nombre fini de contraintes linéaires est un polyèdre convexe $P = \{x \mid Ax \leq b,\ x \geq 0\} \subseteq \mathbb{R}^n$.
(P est convexe si $x \in P$, $y \in P$ et $0 \leq \lambda \leq 1$ alors $\lambda x + (1-\lambda)y \in P$).



convexe



non convexe

Un vecteur $x \in S \subseteq \mathbb{R}^n$ est un point *extrême*(*sommet de S*) si on ne peut pas l'exprimer comme une combinaison convexe de deux autres points de S ($\nexists (y \in S, z \in S, \alpha, 0 < \alpha < 1)$ tq $x = \alpha y + (1-\alpha)z$).

Résultat principal : L'optimum, s'il existe, est atteint en au moins un sommet du polyèdre P.

Points extrêmes du $P \subset \mathbb{R}^n$

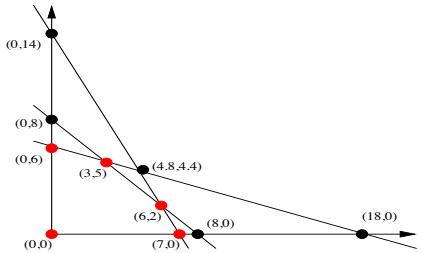
Il y a un nombre fini de façons de choisir n parmi les $m+n$ contraintes.

$$\binom{n+m}{n} = \frac{(n+m)!}{m!n!}$$

Ceci donne le nombre des points extrêmes de P qui est une borne supérieure du nombre de solutions de réalisables.
Exemple du problème du chocolatier :

$$\max\ 20x_1 + 30x_2$$

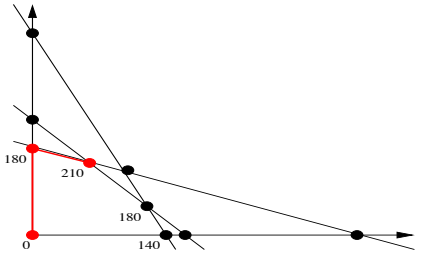
$$\begin{array}{rcrcr} x_1 & + & 3x_2 & \leq & 18 \\ x_1 & + & x_2 & \leq & 8 \\ 2x_1 & + & x_2 & \leq & 14 \\ x_1 & & & \geq & 0 \\ & & x_2 & \geq & 0 \end{array}$$



L'algorithme du simplexe

Approche naïve : On est sûr de trouver la solution optimale dans un sommet. Donc il suffit de parcourir tous les sommets et de prendre le meilleur.
Idée : Parcourir les sommets de P de façon plus intelligente.

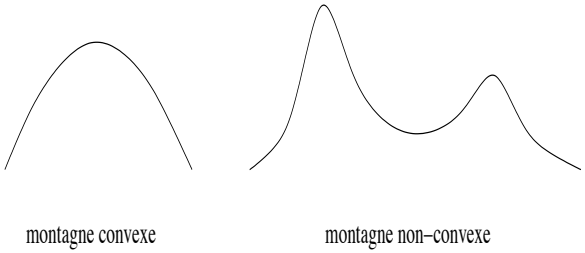
Passer itérativement d'un sommet à un sommet adjacent de façon à augmenter la valeur de la fonction à optimiser jusqu'à trouver un sommet où le maximum est atteint.



L'algorithme du simplexe

- *Initialisation* : Choisir un sommet $x^0 \in P$, $t = 1$
- *Itération* t : Soit $y^1, \ldots, y^k$ tous les sommets voisins de x^t (les sommets reliés avec x^t par une arête).
  - Si $cx^t > cy^s$, $s = 1, \ldots, k$, stop. La solution optimale est x^t.
  - Sinon, choisir un voisin y^s, tel que $cy^s \geq cx^t$. Poser $x^{t+1} = y^s$ et passer à l'itération $t+1$.

Grâce à la convexité du polyèdre des contraintes et à la linéarité de la fonction objectif, il n'y a pas de *maxima locaux*.



137/152

On se déplace d'un sommet à un autre en augmentant la valeur de la fonction z. Le nombre de sommets et fini (au plus $\binom{m+n}{n}$).

- Si la fonction objectif est bornée, on va finir par trouver le maximum
- Si elle ne est pas bornée, on va le détecter.

138/152

La garniture pour la pizza L3_INFO_MIAGE'18 est obtenue on mélangeant trois ingrédients : gruyère, jambon et champignons. La garniture ainsi conditionnée devra comporter au moins 20% de protéines et 2% de graisses, pour se conformer aux exigences de la clientèle. On a indiqué ci-dessous les pourcentages de protéines et de graisses contenus, respectivement, dans les champignons, gruyère, jambon, ainsi que le coût en € par kilogramme de chacun des ingrédients [1] :

ingrédient	1 champignons	2 gruyère	3 jambon	pourcentage requis
pourcentage de protéines	10%	50%	40%	20%
pourcentage de graisses	2%	2%	10%	2%
coût par kilogramme	25	40	40	

On notera $x_j, j = 1,2,3$ la fraction de kilogramme (ou le pourcentage) de l'ingrédient j contenu dans un kilogramme de garniture. On cherche à déterminer la composition, à coût minimal, de la garniture pour la pizza L3_INFO_MIAGE'18. Formuler le programme linéaire correspondant. Réduire la dimension du problème en utilisant la relation entre les trois fractions de kilogramme. Résoudre géométriquement.

1. les chiffres indiqués sont fictifs

139/152

Résoudre les programmes linéaires ci-dessous par l'approche graphique.

Exercice 1 :

$$\begin{array}{lrcrcl} \text{Maximiser } z = & 5x_1 & + & 8x_2 & & \\ \text{s. c.} & x_1 & + & x_2 & \leq & 2 \\ & x_1 & - & 2x_2 & \leq & 0 \\ & -x_1 & + & 4x_2 & \leq & 1 \\ & & & x_1, x_2 & \geq & 0 \end{array}$$

Exercice 2 :

$$\begin{array}{lrcrcl} \text{Maximiser } z = & 2x_1 & + & x_2 & & \\ \text{s. c.} & 4x_1 & + & 2x_2 & \leq & 8 \\ & & & x_2 & \leq & 2 \\ & & & x_1, x_2 & \geq & 0 \end{array}$$

Exercice 3 :

$$\begin{array}{lrcrcl} \text{Maximiser } z = & x_1 & + & 2x_2 & & \\ \text{s. c.} & -x_1 & + & x_2 & \leq & 2 \\ & & & x_2 & \leq & 3 \\ & & & x_1, x_2 & \geq & 0 \end{array}$$

140/152

Le problème du chemin min/max dans les graphes vu comme un PL

Soit le graphe orienté sans circuit $G = (V, E)$ avec des poids w_{ij} (de signe quelconque) sur chaque arc $(i,j) \in E$ et soit $m = |E|$ et $n = |V|$. Il s'agit de trouver le chemin le plus court/long de s à t.
Tout chemin de s à t peut être représenté par un vecteur $\mathbf{x}$ avec $x_e = 1$ si l'arc e est sur le chemin ou $x_e = 0$ sinon, et sous les conditions suivantes :

- un chemin (flot de valeur 1) sort du sommet s et entre dans le sommet t :

$$\sum_{(s,v)\in E} x_{sv} = 1 \text{ et } \sum_{(u,t)\in E} x_{ut} = 1 \tag{14}$$

- conservation du flot pour chaque sommet intermédiaire v :

$$\sum_{(u,v)\in E} x_{uv} = \sum_{(v,u)\in E} x_{vu} \tag{15}$$

- l'objectif est de maximiser/minimiser la longueur totale :

$$\max(\min) z = \sum_{(u,v)\in E} x_{uv} w_{uv} \tag{16}$$

Les équations (15), (14) et (16) représentent un programme linéaire (PL).

Barrage routier par les flots

Le réseau routier d'une île est donné sur la figure ci-dessous. On cherche à bloquer toute possibilité d'aller de A à B. Pour réduire la gêne occasionnée, on envisage bloquer seulement les routes, pas les agglomérations.

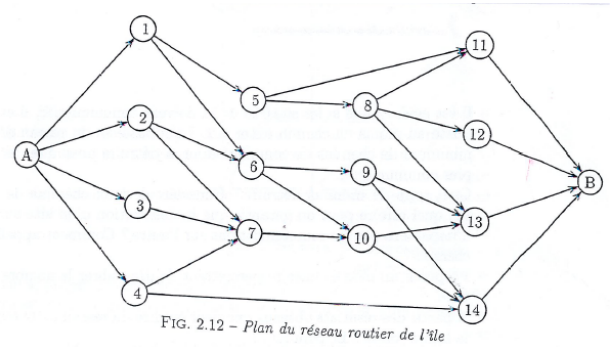


FIG. 2.12 – *Plan du réseau routier de l'île*

- Proposer une méthode générale pour déterminer où placer les barrages routiers.
- Déterminer le nombre minimal de barrages nécessaires et leur emplacement.
- Ecrivez un programme linéaire pour résoudre ce problème.

La forme matricielle du problème du flot maximum (MF)

- Soit a_i la ligne i de la matrice A. La loi de conservation pour le sommet i (sauf s, t) :

$$a_i \phi = 0 \tag{17}$$

- La forme matricielle du problème MF :

notons $d \in R^n$ le vecteur

$$d_i = \begin{cases} -1 & i = s \\ +1 & i = t \\ 0 & \text{otherwise} \end{cases}$$

$$\begin{aligned} \max \phi_0 & & & (18) \\ d\phi_0 + A\phi &= 0 & & (19) \\ \phi &\leq c & & (20) \\ \phi &\geq 0 & & (21) \\ \phi_0 &\geq 0 & & (22) \end{aligned}$$

Le modèle linéaire du problème de la coupe minimale (MC)

Associons les variables $\lambda_i, i \in V$ avec les sommets et les variables $\gamma_{i,j}, (i,j) \in E$ avec les arcs.

$$\begin{aligned} \min \sum_{(i,j)\in E} \gamma_{i,j} c_{i,j} & & & (23) \\ \lambda_i - \lambda_j + \gamma_{i,j} &\geq 0 \ \forall (i,j) \in E & & (24) \\ -\lambda_s + \lambda_t &\geq 1 & & (25) \\ \gamma_{i,j} &\geq 0 \ \forall (i,j) \in E & & (26) \end{aligned}$$

Definition

Une *coupe* $s-t$ est une partition $(A, \bar{A})$ des sommets V, tq $s \in A$ et $t \in \bar{A}$. La *capacité* de la coupe $s-t$ est $C(A,\bar{A}) = \sum_{e\in\omega^+(A)} c_e$ où $\omega^+(A)$ représente le sous-ensemble d'arcs qui ont leur extrémité initiale dans A et leur extrémité terminale dans $\bar{A}$.

Theorem

Les affectations suivantes définissent une solution admisible pour le problème (D) ayant un coût $C(A,\bar{A})$:

$$\gamma_{i,j} = \begin{cases} 1 & (i,j) \ s.t. \ i \in A, j \in \bar{A} \\ 0 & \text{otherwise} \end{cases} \qquad \lambda_i = \begin{cases} 0 & i \in A, \\ 1 & i \in \bar{A} \end{cases}$$

Max-Flow Min-Cut Theorem

Theorem

Quelquesoit la coupe $s-t$, sa capacité $C(A,\bar{A})$ est supérieure ou égale à la valeur ϕ_0 d'un flot de s à t. En outre, la valeur du flot maximum est égale à la capacité de la coupe minimale et un flot ϕ et une coupe de capacité $C(A,\bar{A})$ sont conjointement optimaux ssi

$$\phi_{i,j} = 0 \text{ pour } (i,j) \in E \text{ t.q. } i \in \bar{A} \text{ et } j \in A \tag{27}$$

$$\phi_{i,j} = c_{i,j} \text{ pour } (i,j) \in E \text{ t.q. } i \in A \text{ et } j \in \bar{A} \tag{28}$$

Remarque : La preuve de ce théorème se fait à la base de la théorie de la dualite que sera vue en master..

Fiabilité d'un réseau

On considère le réseau de télécommunication militaire donné figure ci dessous. Il est composé de onze sites (noeuds) reliés par des lignes bidirectionnelles de transmission de données.

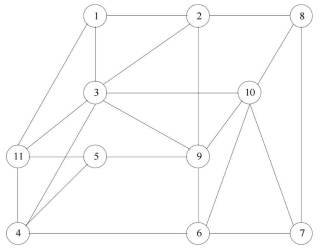


Figure 11.1 – Réseau de télécommunications

Pour des raisons de fiabilité en cas de conflit, le cahier des charges exige que les deux sites 10 et 11 de ce réseau puissent continuer à communiquer malgré la destruction de trois autres sites quelconques.
Ce réseau respecte-t-il cette exigence ?

- Ce problème peut être astucieusement converti en problème de flot maximal (si les sites 10 et 11 peuvent communiquer, il suffit qu'il existe un chemin entre eux).
- Ecrivez un programme linéaire pour résoudre ce problème.

Fiabilité d'un réseau de télécommunication militaire : le modèle linéaire

On oriente d'abord le graphe : avec $\forall$ arête (i,j) on associe les arcs (i,j) et (j,i). On denote par s le sommet 11 et par t le sommet 10. On cherche à maximiser le nombre de chemins de s à t.

- Les variables : $x_{ij} = 1$ si l'arc (i,j) est utilisé par un chemin, 0 sinon ;
- On maximise le nombre de chemins sortant du sommet s :

$$\max \sum_{(s,j)\in E} x_{sj} \tag{29}$$

- La loi de conservation pour chaque sommet intermédiaire $\forall u \in V\backslash\{s,t\}$:

$$\sum_{(u,v)\in E} x_{uv} = \sum_{(v,u)\in E} x_{vu} \tag{30}$$

- On interdit le flot entrant à s.

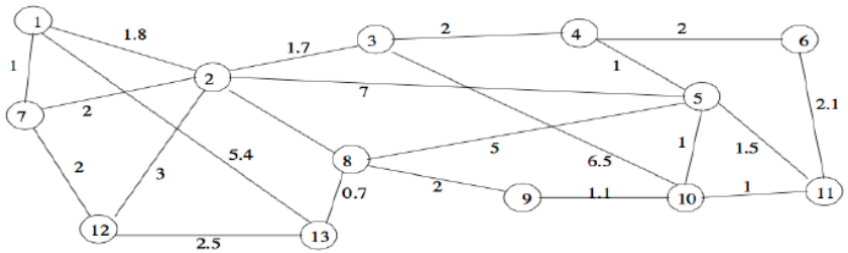
$$\sum_{(u,s)\in E} x_{us} = 0 \tag{31}$$

- On cherche que des chemins "disjoints par sommets", c.a.d.

$$\sum_{(u,v)\in E} x_{uv} \leq 1 \quad \forall u \in V\backslash\{s,t\}. \tag{32}$$

Les lois sur la concurrence et le réseau de télécommunication d'Orange

La figure ci-dessous représente le réseau de télécommunication d'Orange. Les poids sur les arêtes sont proportionnels aux distances $d(i,j)$ entre les sites. Les lois et règlements sur la concurrence et antitrust obligent Orange à déléguer à SFR et à Bouygtel deux sous-réseaux de son réseau. Il s'agit ainsi à trouver une partition de ce réseau en trois parties et chacune de ces parties doit avoir au moins deux sites. De plus, Orange doit renforcer les connections entre les trois sous-réseaux. Le coût de cette opération s'élève à c=25 €/km. [Leo Liberti, Ecole Polytechnique]



- Donner un modèle linéaire pour minimiser le coût des travaux qu'Orange doit effectuer.

Le réseau de télécommunication d'Orange : 1^e solution (non-linéaire)

Soit $G = (V, E)$ le graphe du réseau. On cherche une partition de V en trois sous ensembles disjoints V_1, V_2, V_3 tq la somme des arêtes qui relient V_i avec $V_j, i \neq j$ soit minimale. Ce problème est connu comme "Min-k-Cut problem".

- les parametres du problème :
  - $K = \{1, 2, 3\}$: l'ensemble des sous-ensembles disjoints ;
  - d_{ij} : la distance entre i et j, $\forall (i, j) \in E$;
  - c : le coût unitaire de renforcement des connexions ;
  - m : la taille minimale de chaque sous ensembles V_i.
- les variables
  - Pour $i \in V, h \in K$: $x_{ih} = 1$ si le sommet $i \in V_h$, 0 sinon ;
- les contraintes :
  - $\forall i \in V \sum_{k \in K} x_{ik} = 1$; (affectation des sommets aux sous-ensembles)
  - $\forall h \in K \sum_{i \in V} x_{ih} \geq m$; (cardinalité minimale de chaque sous-ensemble)
- On minimise le coût des travaux sur les arêtes qui relient les sous-ensembles :

$$\min \sum_{h \neq k \in K} \sum_{(i,j) \in E} c d_{ij} x_{ih} x_{jk}$$

Cette une fonction objectif qui est quadratique. Nous montrerons qu'elle peut être linéarisée (le modèle se ramène au modèle linéaire.)

Le réseau de télécommunication d'Orange : la solution linéaire

La fonction objectif

$$\min \sum_{h \neq k \in K} \sum_{(i,j) \in E} c d_{ij} x_{ih} x_{jk} \tag{33}$$

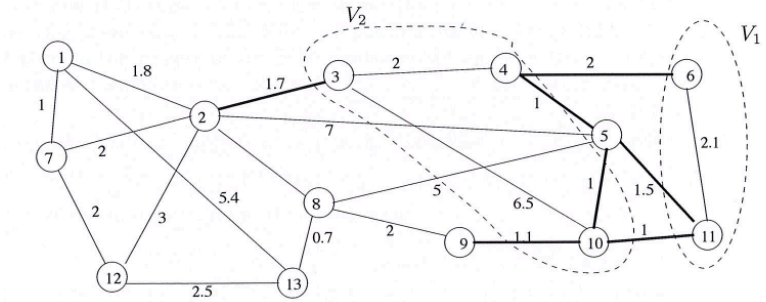
Pour linéariser (33) chaque terme quadratique $x_{ih} x_{jk}$ est remplacé par la variable w_{ij}^{hk} t.q. $0 \leq w_{ij}^{hk} \leq 1$. On obtient ainsi

$$\min \sum_{h \neq k \in K} \sum_{(i,j) \in E} c d_{ij} w_{ij}^{hk}$$

et on ajoute les contraintes suivantes :

- $\forall i, j \in E, h \neq k \in K \; w_{ij}^{hk} \geq x_{ih} + x_{jk} - 1$ (si $x_{ih} = x_{jk} = 1, w_{ij}^{hk} = 1$)
- $\forall i, j \in E, h \neq k \in K \; w_{ij}^{hk} \leq x_{ih}$ (si $x_{ih} = 0, w_{ij}^{hk} = 0$)
- $\forall i, j \in E, h \neq k \in K \; w_{ij}^{hk} \leq x_{jk}$ (si $x_{jk} = 0, w_{ij}^{hk} = 0$).

Le réseau de télécommunication d'Orange : le résultat obtenu pour $k = 3$



Le réseau de télécommunication d'Orange : le résultat obtenu pour $k = 4$