Algorithmique des graphes

Rumen Andonov : CM et TD (G2 en Info et G2 en MIAGE) ,
Yves Mocquard : TD (G1 en Info et G1 en MIAGE)

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 et en largeur. DAGs.
5. Composantes fortement connexes.
6. Les problèmes du plus cours chemins (PCC)
   - Algorithme de Dijkstra. Binary heap (Tas binaire)
   - Algorithme de Bellman-Ford. Découverte des cycles négatives.
   - PCC dans un DAG (Ordre topologique)
7. Algorithmes gourmands pour l'ACM (Minimun spanning tree) : Algs de Prim et de Kruskal
8. Algorithmes pour ordonnancer les tâches. Programmation dynamique.

2/125

Bibliographie

Ce cours est basé essentiellement sur les références suivantes.

- Algorithms, S. Dasgupta, C. H. Papadimitriou, et U. V. Vazirani, McGraw-Hill 2006
- Graphes et algorithmes, Michel Gondran et Michel Minoux, Eyrolles, 1995

3/125

Généralités, notions de base, exemples d'applications

Exemple de graphes I

Obtention d'un diplome 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 :

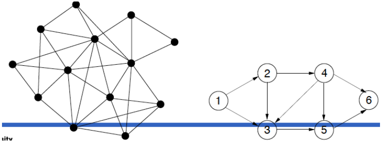
- Système (Programmation)
- AGR1 (Programmation, Math Discrètes)
- Sécurité réseau (Système, Initiation réseau)
- Initiation réseau (Programmation, AGR1)
- Systèmes répartis (AGR1, Système, Initiation réseau)

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.

5/125

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.

6/125

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

- Un *chemin* P dans G est une suite d'arcs dont les extrémités droites/gauches coïncident de la façon suivante : $P = ((u_0, u_1), (u_1, u_2), (u_2, u_3), \dots, (u_{k-1}, u_k))$. 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.a.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*.

7/125

Vocabulaire

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.

Vocabulaire

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

- 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.

9/125

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 ?

10/125

Exemple de graphes II

Construction d'un pavillon

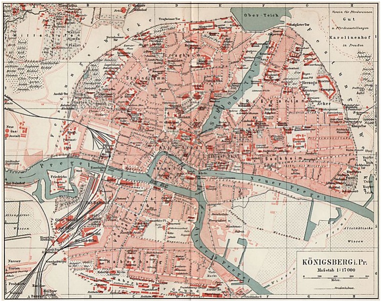
La construction d'un pavillon demande la réalisation d'un certain nombre de tâches. La liste des tâches à effectuer, leur durée et les contraintes d'antériorités à respecter sont données dans le table ci-dessus. Le travail commençant à la date 0, on cherche un planning des opérations qui permet de minimiser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

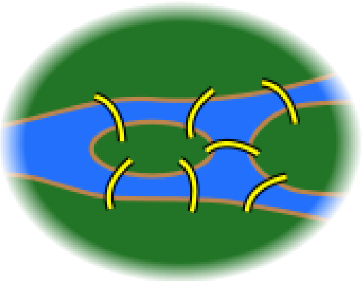
Code tâche	libellé	durée (semaines)	antériorité
A	Travaux de maçonnerie	7	-
B	Charpente de la toiture	3	A
C	Toiture	1	B
D	Installation électrique	8	A
E	Façade	2	D,C
F	Fenêtres	1	D,C
G	Aménagement du jardin	1	D,C
H	Travaux de plafonnage	3	F
J	Mise en peinture	2	H
K	Emménagement	1	E,G,J

FIGURE 1 : Liste des tâches, durée et contraintes

11/125

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 1759) est considéré comme l'origine de la théorie des graphes.

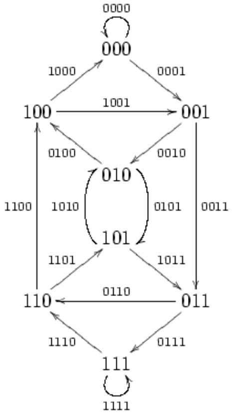
 12/125

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 pairs sauf au plus deux.
  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 $n = 3$.



14/125

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 doiventtraverser. D'autre part, pour des raisons de sécurité, le passeur ne doit jamais laisser seuls lachèvre et le chou, ou le loup et la chère. (Par contre, le loup ne mangera pas le chou... etréciproquement). 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 ?

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 ?

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

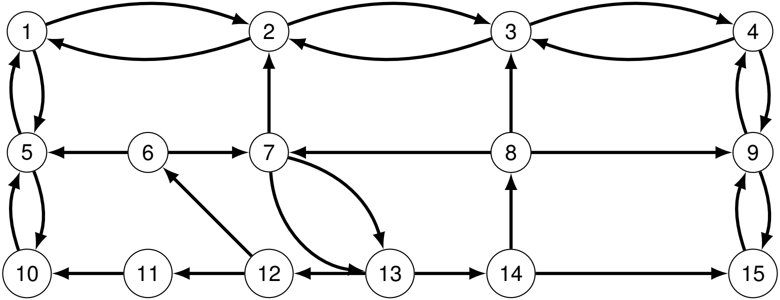
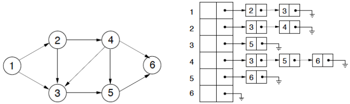


FIGURE 2 : Le plan de sens uniques du centre ville

Représentations des graphes

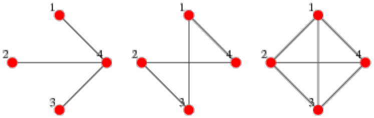
Soit le graphe $G = (V, E)$ où $|V| = n$ et $|E| = m$. G peut être représenté en mémoire soit par des listes d'adjacence, soit par une matrice d'adjacence.

- 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 :

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



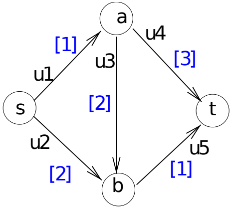
$$\begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix} \quad \begin{pmatrix} 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \end{pmatrix} \quad \begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix}$$

 19/125

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 :

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



$$A = \begin{array}{c|ccccc|} & u1 & u2 & u3 & u4 & u5 \\ \hline s & +1 & +1 & 0 & 0 & 0 \\ t & 0 & 0 & 0 & -1 & -1 \\ a & -1 & 0 & +1 & +1 & 0 \\ b & 0 & -1 & -1 & 0 & +1 \end{array}$$

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

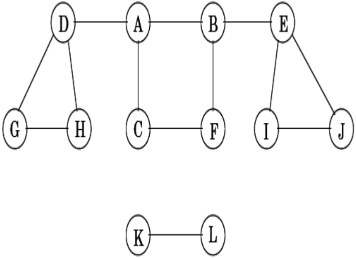
20/125

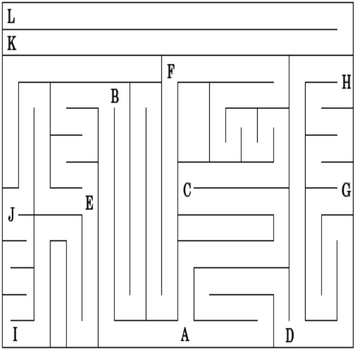
Exploration des graphes :
Parcours en profondeur et parcours en largeur

21/125

Exploration des graphes

- L'exploration d'un graphe (c.a.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)

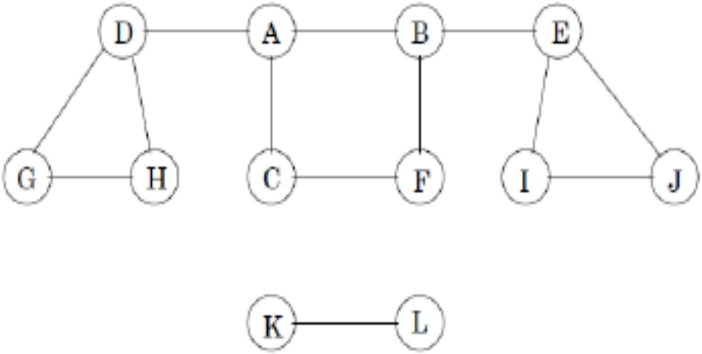




 22/125

Parcours dans les graphes

- Deux parcours principaux :
  - Parcours en profondeur (Depth-first search (DFS)) : avancer sur les arêtes jusque la rencontre d'un sommet déjà visité ; alors retour arrière (backtrack).
  - Parcours en largeur (Breadth-first search (BFS)) : explorer tous les sommets à une distance donnée à partir du sommet de départ ; alors augmenter la distance à parcourir.
  - Exemple : Appliquer ces deux parcours dans le graphe suivant avec le sommet A comme point de départ.



23/125

Parcours en profondeur : 1^{re} 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*.

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

where :

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

24/125

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

```
Initialisation : ∀v ∈ V : v.visited :=false ; clock :=1 ; previsit(s) ; F := [s] ;
u ← P.head() ( u reçoit le sommet de la file F )
while (u ≠ nil ) do
   if (∃(u, v) ∈ E | v.visited = false ) then
      { previsit(v) ; P.put(v) } { v est inséré dans P }
   else
      { postvisit(u) ; delete(u,P) } { le sommet u est supprimé de P }
   end if
   u ← P.head() { u reçoit le sommet de la pile P }
end while
```

where :

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

25/125

Une application

- Appliquer l'algorithme (1) sur le graphe (3).
- Appliquer l'algorithme (2) sur le graphe (3).
- Commenter les différences constatées.

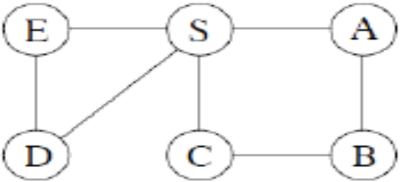


FIGURE 3 : a) Un graphe non-orienté *G*.

26/125

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 :

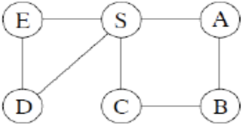
Algorithm 3 BFS(G,s) (2ème version du parcours en largeur d'abord)

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.

Initialisation : $\forall v \in V$: $v.visited := false$ and $v.dist = \infty$; F :=[s] ; s.dist :=0 ; s.visited :=true ;
while ($F \neq \emptyset$) **do**
  $u \leftarrow eject(F)$ { u reçoit le sommet de la file F, le dernier est supprimé de F }
  while ($\exists (u, v) \in E \mid v.visited = false$) **do**
    v.visited :=true
    v.dist :=u.dist+1
    inject(F,v) { le sommet v est inséré dans F }
  end while
end while

27/125

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	$[\,]$

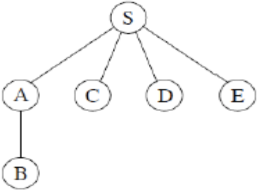


FIGURE 4 : Illustration du fonctionnement de l'algorithme (4)

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

28/125

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 $O(|V| + |E|)$.

Parcours en profondeur d'abord : version récursive

30/125

Exploration de la descendance d'un sommet donné v

Algorithm 4 Explore(G,v)

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

```
v.visited :=true
previsit(v)
for all each edge (v,u)∈E do
    if not u.visited then Explore(G,u)
end for
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.

 31/125

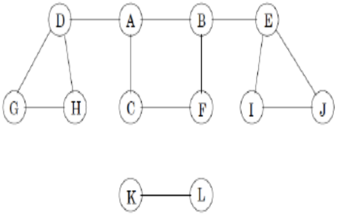
Explore(G,v) : validation formelle

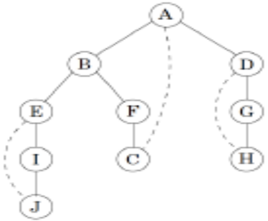
- **Proposition** : Explore(G,v) visite chaque sommet joignable à partir de v.
- *Preuve* : Supposons que le sommet u est joignable à partir de v, mais n'a pas été visité par Explore(G,v). Notons P le chemin allant de v à u. Considérons le dernier sommet z visité par Explore(G,v) sur ce chemin et soit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

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

32/125

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) : depth-first search algorithm

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

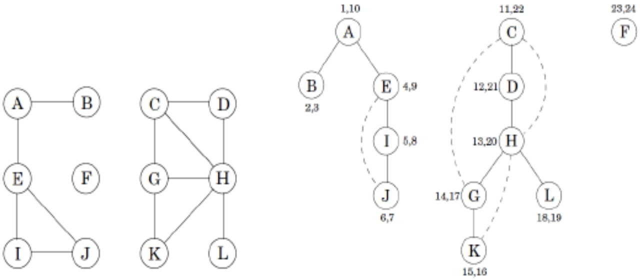
for all v∈V **do**
  v.visited :=false
end for
for all v∈V **do**
  if not v.visited then Explore(G,v)
end for

- 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).

34/125

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).

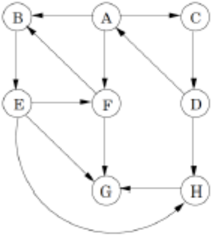
Calcul des étiquettes in/out (prev/post)

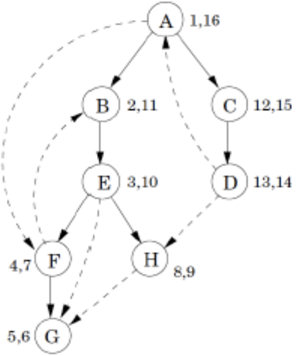
- les valeurs in/out peuvent être calculées dans les procédures previsit et postvisit.
- On utilise pour ce faire la compteur "clock" initialisé à 1 et défini comme suit.
  - previsit(v)={v.in :=clock ; clock :=clock+1}
  - postvisit(v)={v.out :=clock ; clock :=clock+1}
- **Proposition** : Pour chaque 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.

36/125

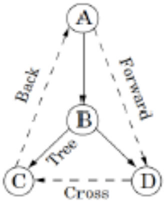
Parcours en profondeur dans les graphes orientés

- 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).





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.a.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

 39/125

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

- **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 : $v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_k \rightarrow v_0$. 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 $v_{i-1} \rightarrow v_i$ est un arc retour (si $i = 0$ alors $v_k \rightarrow v_0$).

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 Algorithme alternatif pour linéariser un DAG

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

Composantes fortement connexes

 42/125

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 unchemin 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.

$$uRv \equiv (u \in \Gamma_v^*) \wedge (v \in \Gamma_u^*) \tag{1}$$

  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.a.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 (5).

44/125

Application de DFS : Composantes fortement connexes (suite)

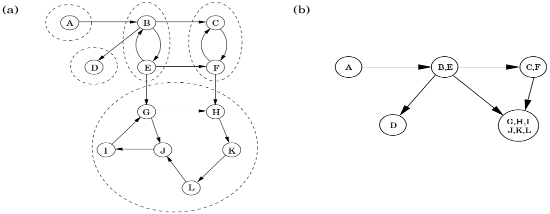


FIGURE 5 : a) Un graphe orienté G et ses c.f.c. ; b) le graphe réduit $\tilde{G}$.

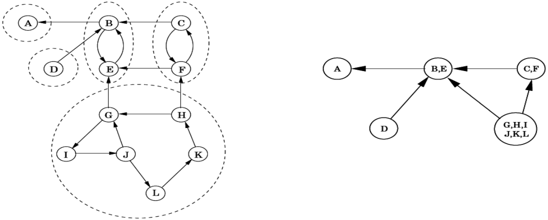


FIGURE 6 : a) G_R : le graphe renversé du graphe G et ses c.f.c. ; b) $\tilde{G_R}$: le graphe réduit du G_R.

45/125

Application de DFS : Composantes fortement connexes (suite)

- 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 :

$$\max_{u \in C}\{post(u)\} > \max_{u \in C^{'}}\{post(u)\} \tag{2}$$

- *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, $\underset{u \in C}{post}(u) > \underset{u \in C^{'}}{post}(u)$.

Application de DFS : Composantes fortement connexes (suite)

- Grace à la propriété 3, nous savons comment trouver une c.f.c. source dans G comment trier topologiquement 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.a.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 (5) et (6).
- 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.

Application de DFS : Composantes fortement connexes (suite)

Algorithm 8 determining the strongly connected components (SCC) of a digraph 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. (5.a)

48/125

Calcul des chemins les plus courts dans les graphes

Le cas des valeurs positives des arcs/arêtes

49/125

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

Algorithm 9 procedure 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

Initialisation : $\forall u \in V$: $dist(u) := \infty$ and $prev(u) := nil$; $dist(s) := 0$;
$P := \{\emptyset\}$, (P contient les sommets v, tq $dist(v)$ est calculée définitivement)
$T := \{V\}$ (T contient des sommets v, tq $dist(v)$ n'est qu'une borne supérieure)
while T is not empty **do**
    $u := \underset{v \in (T)}{\operatorname{argmin}}\, dist(v)$
    $P := P \bigcup \{u\}$
    $T := T \backslash \{u\}$
    for all edges $(u, v) \in E$ **do**
        update_a ($dist(v)$)
    end for
end while

- où

50/125

Algorithme de Dijkstra (suite)

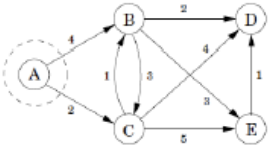
Algorithm 10 procedure update_a($dist(v)$)

```
if dist(v) > dist(u) + l(u,v) then
    dist(v) = dist(u) + l(u,v)
    prev(v) :=u
end if
```

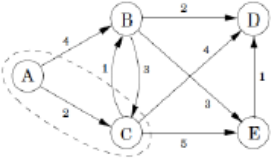
Exemple :

 51/125

Application de l’algorithme de Dijkstra sur un graphe.



A: 0	D: ∞
B: 4	E: ∞
C: 2	

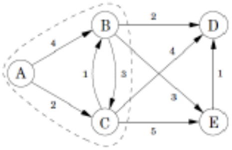


A: 0	D: 6
B: 3	E: 7
C: 2	

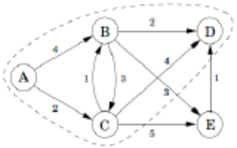
FIGURE 7 : A est le sommet de départ dans cet exemple. Les valeurs de dist sont données sont données dans la table associée.

 52/125

Application de l'algorithme de Dijkstra sur un graphe.



A: 0	D: 5
B: 3	E: 6
C: 2	



A: 0	D: 5
B: 3	E: 6
C: 2	

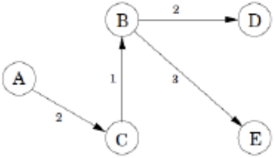


FIGURE 8 : A est le sommet de départ dans cet exemple. Les valeurs de dist sont données sont données dans la table associée. L'arbre des chemins les plus courts est aussi visualisé.

53/125

Justification de l'alg. de Dijkstra

- Soit $G = (V, E, l_e > 0 : e \in E)$. On cherche les PCC(s⇝u) : les chemins les plus courts du s aux $u \in V \backslash \{s\}$.
- Soit $dist^*(u)$: la longueur du PCC(s⇝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.a.d.

$$u := \operatorname*{argmin}_{v \in (T)} dist(v)) \tag{3}$$

Alors $dist^*(u) = dist(u)$.
- **Preuve :** Soit v tq $v = prev(u)$ sur le PCC(s⇝u). Puisque $l(v,u) > 0 \Rightarrow v \in P$ (sinon contradiction avec (3)). C.a.d. une seule arête sépare u de l'ensemble P (voir Fig. (9)). 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)

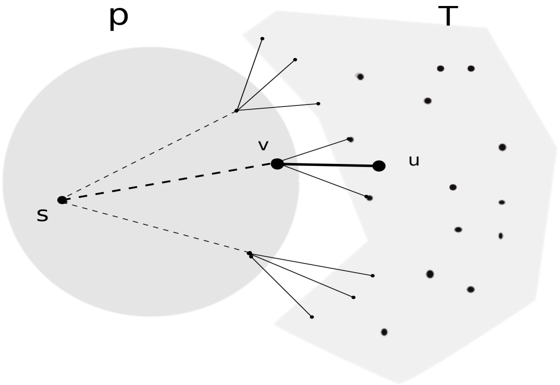


FIGURE 9 : P est élargi avec le sommet u tq $dist(u) = \min_{v \in P, v \in \Gamma^{-1}(u)} dist(v) + l(v,u) = dist^*(u)$.

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

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

55/125

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 :** un tas si :

$$\pi(j) \geq \pi(\lfloor \frac{j}{2} \rfloor) \quad \forall j \tag{4}$$

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

- On peut représenter le tas par un **arbre parfait partiellement ordonné** (c.a.d.) :
  - $\forall$ niveau est rempli de gauche à droite
  - $\forall$ niveau est complètement rempli avant de commencer le niveau suivant
  - condition (4) 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.

56/125

Tas binaire

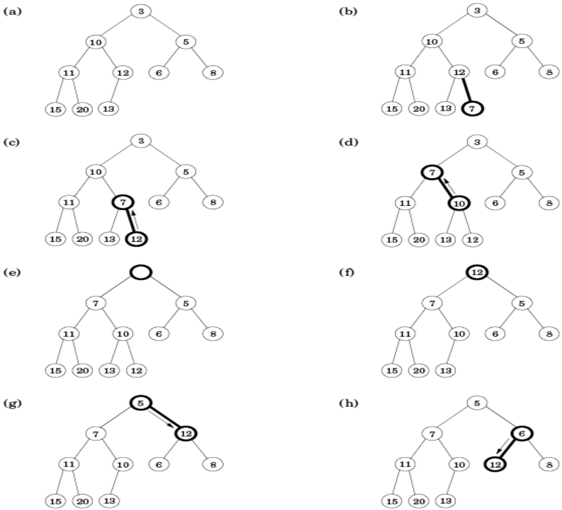


FIGURE 10 : (a) : un tas binaire avec 10 éléments. Seulement les clés sont indiquées. (b)-(d) : Illustration de l'opération "insertion". (e)-(g) : Illustration de l'opération "extraction et suppression du minimum".

57/125

Version tas binaire de l'alg. de Dijkstra

Algorithm 11 procedure Dijkstra_b (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

$\forall u \in V$: initialize $dist(u) := \infty$ and $prev(u) := nil$

$dist(s) := 0$

$P := \{s\}$ {P contient les sommets v pour lesquels la valeur $dist(v)$ est calculée définitivement}

$T :=$ **makequeue(V)** {créer une file de priorité T avec $dist(v), v \in (T)$ comme clés}

while T is not empty **do**

    $u :=$ **ExtractMin(T)**

    $P := P \bigcup \{u\}$

    $T := T \setminus \{u\}$

    for all edges $(u, v) \in E$ **do**

        update_b(T,(u, v))

    end for

end while

58/125

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

Algorithm 12 procedure update_b(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**
  $dist(v) = dist(u) + l(u, v)$
  prev(v) :=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 8) nécessite $O(|V|)$ **ExtractMin** plus $O(|E|)$ **ChangeKey (updates)** : $O((|V| + |E|) \log |V|)$ opérations.
- En total : $O((|V| + |E|) \log |V|)$.

59/125

Calcul des chemins les plus courts dans les graphes

Présence de poids négatifs

60/125

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 procedure 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 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) $| PCC(s \rightsquigarrow v) | \leq | V | -1$

(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 $dist(v) = dist^*(v)$.

Présence de poids négatifs : Algorithme de Bellman-Ford

Algorithm 14 procedure Bellman-Ford (G,l,s)

Require: Graphe G=(V,E) directed, edge weights $\{l_e : e \in E\}$ with no negative cycles ; vertex $s \in V$

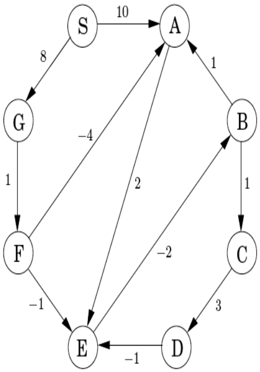
Ensure: For all vertices $u \in V$ reachable from s, $dist(u)$ is set to the shortest distance from s to u ;

for all $u \in V$ **do**
    $dist(u) := \infty$
    $prev(u) := nil$
end for
$dist(s) := 0$
k :=1
while $(k \leq |V| - 1)$ **do**
    for all edges $(u, v) \in E$ **do**
        $dist(v) = \min\{dist(v), dist(u) + l(u, v)\}$
    end for
    k :=k+1
end while

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

62/125

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 11 :

63/125

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

$$dist(v) = \min_{u \in \Gamma^{-1}(v)} \{dist(u) + l(u,v)\} = \min\{dist(v), \min_{u \in \Gamma^{-1}(v)} \{dist(u) + l(u,v)\}\} \quad (5)$$

On obtient la variante suivante.

Algorithm 15 procedure 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.

```
Initialisation : ∀u ∈ V : dist^0(u) := ∞ ;  dist^0(s) := 0 ;  stable := false ;  k :=1 ;
while ((k ≤ |V|) and (nonstable)) do
    dist^k(s) := 0
    for all vertices v ∈ V \ {s}  do
        dist^k(v) = min{dist^{k-1}(v), min_{u ∈ Γ^{-1}(v)} dist^{k-1}(u) + l(u,v)}
    end for
    stable := (dist^k(v) = dist^{k-1}(v), for all v ∈ V ) ; k :=k+1 ;
end while
if (k = |V| + 1) then ∃ negative cycle
end if
```


64/125

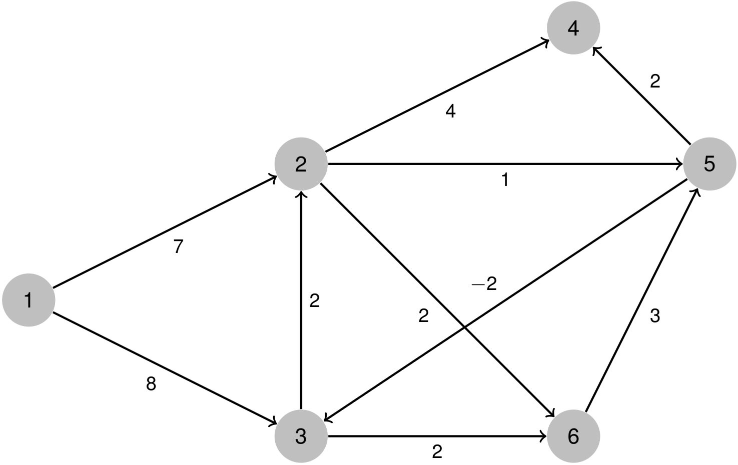
Illustration pour l’algorithme de Bellman-Ford_bis

65/125

Techniques d'accélération de l'algorithme de Bellman-Ford_bis

- $dist^k(v)$ 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.

Accélérations posibles pour l'algorithme de Bellman-Ford_bis (exemple)



67/125

Calcul des chemins les plus courts/longues dans les graphes

L'approche programmation dynamique

68/125

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.a.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".

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.a.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 (16).
- 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 16 procedure 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

Initialisation : $\forall u \in V : d^-(u) := |\Gamma^{-1}(u)|$; $k := 0$; $S_0 := \{s\}$;
while $|S_k| > 0$ **do**
    $S_{k+1} := \emptyset$
    for all $u \in S_k$ **do**
        $rank(u) := k$
        for all edges $(u,v) \in E$ **do**
            $d^-(v) := d^-(v) - 1$
            if $(d^-(v) = 0)$ **then**
                $S_{k+1} := S_{k+1} \bigcup \{v\}$
            end if
        end for
    end for
    $k := k + 1$
end while

- Complexité de l'algorithme (16) : $O(|V| + |E|)$.

71/125

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

72/125

Cas des graphes sans circuit (DAG) (suite)

Algorithm 17 procedure 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

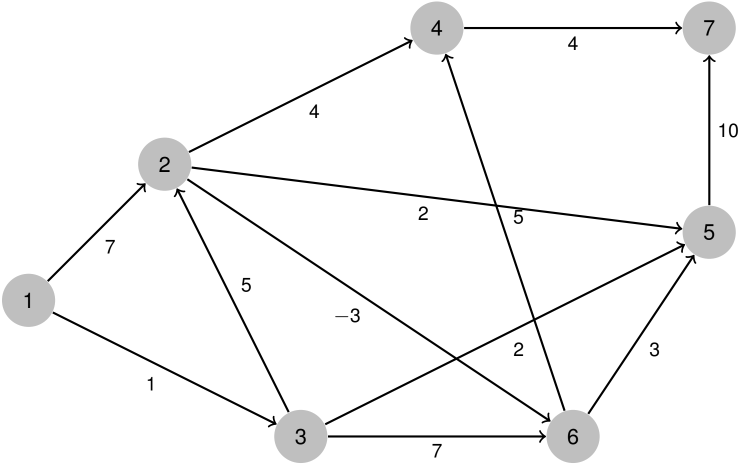
Ensure: For all vertices u reachable from s, $dist(u)$ is set to the distance from s to u

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

- Complexité de l'algorithme (17) : $O(|V|)$ + le coût des boucles (5) et (6) ($\Sigma_u d^+ u$) = $O(|V| + |E|)$.

73/125

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



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 (17) est donné par l'algorithme suivant.

Algorithm 18 procedure 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
```


75/125

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

Une variante de l'algorithme (18) est la suivante :

Algorithm 19 procedure 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 $u \in V$ **do**

  $dist(u) := \infty$

end for

$dist(s) := 0$

linearize G

for all $v \in V \setminus \{s\}$, in linearized order **do**

  for all edges $(u,v) \in E$ **do**

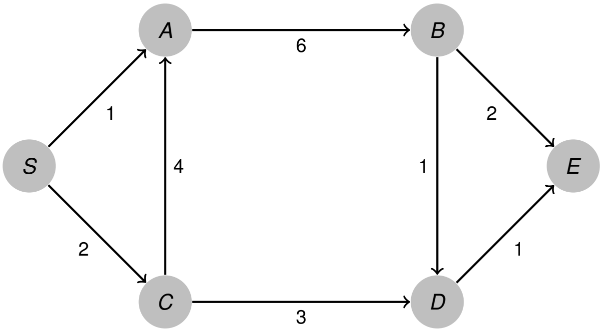
    $dist(v) = \min\{dist(v), dist(u) + l(u,v)\}$

  end for

end for

76/125

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

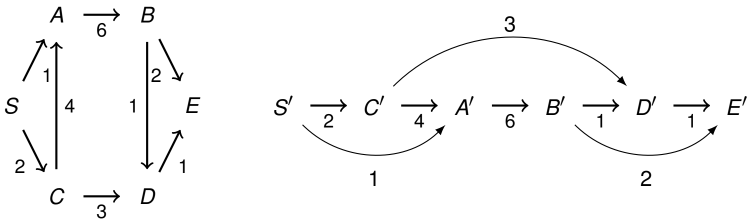


77/125

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.a.d. numéroter les sommets de façon que pour chaque arc $(u, v) \in E$ on a $num(u) < num(v)$).
- Exemple

FIGURE 12 : A gauche : un graphe G ; A droite : Le même graphe après l'avoir trié/ordonné topologiquement/linéairement).



 78/125

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

79/125

Les principes de la programmation dynamique : rappel

- L'algorithme (19) 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)).

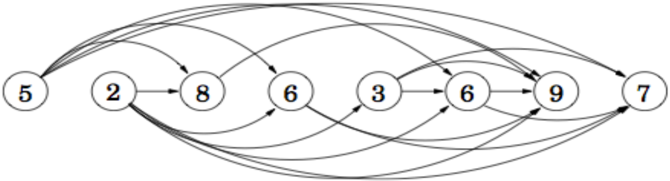
Exemple : La plus longue sous-séquence croissante de $S = 5, 2, 8, 6, 3, 6, 9, 7$ est $2, 3, 6, 9$.

81/125

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 correspondance" entre les chemins dans G et les sous-séquences croissantes dans S.
- L'objective est de trouver le plus longue chemin (PLC) dans ce graphe.

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



82/125

Algorithme pour la plus longue sous-séquence croissante

Algorithm 20 Longest increasing subsequence algorithm

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 $\max_j L(j)$

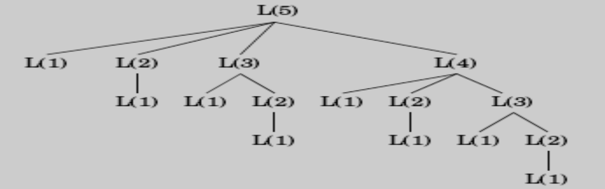
where $\max\{\} = 0$.

- 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 (20) : 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 (20) ne trouve que la longueur de la PLSSC. Pour connaitre la séquence même, il faut sauvegarder les indices argmax de la ligne (2).

 83/125

Attention aux appels récursifs naïfs ! ! ! !

- La ligne 2 de l'algorithme (20) 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).

84/125

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 "gap" (omission). 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.

La distance d'édition par la programmation dynamique

- 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	

86/125

La distance d'édition par la programmation dynamique

- 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.

x[i]	où	_	où	x[i]
_		y[j]		y[j]

- 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.a.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.a.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.a.d. le sous-problème $E(i-1,j-1)$) + 1 si $x[i] \neq y[j]$ et 0 si $x[i] = y[j]$.

87/125

La distance d'édition par la programmation dynamique

- $E(i,j)$ 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.

La distance d'édition par la programmation dynamique

Algorithm 21 Dynamic programming algorithm for edit distance

---

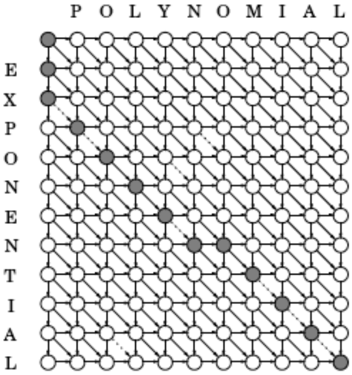
for $i = 0$ to m **do**
    $E(i,0) = i$
end for
for $j = 0$ to n **do**
    $E(0,j) = j$
end for
for $i = 0$ to m **do**
    for $j = 0$ to n **do**
        $E(i,j) = \min\{1 + E(i-1,j), 1 + E(i,j-1), \text{diff(i,j)} + E(i-1,j-1)\}$
    end for
end for
return $E(m,n)$

89/125

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).



Recherche du plus courts chemin entre chaque couple de sommets :

Algorithme de Floyd-Warshall

 91/125

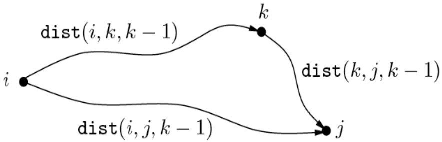
Recherche du plus courts chemin entre chaque couple de sommets

- L'application de l'alg. de Bellman-Ford à partir de chaque sommet génère une complexité de $O(|V|^2|E|)$.
- Une meilleur alternative est l'alg. de Floyd-Warshall qui a une complexité en $O(|V|^3)$.
- Idée : les sous-problèmes sont obtenus par restriction sur les sommets intermédiaires admissibles.
- Numérotons les sommets de V par $\{1, 2, \ldots, n\}$ et soit $dist(i, j, k)$ la longueur du $PCC(i \rightsquigarrow j)$ qui ne comporte que les sommets $\{1, 2, \ldots, k\}$ comme sommets intermédiaires.

92/125

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 :
- $dist(i,j,k) := \min\{dist(i,j,k-1), dist(i,k,k-1) + dist(k,j,k-1)\}$.



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

Algorithm 22 procedure 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 $i, j, \in V$

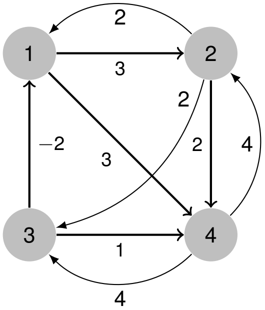
for i = 1 to n **do**
    for j = 1 to n **do**
        $dist(i, j, 0) := \infty$
    end for
end for
for all $(i, j) \in E$ **do**
    $dist(i, j, 0) := l(i, j)$
end for
for k = 1 to n **do**
    for i = 1 to n **do**
        for j = 1 to n **do**
            $dist(i, j, k) := \min\{dist(i, j, k-1), dist(i, k, k-1) + dist(k, j, k-1)\}$
        end for
    end for
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.

94/125

Algorithme de Floyd-Warshall : illustration

FIGURE 13 : Appliquer l'alg. de Floyd-Warshall au graphe suivant.



95/125

Arbres couvrants minimaux (ACM)

96/125

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.

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.*

97/125

Algorithme générique pour la construction d'un ACM

Algorithm 23 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.

    $X := \emptyset$
    while X ne forme pas un arbre couvrant **do**
        trouver pour X une arête sure (u, v)
        $X := X \cup \{(u, v)\}$
    end while
return X

98/125

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 24 function ACM_Prim (G(V,E),w,r)

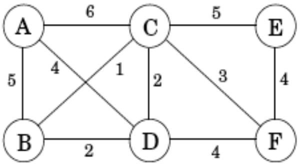
Require: Un graphe $G = (V, E)$ avec des poids $w_e, e \in E$. r est le sommet initial.
Ensure: Un ACM défini par les pointeurs π tq $\pi(v)$ désigne le père de v dans l'arbre.

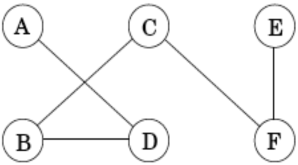
1: **Initialisation :** $\forall u \in V$: $clef(u) := \infty$ and $\pi(u) := nil$; $S := \emptyset$; $clef(r) := 0$;
2: $F :=$ **maketbl(V)** {crée un tbl F avec $clef(v), v \in (V)$ comme clés}
3: **while** F is not empty **do**
4:     $u :=$ **ExtractMin(F)** {l'élément u sera considéré hors F }
5:     $S := S \cup \{u\}$
6:     **for all** edges $(u, v) \in E$ **do**
7:         **if** $clef(v) > w(u, v)$ **then**
8:             $clef(v) = w(u, v)$
9:             $\pi(v) := u$
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 $O(|V|^2 + |E|) = O(|V|^2)$.

99/125

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





Set S	A	B	C	D	E	F
$\{\}$	$0/\mathrm{nil}$	∞/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 14 : Un graphe et son arbre couvrant minimal trouvé par l'algorithme de Prim

Algorithme de Kruskal pour la construction d'un ACM

Algorithm 25 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
  $X := \emptyset$
  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**
      $X := X \cup \{(u, v)\}$
      union(u,v)
    end if
  end for
return X

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

101/125

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

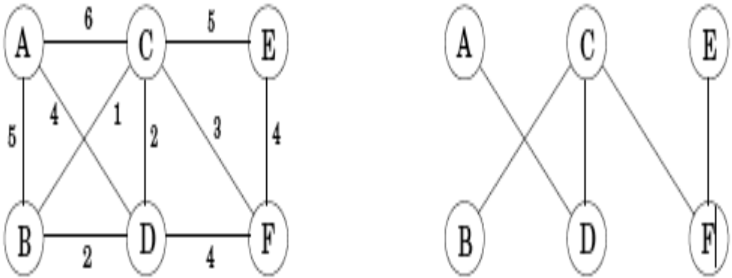


FIGURE 15 : Un graphe et son arbre couvrant minimal trouvé par l'algorithme de Kruskal

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.

103/125

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

Algorithm 26 procedure makeset (x)

Ensure: crée un nouvel ensemble dont le seul élément (et donc le représentant) est x.

$\pi(x) := x$

$rank(x) := 0$

Algorithm 27 function find(x)

Ensure: retourne un pointeur vers le représentant de l'ensemble (unique) contenant x.

while x$\neq \pi(x)$ **do**

  $x := \pi(x)$

end while

return x

Algorithm 28 function find_bis(x)

Ensure: effectue une compression du chemin lors de l'opération find

if x$\neq \pi(x)$ **then**

  $\pi(x)$:=find($\pi(x)$)

end if

return $\pi(x)$

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

Algorithm 29 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 := find(x)
r_y := find(y)
if $r_x = r_y$ **then**
  return
end if
if rank(r_x) $>$ rank(r_y) **then**
  $\pi(r_y) := r_x$
else
  $\pi(r_x) := r_y$
  if rank(r_x) $=$ rank(r_y) **then**
    rank(r_y) $:=$ rank$(r_y) + 1$
  end if
end if

Illustration pour les opérations sur les ensembles disjoints

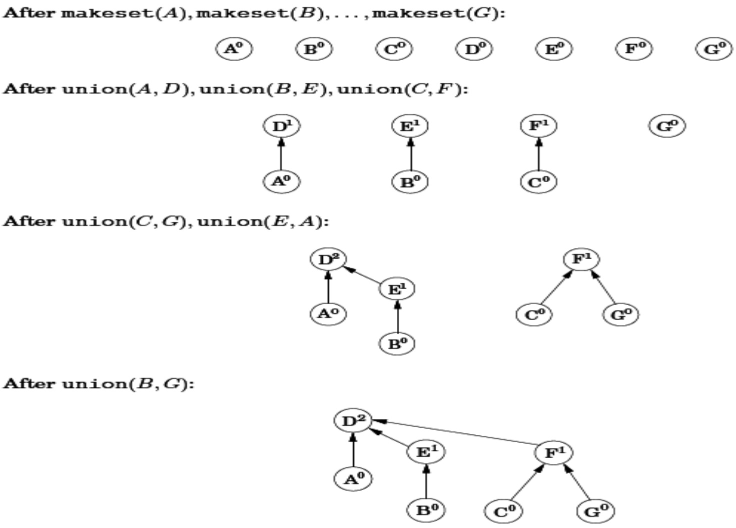


FIGURE 16 : Résultat des opérations **makeset** et **union** sur des ensembles disjoints

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).

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

Chaque élément est caractérisé par le couple (nom, valeur numérique) $(i, \pi(i))$.
Définition : un tas si :

$$\pi(j) \geq \pi(\lfloor \frac{j}{2} \rfloor) \quad \forall j \tag{6}$$

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

On peut représenter le tas par un **arbre parfait partiellement ordonné** (c.a.d.) :

- $\forall$ niveau est rempli de gauche à droite
- $\forall$ niveau est complètement rempli avant de commencer le niveau suivant
- condition (6) 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.

Illustration pour les opérations sur les ensembles disjoints

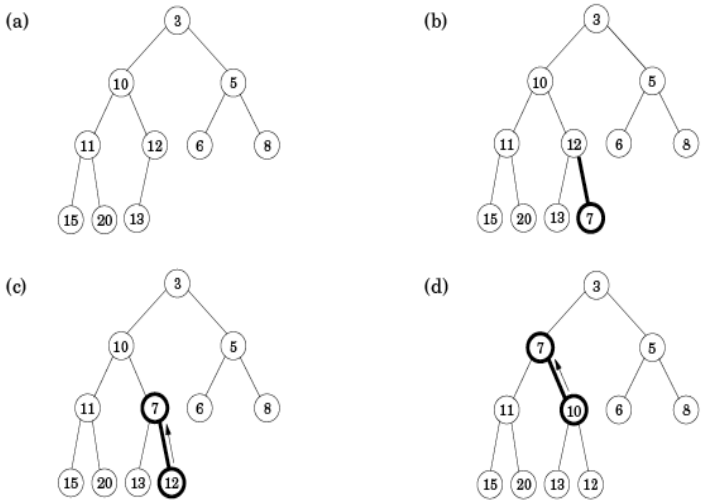


FIGURE 17 : a) : Un tas binaire avec 10 éléments. Ne sont présentées que les valeurs clés $(\pi(i))$; b)-d) : Insertion d'un élément ayant une clé de valeur 7 ;

Illustration pour les opérations sur les ensembles disjoints (suite)

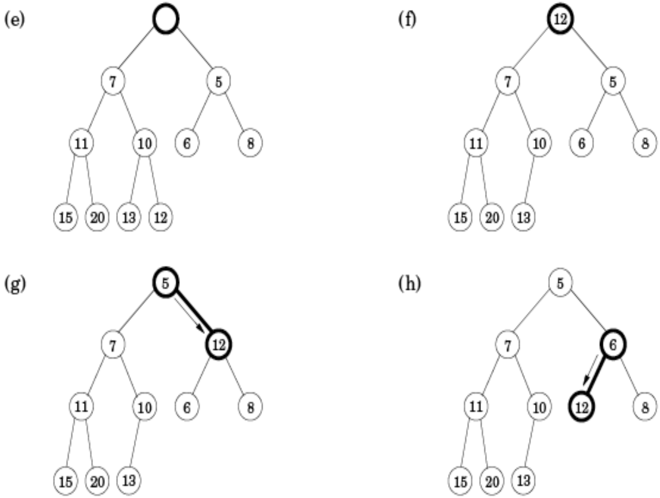


FIGURE 18 : (e)-(h) : Suppression de l'élément avec la clé de valeur minimale.

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

Algorithm 30 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 :** $\forall u \in V$: $clef(u) := \infty$ and $\pi(u) := nil$; $clef(r) := 0$;
2: $F :=$ **makequeue(V)** {crée une file de priorité F avec $clef(v), v \in (V)$ comme clés}
3: **while** F is not empty **do**
4:     $u :=$ **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:             $\pi(v) := u$
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 : $O(|V| \log |V| + |E| \log |V|) = O(|E| \log |V|)$.

111/125

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

112/125

Les réseaux de transport : définitions

- Soit $R = (X, U, C)$ un graphe connexe (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.

$$\sum_{e \in \Gamma^+(u)} \phi_e = \sum_{e \in \Gamma^-(u)} \phi_e \tag{7}$$

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

$$\sum_{e \in \Gamma^+(s)} \phi_e = \sum_{e \in \Gamma^-(t)} \phi_e = \phi_0 \text{ (valeur du flot)} \tag{8}$$

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

Exemple de réseau

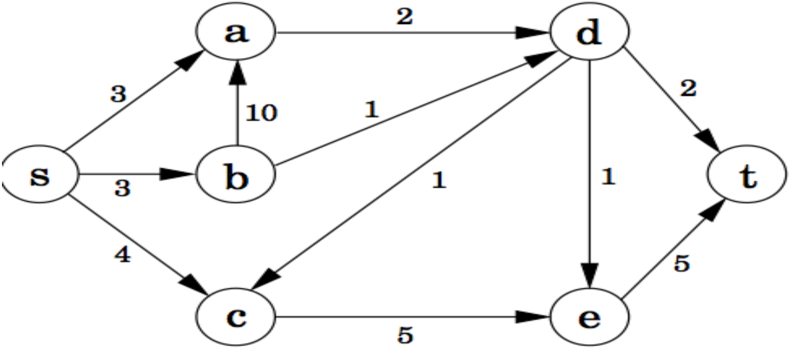


FIGURE 19 : Trouver le flot maximum entre les sommets *s* et *t*

Définition d'une coupe et de sa capacité

- *Une coupe séparant s et t* (notée $s - t$) est une partition $(A, \bar{A})$ of X, telle que $s \in A$ and $t \in \bar{A} = X \setminus A$.
- *La capacité $C(A, \bar{A})$ de la coupe $s - t$* est définie par $C(A, \bar{A}) = \sum_{e \in \Gamma^+(A)} c_e$.
- Lemme : La valeur max d'un flot de s à t compatible avec c_u n'excède jamais la capacite d'une coupe séparant s et t.
- On démontrera le théorème suivant :
  La valeur max d'un flot de s à t est égale à la capacité d'une coupe de capacité minimale séparant s et t.

115/125

Graphe d'écart

- Soit $\phi = [\phi_1, \phi_2, \dots \phi_M]^T$ 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 amélorant 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 amélorant π.
- Alors on peut obtenir un nouveau flot de valeur $\phi_0^{'} = \phi_0 + \varepsilon$.

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 $\exists$ un chemin amélorant $s \to t$ dans $\bar{G}(\phi^k)$ faire

  soit π^k un chemin amélorant de $s \to t$ dans $\bar{G}(\phi^k)$.
  Soit ε^k le minimum des capacités résiduelles du π^k.
  Définir le flot ϕ^{k+1} par

$$\phi_u^{k+1} = \phi_u^k + \varepsilon^k \text{ si } u^+ \in \pi^k \tag{9}$$

$$\phi_u^{k+1} = \phi_u^k - \varepsilon^k \text{ si } u^- \in \pi^k \tag{10}$$

$$\phi_0^{k+1} = \phi_0^k + \varepsilon^k \tag{11}$$

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

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 amélorant.
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 $L = \{v \in X : \text{il existe un chemin de} s à t dans \bar{G}(\phi)\}$.
  Soit $R = X \setminus L$. La partition (L, R) est une coupe : de façon triviale $s \in L$ et $t \in R$.
  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 $\phi(v, u) = 0$.
- $(3) \Rightarrow (1)$: Conséquence du lemme.

Ordonnancement des tâches

119/125

Ordonnancement des tâches

Problème : Déterminer l'ordre et le calendrier d'exécution des N tâches, soumises à des contraintes de successions dans le temps, afin de minimiser la durée totale.

L'approche "graphe potentiels-tâches"

A partir du projet donné on construit un graphe sans circuit (DAG) de la façon suivante :

- A $\forall$ tâche i, on associe un sommet $\textcircled{i}$;
- On ajoutera l'arc (i, j) de longueur d_i, noté par $\textcircled{i} \xrightarrow{d_i} \textcircled{j}$, si la tâche i de durée d_i doit précéder la tâche j ;
- On ajoute deux tâches fictives α (début) et ω (fin) ;
- α est reliée aux sommets sans prédécesseurs ;
- ω est reliée aux sommets sans successeurs.
- On représente le graphe par niveaux après avoir calculé sa fonction rang ;

120/125

Exemple : 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 20 : Liste des tâches, durée et contraintes

121/125

Notions principales :

- *La date au plus tôt* t_i de début de la tâche i : $t_i = \max\limits_{j \in \Gamma_i^{-1}} (t_j + d_j)$
  (c.a.d. la longueur du *plus long chemin* de α à i $l(PLC(\alpha \to i))$).
- La *durée minimale* t_ω : $l(PLC(\alpha \to \omega))$.
- Si la durée du projet est fixé à t_ω, la *date au plus tard* T_i pour commencer la tâche i sans influencer t_ω est :
  $T_\omega = t_\omega;\ T_i = \min_{j \in \Gamma_i} (T_j - d_i) \Rightarrow T_i = t_\omega - l(PLC(i \to \omega))$
- La *marge* m_i de la la tâche i : $m_i = T_i - t_i$.
- Si $m_i = 0$, la tâche i est dite *tâche critique*.

Remarque : Entre deux tâches i et j, tq $j \in \Gamma_i$, on doit avoir $t_j \geq t_i + d_i$. La valeur t_i, associée à $\forall i$, peut être considérée comme un potentiel, d'où la dénomination *graphe potentiels-tâches*.
Calcul : Pour calculer les PLC on appliquera la programmation dynamique dans un DAG avec l'opérateur max.

Ordonnancement des tâches : dates au plus tôt et dates au plus tard

Soit le graphe G=(V,E) orienté, sans circuit, avec les durées des tâches comme des poids sur les arcs $\{d_e : e \in E\}$, le sommet α pour le début des tâches, le sommet ω pour la fin ;

Algorithm 31 procedure Dates_au_plus_tôt (G, d, α, ω)

Ensure: Les dates au plus tôt t_i pour $\forall i$;

  $t_\alpha := 0$

  for all $i \in V$ **do** dans l'ordre croissant des rangs

    $t_i := \max\limits_{j \in \Gamma_i^{-1}} (t_j + d_i)$

  end for

Algorithm 32 procedure Dates_au_plus_tard (G, d, α, ω)

Require: Que t_ω soit calculé ;

Ensure: Les dates au plus tard T_i pour $\forall i$;

  $T_\omega := t_\omega$;

  for all $i \in V$ **do** dans l'ordre décroissant des rangs

    $T_i := \min\limits_{j \in \Gamma_i} (T_j) - d_i$

  end for

Compléments pratiques

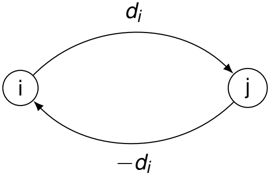
Les contraintes de type potentiel (la tâche j doit commencer après la fin de i, ou bien après la moitié de la réalisation de de i, ou bien un certain temps après la fin de i, etc) se modélisent facilement dans l'approche "graphe potentiels-tâches".

- la contrainte : j ne doit pas commencer avant la moitié de la réalisation de la tâche i se représente par un arc supplémentaire (i, j) de longueur $\frac{d_i}{2}$, c.a.d.
  $$i \xrightarrow{d_i/2} j.$$
- la contrainte : j ne peut commencer qu'un temps t après la fin de i se représente par un arc (i, j) de longueur $d_i + t$, c.a.d. $i \xrightarrow{d_i + t} j$.
- la contrainte : j ne peut commencer qu'après la date b_j, se représente par un arc (α, j) de longueur b_i, c.a.d. $\alpha \xrightarrow{b_j} j$.
- la contrainte : j doit commencer avant la date c_j, se représente par un arc (j, α) de longueur $-c_j$, $\alpha \xleftarrow{-c_j} j$

124/125

Compléments pratiques (suite)

- la contrainte : j doit suivre immédiatement i, s'écrit $t_i + d_i = t_j$ et se représente par un arc (i, j) de longueur d_i et un arc (j, i) de longueur $-d_i$



Remarque : Les deux dernières contraintes introduisent des circuits et des arcs de longueurs négatives. La condition d'existence d'un ordonnancement deviendra alors *il n'existe pas de circuit de longueur strictement positive* (au lieu de *il n'existe pas de circuit*). On devra utiliser un des algorithmes vus en cours plus adaptés à de telles situations que la programmation dynamique. On veillera aussi à remplacer chaque fois l'opérateur min par max (puisqu'on cherche des plus longs chemins).