

COMPLÉMENTS SUR LES GRAPHS

I - Rappels de première année

I.1 - Graphes non orientés, Connexité, Arbres

Définition 1

Chemin, chemin simple, chemin élémentaire :

- Un **chemin** de longueur n dans un graphe $G = (S, A)$ est une suite de $n + 1$ sommets x_0, \dots, x_n tels que $\{x_i, x_{i+1}\} \in A$ pour tout $i \in [0 \dots n - 1]$.
- Un **chemin simple** est un chemin n'utilisant pas deux fois la même arête : les $\{x_i, x_{i+1}\}$ sont distincts.
- Un **chemin élémentaire** est un chemin n'utilisant pas deux fois le même sommet : les x_i sont distincts.
- Un **cycle** est un chemin simple non réduit à un sommet vérifiant $x_0 = x_n$.

Un graphe $G = (S, A)$ non orienté est dit **connexe** si pour tous sommets $x, y \in S$, il existe un chemin reliant x et y .

Remarques :

- Un chemin de longueur zéro contient un sommet et aucune arête, et relie un sommet à lui-même.
- Un cycle est nécessairement de longueur supérieure ou égale à 3 (\triangle dans un graphe non orienté).
- Faites bien attention à lire attentivement les définitions du sujet à chaque fois : il y a des variations.

Proposition 1

Les trois propositions suivantes sont équivalentes :

- Il existe un chemin reliant x à y .
- Il existe un chemin simple reliant x à y .
- Il existe un chemin élémentaire reliant x à y .

Démonstration. (3) \Rightarrow (2) et (2) \Rightarrow (1) sont évidentes. Supposons donc l'existence d'un chemin entre x et y , et notons $x = x_0, x_1, \dots, x_n = y$ un tel chemin **de longueur minimale**. Ce chemin est nécessairement élémentaire : en effet, si on avait $x_i = x_j$ avec $i < j$, alors $x_0, \dots, x_i, x_{j+1}, \dots, x_n$ fournirait un chemin strictement plus court. \square

Définition 2

Arbre, forêt :

Un **arbre** est un graphe acyclique et connexe.
Une **forêt** est un graphe acyclique.

Remarque :

- Un graphe est donc une forêt si et seulement si chacune de ses composantes connexes est un arbre.

Définition 3

Pour un graphe $G = (S, A)$ et une paire $a = \{x, y\}$ d'éléments de V , on note :

- $G + a$ ou $G + xy$ le graphe $(S, A \cup \{a\})$;
- $G - a$ ou $G - xy$ le graphe $(S, A \setminus \{a\})$.

Remarques :

- Ces notations sont relativement standard, mais il est conseillé de les redéfinir explicitement si vous souhaitez les utiliser.
- On rappelle qu'une **paire** est un ensemble de cardinal deux. x et y sont donc supposés distincts.

Proposition 2

Soient $G = (S, A)$ et x, y deux sommets distincts et non adjacents de G .

- Si x et y sont dans la même composante connexe de G , alors $G + xy$ a le même nombre de composantes connexes que G et possède un cycle passant par l'arête xy .
- Sinon, $G + xy$ a une composante connexe de moins que G et ne possède pas de cycle passant par l'arête xy .

Démonstration. Se référer au cours de première année. \square

Soient $G = (S, A)$ un graphe et $xy \in A$.

- Si G contient un cycle passant par xy , alors $G - xy$ a le même nombre de composantes connexes que G .
- Sinon, $G - xy$ possède une composante connexe de plus que G : x et y ne sont plus dans la même composante connexe.

Démonstration. Corollaire immédiat de la propriété précédente. \square

Caractérisations des arbres :

Pour un graphe $G = (S, A)$ avec $|S| = n \geq 1$, les propositions suivantes sont équivalentes :

1. G est un arbre ;
2. G est sans cycle et $|A| = n - 1$;
3. G est connexe et $|A| = n - 1$;
4. G est minimalement connexe (on ne peut enlever d'arête à G en préservant la connexité) ;
5. G est maximalement sans cycle (on ne peut rajouter d'arête à G en préservant l'acyclicité) ;
6. pour tous sommets x, y de G , il existe un unique chemin élémentaire reliant x et y .

Démonstration. Vous trouverez cette preuve dans votre cours de première année ; cependant, il n'est pas inutile d'en redonner le principe (en dehors du dernier point qui est d'une nature un peu différente). Tout part de la remarque suivante : si $|A| < n$, un graphe à n sommets et $|A|$ arêtes possède au moins $n - |A|$ composantes connexes. Le cas d'égalité est celui où le graphe est acyclique. Cette remarque est une conséquence immédiate de la proposition 3. \square

I.2 - Graphes orientés, Forte Connexité

Un graphe orienté est dit **fortement connexe** si, pour tout couple de sommets (u, v) de G , il existe un chemin de u vers v .

La relation \mathcal{R} définie sur les sommets d'un graphe G par :

$$x\mathcal{R}y \Leftrightarrow (x \text{ accessible depuis } y \text{ et } y \text{ accessible depuis } x)$$

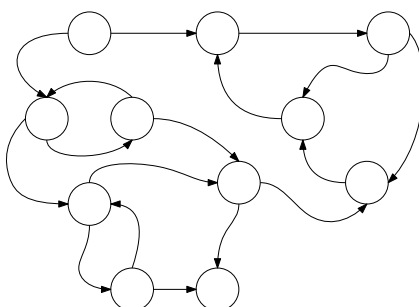
est une relation d'équivalence.

Les graphes induits par les classes d'équivalence de \mathcal{R} sont appelés **composantes fortement connexes** de G .

Remarques :

- La réflexivité, la symétrie et la transitivité de \mathcal{R} se vérifient aisément.
- Puisqu'une composante fortement connexe est un sous-graphe induit, elle est entièrement définie par son ensemble de sommets. On confondra donc parfois la composante connexe et son ensemble de sommets.
- Ainsi, les composantes fortement connexes d'un graphe forment une partition de S en S_1, \dots, S_k telle que x et y appartiennent au même S_i si et seulement si x est accessible depuis y et y depuis x .
- De manière similaire au cas non orienté, les composantes fortement connexes de G sont les sous-graphes induits maximalement fortement connexes de G .
- On utilisera parfois l'abréviation CFC pour **composante fortement connexe** (ou éventuellement SCC dans le code pour *Strongly Connected Component*).

Exercice/rappel : Entourer les composantes fortement connexes du graphe ci-dessous.



Cycle orienté :

Un cycle orienté de longueur n est une suite de $n + 1$ sommets x_0, \dots, x_n tels que :

- $n \geq 1$
- pour tout $i \in \llbracket 0; n - 1 \rrbracket$, (x_i, x_{i+1}) soit un arc du graphe ;
- $x_0 = x_n$.

Remarques :

- Par rapport aux cycles non orientés, on a enlevé l'exigence que le chemin soit élémentaire (sans répétition d'arcs). Cela ne pose pas de problème ici : ce qu'il fallait éviter dans le cas non orienté, c'était d'utiliser deux fois la même arête, une fois dans un sens et une fois dans l'autre.
- S'il existe un cycle orienté passant par x , alors on peut considérer le plus petit chemin non vide reliant x à x pour obtenir un cycle orienté élémentaire (sans répétition de sommet autre que les extrémités).
- $n = 1$ n'est possible qu'en présence de boucles, puisque le cycle est alors $x_0 \rightarrow x_0$. Comme dit plus haut, et sauf mention explicite du contraire, on supposera les graphes sans boucles.

On appelle **graphe orienté acyclique** (ou plus couramment **DAG** pour *Directed Acyclic Graph*) un graphe orienté ne possédant pas de cycle (orienté).

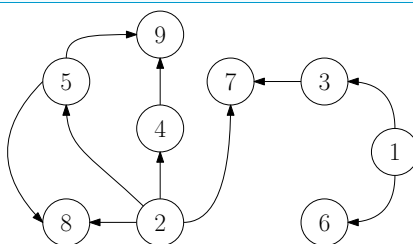


FIGURE 1 – Un exemple de DAG.

Dans un graphe orienté, on parle de :

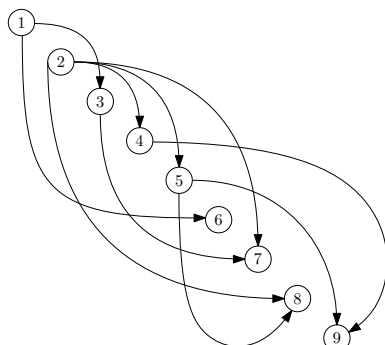
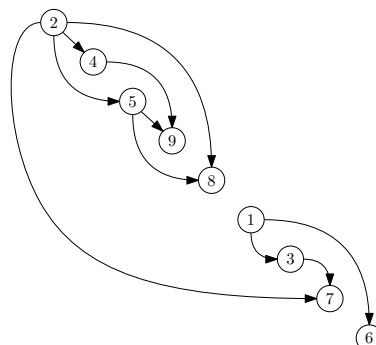
- **sommet source** (ou racine) pour un sommet de degré entrant nul ;
- **sommet puits** (ou feuille) pour un sommet de degré sortant nul.

Tout graphe orienté acyclique possède au moins un sommet source et au moins un sommet puits.

Démonstration. Soit n l'ordre du graphe. Si $x_1 \rightarrow \dots \rightarrow x_r$ est un chemin dans G avec $r > n$, alors par le principe des tiroirs il existe $i < j$ tels que $x_i = x_j$. Le graphe possède alors un cycle $x_i \rightarrow \dots \rightarrow x_j = x_i$, ce qui est absurde. La longueur des chemins de G est donc majorée (par $n - 1$), ce qui permet de considérer un chemin $x_1 \rightarrow \dots \rightarrow x_r$ avec r maximal.

- x_1 est nécessairement une racine, car s'il existait x_0 tel que $x_0 \rightarrow x_1$ alors $x_0 \rightarrow x_1 \rightarrow \dots \rightarrow x_r$ fournirait un chemin strictement plus long.
- De même, x_r est nécessairement une feuille. □

Un **ordre topologique** sur un graphe orienté G est un ordre total sur les sommets de G tel que, s'il y a un arc de u vers v , alors $u \prec v$.

(a) L'ordre $1 \prec 2 \prec 3 \prec \dots \prec 9$ est un ordre topologique sur le DAG de la figure 1.(b) L'ordre $2 \prec 4 \prec 5 \prec 9 \prec 8 \prec 1 \prec 3 \prec 7 \prec 6$ est un autre ordre topologique sur le même DAG.

Un graphe orienté admet un ordre topologique si et seulement si il est acyclique.

Démonstration. Se référer au cours de première année (exercice). □

Remarque :

- Dans un graphe acyclique (et seulement dans un graphe acyclique), la relation d'accessibilité est une relation d'ordre (partielle). Un ordre topologique est donc un ordre total qui étend cet ordre partiel.

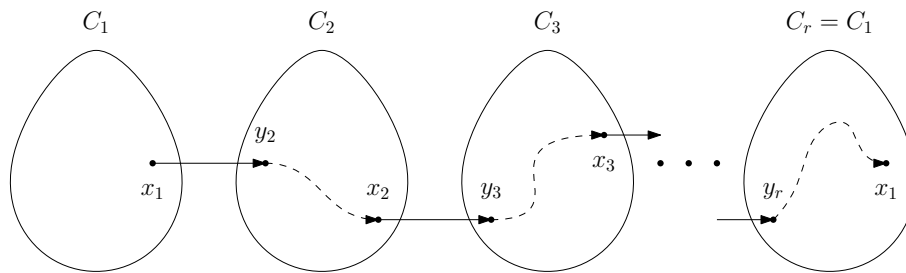
Graphe des CFC :

À un graphe orienté $G = (S, A)$, on peut associer le graphe $G_{CFC} = (S_{CFC}, A_{CFC})$ de ses composantes fortement connexes, défini par :

- S_{CFC} est l'ensemble $\{C_1, \dots, C_k\}$ des CFC de G ;
- G_{CFC} contient un arc $C_i C_j$ si et seulement si il existe $x \in C_i$ et $y \in C_j$ tels que $(x, y) \in A$.

Pour tout graphe orienté G , le graphe G_{CFC} est acyclique.

Démonstration. Supposons qu'on ait un cycle $C_1 \rightarrow C_2 \rightarrow \dots \rightarrow C_r = C_1$. Pour $1 \leq i < r$, il existe alors $(x_i, y_{i+1}) \in C_i \times C_{i+1}$ tels que $x_i \rightarrow y_{i+1}$. De plus, comme x_{i+1} et y_{i+1} sont dans la même CFC C_{i+1} , il existe un chemin $y_{i+1} \xrightarrow{*} x_{i+1}$ pour chaque $i \in \{1, \dots, r-1\}$. On obtient donc $x_1 \rightarrow y_2 \xrightarrow{*} x_2 \rightarrow y_3 \xrightarrow{*} \dots \rightarrow y_r$. Comme de plus $C_1 = C_r$, on peut prolonger par un chemin de y_r à x_1 : ainsi, on a $x_1 \xrightarrow{*} x_2 \rightarrow x_1$. Cela montre que x_1 et x_2 sont dans la même CFC, ce qui est absurde. Donc G' est bien acyclique.



□

I.3 - Parcours en profondeur et tri topologique

I.3.i - Parcours en profondeur récursif

Rappelons le code d'un parcours en profondeur récursif simple (à savoir écrire sans difficulté). Plutôt qu'un tableau `deja_vu`, on reprend la distinction plus fine Vierge / Ouvert / Fermé¹ sur les sommets, et on stocke ces états dans un tableau `statut` :

- **Vierge** : Le sommet n'a pas encore été vu.
- **Ouvert** : L'exploration récursive a été lancée depuis le sommet, mais n'a pas encore terminé (on explore toujours voisins / descendants).
- **Fermé** : L'exploration récursive a été lancée depuis le sommet, et s'est terminée (on a fini d'explorer ses descendants).

```

1 Entrées : G un graphe, s un sommet de G,
2 statut un tableau d'états du sommet (initialisé à "Vierge").
3 Effet : parcours en profondeur des sommets accessibles depuis s dans G.
4
5 PARCOURS(G, s, statut) :
6   statut[s] ← Ouvert
7   Visiter(s) //action variée, par exemple un affichage
8   Pour chaque voisin v de s dans G Faire :
9     Si statut[v] = Vierge Alors :
10      PARCOURS(G, v, statut)
11   statut[s] ← Fermé //on a (enfin) fini de traiter le sommet s
12 //Remarque : statut peut aussi être une variable globale (petit gain mémoire)

```

1. elle n'est pas nécessaire ici, mais elle sera très utile dans ce chapitre

Si le graphe est non orienté, on explore ainsi la composante connexe de s . Pour parcourir le graphe en entier, composante connexe par composante connexe, il suffit de lancer un parcours depuis chaque sommet si celui-ci n'a pas déjà été exploré :

Pseudo-code

```
1 Pour  $s \in S$  Faire :
2   Si statut[s] = Vierge alors
3     PARCOURS( $s$ )
```

Tri topologique

Pour trouver un ordre topologique sur le graphe, on utilise le théorème suivant :

Théorème 2

Soit $G = (S, A)$ un DAG. Ordonner les sommets de S par **date de fin de parcours décroissante** dans un parcours en profondeur récursif complet définit un ordre topologique du graphe.

Démonstration. Se référer au cours de première année. □

Exemple : Si on lance les parcours récursifs sur les sommets par ordre croissant d'étiquette, pour la *figure 1* (DAG), un exemple possible d'ordre de fins d'exploration des sommets est le suivant : 6,7,3,1,9,8,5,4,2.

Par exemple, comme on commence par lancer le DFS sur 1, on peut ensuite explorer son voisin 6, qui n'a pas de voisin, ainsi on finit d'explorer 6 avant de revenir sur 1 et de continuer à explorer ses autres voisins (3, etc.).

Donc on obtient le tri topologique suivant : 2, 4, 5, 8, 9, 1, 3, 7, 6.

Implémentation :

On peut traduire directement le théorème précédent en code : on lance des parcours en profondeur (DFS) successifs, en remplissant le tri selon l'ordre des dates de fin de parcours ("Fermé") trouvées.

Pseudo-code

```
1 Entrées :  $G = (S, A)$  un graphe
2 Effet : un ordre topologique des sommets de  $G$ 
3
4 TRI_TOPOLOGIQUE( $G$ ) :
5   // Initialisation :
6   statut ← [ Vierge; Vierge; ...; Vierge ] //autant de cases que de sommets dans  $S$ .
7   ordre_topo ← []
8
9   // Parcours en profondeur récursif pour le tri :
10  DFS( $s$ ) :
11    statut[s] ← Ouvert
12    Pour chaque voisin  $v$  de  $s$  Faire :
13      Si statut[v] = Vierge Alors :
14        DFS( $v$ )
15      Si couleur[v] = Ouvert Alors :
16        Renvoyer une erreur "Cycle détecté" (arrêter le programme)
17    statut[s] ← Fermé
18    Ajouter  $s$  au début de ordre_topo //dates de fin décroissantes
19
20  // Parcours du graphe entier (avec un ou plusieurs DFS) :
21  Pour chaque  $s$  dans  $S$  :
22    Si couleur[s] = Blanc Alors : //le sommet n'a pas encore été vu
23      DFS( $s$ )
24
25  Renvoyer ordre_topo
```

I.4 - Algorithme de Dijkstra

Définition 11

Distance dans un graphe pondéré :

Si G est un graphe pondéré ne possédant pas de cycle de poids strictement négatif, on définit la **distance** de x à y par :

$$d_G(x, y) = \inf\{\rho(c) \mid c \text{ chemin de } x \text{ à } y\},$$

avec la convention que $\inf \emptyset = +\infty$.

Un **plus court chemin** de x à y est un chemin c de x à y vérifiant $\rho(c) = d(x, y)$.

Remarques :

- Cette définition a bien un sens, puisque la condition sur l'absence de cycle de poids négatif permet de se limiter aux chemins élémentaires (à partir d'un chemin quelconque de x à y de poids p , on peut toujours obtenir un chemin élémentaire de x à y de poids $p' \leq p$ en supprimant les cycles). Les chemins élémentaires étant en nombre fini (ils possèdent au plus $n - 1$ arêtes), on a en fait $d_G(x, y) = \min\{\rho(c) \mid c \text{ chemin élémentaire de } x \text{ à } y\}$.
- On peut ici avoir une distance négative, puisqu'on n'exige pas que les arcs soient à poids positif.

Proposition 7

Propriétés de la distance dans un graphe pondéré :

Soit $G = (S, A, \rho)$ un graphe pondéré sans cycle de poids strictement négatif.

- d_G vérifie l'inégalité triangulaire : $d_G(x, z) \leq d_G(x, y) + d_G(y, z)$.
- Si G est à poids strictement positif, alors $d_G(x, y) \geq 0$ et $d_G(x, y) = 0$ équivaut à $x = y$.
- Si G est non orienté, alors d_G est symétrique : $d_G(x, y) = d_G(y, x)$.
- Si G est non orienté, connexe, à poids strictement positifs, alors d_G est une distance au sens mathématique du terme.

L'algorithme de Dijkstra est fondamentalement une adaptation du parcours en largeur au cas des graphes pondérés, obtenue en remplaçant la file du BFS par une file de priorité. On visite les sommets du graphe par ordre croissant de distance à la source.

Pseudo-code générique :

Voici un pseudo-code général de l'algorithme de Dijkstra, que vous avez vu en première année :

Pseudo-code

```
1  Entrées :  $G = (S, A, \rho)$  un graphe pondéré avec  $\rho(A) \subseteq \mathbb{R}^+$ ,  $s \in S$  un sommet (source) de  $G$ .
2  Sortie : tableau  $dist$  des distances à  $s$  :  $dist[t] = d(s, t)$  pour tout sommet  $t \in S$ 
3  DIJKSTRA( $G, s$ ):
4       $dist[t] \leftarrow +\infty$  pour tout  $t \in S$ 
5       $dist[s] \leftarrow 0$ 
6       $F \leftarrow \{s\}$ 
7      TantQue  $F \neq \emptyset$  Faire :
8           $u \leftarrow$  Extraire de  $F$  le sommet dont la valeur dans  $dist$  est minimale
9          Pour  $v$  voisin de  $u$  Faire :
10             Si  $dist[v] = +\infty$  Alors : //condition équivalente à "v n'a jamais été vu"
11                 Insérer  $v$  dans  $F$ 
12             Si  $dist[u] + \rho(u, v) < dist[v]$  Alors :
13                  $dist[v] \leftarrow dist[u] + \rho(u, v)$  //mise à jour de la priorité dans  $F$ , donc
14  Renvoyer  $dist$ 
```

Proposition 8

Propriétés de l'algorithme de Dijkstra :

Si $G = (S, A, \rho)$ est un graphe pondéré à poids positifs, l'algorithme de Dijkstra :

- extrait les sommets de la file par distance croissante au sommet initial s .
- renvoie un tableau $dist$ vérifiant $dist[v] = d_G(s, v)$ pour tout sommet v .
- a une complexité en espace en $(|S|)$.
- effectue au plus $|S|$ ajouts dans F , $|S|$ extractions de minimum et $|A|$ mises à jour de distance (diminution de priorité).
- peut être implémenté avec une complexité $O((|E| + |V|) \log |V|)$ en réalisant la file de priorité F par un tas binaire.

Remarque :

- Il est possible d'implémenter l'algorithme de Dijkstra en utilisant une file-min de priorité « basique » dont les seules opérations fournies sont la création de file de priorité vide, l'insertion avec une priorité, et l'extraction du minimum (sans opération de mise à jour des priorités dans le tas).

Dans cette variante de l'algorithme, on ré-insère le sommet avec une priorité plus basse plutôt que de diminuer sa priorité dans la file. On extraira donc plusieurs fois un même sommet, mais on ignorera toutes ces extractions sauf la première (il faut tester que le sommet n'a pas déjà été traité quand on l'extrait). La complexité spatiale passe en $O(|S| + |A|)$, mais ce n'est que très rarement gênant.

Construction du plus court chemin :

Pour une version avec calcul du plus court chemin, il faut retenir en plus quelle arête $u \rightarrow v$ a provoqué la première exploration de v . Dans la file de priorité, on peut stocker un couple (v, u) pour retenir le parent au lieu de simplement v . On peut aussi faire les mises à jour de distance et de parent au fur et à mesure. Au moment où on extrait un sommet u de la file, on sait que sa distance est optimale et le sommet a été traité.

Les structures de données utilisées dans l'algorithme de Dijkstra décrit ci-dessous sont les suivantes :

- tableau **dist** : indique pour chaque sommet u le poids du plus court chemin trouvé jusqu'à présent de s à u .
- file de priorité **ouverts** : contient les sommets u à explorer, avec comme priorité le poids du plus court chemin entre s et u trouvé jusqu'à maintenant (**dist**[u]).
- tableau **parents** : code un arbre permettant de reconstruire un chemin depuis la source jusqu'à chaque sommet exploré.

Pseudo-code

```
1  Entrées :  $G = (S, A, \rho)$  un graphe pondéré avec  $\rho(A) \subseteq \mathbb{R}^+$ ,  $s \in S$  un sommet (source) de  $G$ .
2  Sortie : tableau parents de l'arbre des plus courts chemins depuis  $s$ 
3  DIJKSTRA( $G, s$ ):
4      dist  $\leftarrow [\infty, \dots, \infty]$ 
5      dist[s]  $\leftarrow 0$ 
6      parents  $\leftarrow (\text{Nil}, \text{Nil}, \dots \text{Nil})$ 
7      parents[s]  $\leftarrow s$ 
8      ouverts  $\leftarrow \text{FILEPRIOVIDE}()$ 
9      TantQue  $F \neq \emptyset$  Faire :
10          $u \leftarrow$  Extraire de  $F$  le sommet dont la valeur dans dist est minimale
11         Pour chaque  $v$  successeur de  $u$  Faire :
12              $d \leftarrow \text{dist}[u] + \rho(u \rightarrow v)$ 
13             Si  $d < \text{dist}[v]$  Alors :
14                 Si  $\text{dist}[v] = +\infty$  Alors : //condition équivalente à "v n'a jamais été vu"
15                     Insérer  $v$  dans  $F$ 
16                     parents[v]  $\leftarrow u$ 
17                     dist[v]  $\leftarrow d$  //mise à jour de la priorité dans  $F$ , donc
18 Renvoyer dist
```

Remarque :

- Il existe des réalisations de la structure de file de priorité dotées de meilleurs complexités (amorties) que les tas binaires pour certaines opérations comme la diminution des priorités. C'est le cas, par exemple, des tas de Fibonacci. On peut ainsi obtenir une complexité totale pour Dijkstra en $(|A| + |S| \cdot \log |S|)$.

Notons toutefois que cette amélioration est surtout théorique : un tas de Fibonacci est bien plus compliqué à implémenter qu'un tas binaire, et sera plus lent en pratique dans presque tous les cas.

II - Arbre couvrant minimal

II.1 - Arbre couvrant

Définition 12

Arbre couvrant :

Exemple :

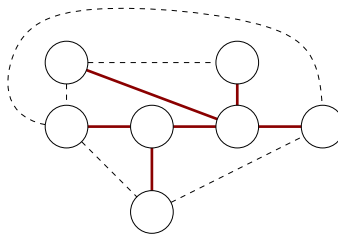


FIGURE 3 – Un graphe connexe et l'un de ses arbres couvrants.

Remarques :

- Un arbre couvrant est donc entièrement déterminé par son ensemble A_H d'arêtes : on s'autorisera donc parfois à confondre H et A_H .
- Un graphe non connexe ne peut évidemment pas posséder d'arbre couvrant. On appellera **forêt couvrante** une collection d'arbres couvrants (un par composante connexe).

Théorème 3

Existence d'un arbre couvrant :

Démonstration. Il y a deux démonstrations naturelles, toutes deux constructives, qui correspondent à deux algorithmes de construction d'un arbre couvrant (on note $C_T(x)$ la composante connexe de x dans T).

Algorithme 1 Construction d'un arbre couvrant par suppression d'arêtes

Entrées : Un graphe connexe $G = (S, A)$.

Sorties : Un arbre couvrant T de G .

```

1  $T \leftarrow (S, A)$ 
2 pour chaque  $xy \in A$  faire
3   | si  $T - xy$  est connexe alors
4   |   |  $T \leftarrow T - xy$ 
5 renvoyer  $T$ 
```

Algorithme 2 Construction d'un arbre couvrant par ajout d'arêtes

Entrées : Un graphe connexe $G = (S, A)$.

Sorties : Un arbre couvrant T de G .

```

1  $T \leftarrow (S, \emptyset)$ 
2 pour chaque  $xy \in A$  faire
3   | si  $C_T(x) \neq C_T(y)$  //composantes connexes alors
4   |   |  $T \leftarrow T + xy$ 
5 renvoyer  $T$ 
```

Prouvons maintenant (*sur feuille!*) que ces algorithmes sont totalement corrects.

□

II.2 - Arbre couvrant minimal

On considère à présent des graphes non orientés et pondérés. Un tel graphe sera noté $G = (S, A, \rho)$, où $\rho : A \rightarrow \mathbb{R}$ associe à chaque arête son *poids*. On s'autorise des poids négatifs, et l'on suppose G connexe.

Définition 13

Poids d'un sous-graphe :

Si H est un sous-graphe de G , le poids de H , noté $\rho(H)$, est :

Définition 14

Arbre couvrant minimal (ACM) :

Soit $H = (S_H, A_H)$ un sous-graphe de G . H est un **arbre couvrant minimal** de G si H est un arbre couvrant de G et que, de plus :

Exemple :

L'arbre couvrant minimal est utile pour résoudre le problème très commun suivant. On considère un ensemble de n points A_1, \dots, A_n du plan, et l'on souhaite relier tous ces points entre eux à l'aide de segments $[A_i A_j]$, en minimisant la somme des longueurs des segments.

Par exemple, on veut construire des ponts entre les îles d'un archipel en minimisant la quantité de matériaux à utiliser, de sorte que toutes les îles soient reliées. Ou encore poser des câbles sur un réseau électrique.

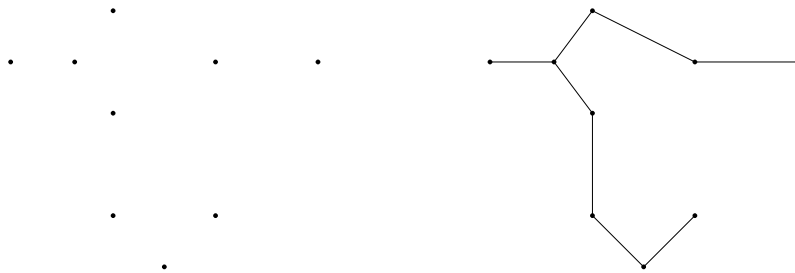


FIGURE 4 – Une instance et une solution possible (qui doit être optimale).

Remarques :

- L'ensemble des arbres couvrants de G est un ensemble non vide (puisque G est connexe) et fini, ce qui garantit l'existence d'un arbre couvrant minimal.
- ⚠ Il n'y a en revanche aucune raison que cet arbre couvrant minimal soit unique.

Proposition 9

Soit $X \subset S$ avec $1 \leq |X| \leq n - 1$ et E_X l'ensemble des arêtes $\{x, y\}$ avec $x \in X$ et $y \in S \setminus X$.

- E_X est non-vide.
- Si T est un arbre couvrant minimal de G , alors T contient une arête de E_X de poids minimal.

Démonstration. Sur papier !

□

II.3 - Algorithme de Kruskal

L'algorithme de Kruskal est un algorithme glouton très simple :

Algorithme 3 Algorithme de Kruskal

Entrées : Un graphe pondéré $G = (S, A, \rho)$ connexe, et une liste $A = (a_1, \dots, a_p)$ de ses arêtes.

Sorties : Un arbre couvrant minimal de G

```

1  $T \leftarrow (S, \emptyset)$ 
2 pour chaque  $xy \in A$ , dans l'ordre croissant de poids faire
3   | si  $x$  et  $y$  sont dans deux composantes connexes distinctes de  $T$  alors
4   |   |  $T \leftarrow T + xy$ 
5 renvoyer  $T$ 
```

Proposition 10

Correction de l'algorithme de Kruskal :

Si G est connexe, l'algorithme de Kruskal renvoie bien un arbre couvrant minimal de G .

Démonstration. Sur papier !

□

Complexité

Dans toute cette section, on note $n = |S|$ et $p = |A|$. $G = (S, A, \rho)$ est un graphe pondéré à n sommets et p arêtes.

- Si la liste des arêtes A de G n'est pas donnée triée, il faut commencer par trier les arêtes de G en temps (pour un tri par comparaison) :

Version naïve :

On peut représenter naïvement l'arbre $T = (S, A_T)$ (sous-graphe de $G = (S, A)$) par listes d'adjacence. On obtient alors :

- Complexité du test : On teste si deux sommets x et y sont dans la même composante connexe par un parcours de graphe. La complexité d'un parcours de graphe représenté par listes d'adjacence est linéaire en la taille du graphe :

². Ici, on peut simplifier un peu cette expression. Comme T est acyclique, on sait que

$|A_T| \leq n - 1$. La complexité de ce test est donc en $O(n)$ pire cas.

- Complexité de l'algorithme de Kruskal naïf : La boucle parcourt chaque arête de G , donc effectue un nombre $p = |A|$ d'itérations, et chaque itération est en temps $O(n)$. On obtient une complexité $O(np)$ au total, qui domine la complexité du tri initial des arêtes ($O(p \log p)$) (si on doit le faire).

Remarque :

- Ici on sait qu'il y aura moins de n arêtes dans T (il y en aura $n - 1$ au total en fin d'algorithme), donc la représentation par listes d'adjacence est bien meilleure que la représentation par matrice d'adjacence. Chaque sommet a en moyenne au plus 2 voisins, les listes d'adjacence contiennent donc en moyenne au plus deux éléments.

Structure Union-Find :

On a vu dans le chapitre précédent une structure de données spécifiquement adaptée à la gestion de partitions qui deviennent de plus en plus grossières (ici, les composantes connexes de T). Utilisons la !

On peut alors ré-écrire le pseudo-code de l'algorithme de Kruskal de la manière suivante, en utilisant les opérations définies sur la structure d'*union-find* :

Algorithme 3 Algorithme de Kruskal avec structure UnionFind

Entrées : Un graphe pondéré $G = (S, A, \rho)$ connexe, et une liste $A = (a_1, \dots, a_p)$ de ses arêtes.

Sorties : Un arbre couvrant minimal de G .

```
1  $T \leftarrow \{\}$ 
2  $X \leftarrow \text{CREATEPARTITION}(n)$ 
3 pour chaque  $xy \in A$ , dans l'ordre croissant de poids faire
4   si  $\text{FIND}(X, x) \neq \text{FIND}(X, y)$  alors
5      $\text{UNION}(X, x, y)$ 
6      $T \leftarrow T \cup \{xy\}$ 
7 renvoyer  $T$  //on confond un arbre couvrant et ses arêtes
```

Proposition 11

Complexité de l'algorithme de Kruskal :

Pour un graphe G à n sommets et p arêtes, l'algorithme de Kruskal a une complexité temporelle dans le pire cas de :

- $O(p \log n)$ si les arêtes doivent être triées (par comparaison) au départ ;
- $O(p \log^* n)$ si les arêtes sont déjà triées (ou s'il est possible de les trier en temps linéaire).

Démonstration. Sur papier !

□

2. Assurez-vous de savoir le démontrer !

III - Composantes fortement connexes

Dans un graphe *orienté*, déterminer les composantes connexes se fait par un simple parcours de graphe, en temps linéaire en la taille du graphe (i.e. $|S| + |A|$). L'étude des composantes fortement connexes d'un graphe *non orienté* est plus subtile, mais peut aussi se faire en temps linéaire. Nous allons l'étudier dans ce chapitre.

Dans toute cette partie, on s'intéresse à des graphes orientés et non pondérés. On considère des graphes sans boucles (sans arc de la forme (x, x)), mais la présence de boucles ne modifierait pas fondamentalement les résultats.

III.1 - Compléments sur le parcours en profondeur

Vous avez déjà vu en première année plusieurs utilisations du parcours en profondeur récursif et du marquage Vierge / Ouvert / Fermé. L'une d'entre elle en particulier utilise les dates de fin de parcours : le tri topologique. De manière plus générale, il peut être utile de marquer et de retenir les **dates de début et de fin de parcours de chaque sommet**, à d'autres fins.

L'algorithme suivant effectue un parcours en profondeur du graphe dans l'ordre croissant des étiquettes et renvoie les temps de début et de fin de parcours obtenus pour chaque sommet, dans deux tableaux pre et post respectivement, indexés par les sommets S .

Pseudo-code

```

1  Entrées :  $G = (S, A)$  un graphe orienté
2  Sortie : Deux tableaux pre et post indexés par  $S$ , une partition des arcs de  $G$ 
3
4  PARCOURS_PROF_AVEC_STOCKAGE_DES_TEMPS( $G$ ) :
5      pre  $\leftarrow$  [0, . . . , 0]
6      post  $\leftarrow$  [0, . . . , 0]
7      t  $\leftarrow$  1 //t l'instant actuel, initialisé à 1, incrémenté à chaque marquage
8
9      //parcours à partir d'un sommet :
10     fonction EXPLORER( $u$ ) :
11         pre[ $u$ ]  $\leftarrow$  t
12         t  $\leftarrow$  t+1
13         Pour tout  $v \in \text{successeurs}(u)$  Faire :
14             Si pre[ $v$ ] = 0 Alors : //(u, v) est un arc "arbre"
15                 EXPLORER( $v$ )
16             //Sinon on ne lance pas d'exploration depuis v (déjà vu).
17             // et (u,v) est alors un arc "avant", "latéral" ou "arrière".
18         post[ $u$ ]  $\leftarrow$  t
19         t  $\leftarrow$  t+1
20
21     // boucle principale : parcours de tout le graphe
22     Pour chaque  $s \in S$  faire
23         Si pre[ $s$ ] = 0 Alors :
24             EXPLORER( $s$ )

```

Exercice :

Sur le graphe G_1 de la figure 5, donner les dates d'ouverture et de fermeture de chaque sommet, sous la forme des tableaux pre et post. Quel lien y a-t-il entre les valeurs de pre et post ?

	0	1	2	3	4	5	6	7	8	9	10	11
pre :												
post :												

Rappel (accessibilité et ordre de traitement) :

Si y est ouvert pendant l'exploration de x , c'est-à-dire si $\text{pre}(x) < \text{pre}(y) < \text{post}(x)$, alors :

- y est accessible depuis x ;
- $\text{pre}(x) < \text{pre}(y) < \text{post}(y) < \text{post}(x)$.

De même, si y est fermé pendant l'exploration de x , c'est-à-dire si $\text{pre}(x) < \text{post}(y) < \text{post}(x)$, alors :

- y est accessible depuis x ;
- $\text{pre}(x) < \text{pre}(y) < \text{post}(y) < \text{post}(x)$.

Remarque :

- Autrement dit, les instants **pre** et **post** respectent une propriété de bon parenthésage : en notant x l'instant d'ouverture d'un sommet x et X son instant de fermeture, les ordres autorisés sont :

$$xyYX \quad yxXY \quad xXyY \quad yYxX$$

Les ordres $xyXY$ et $yxYX$, qui correspondent à des "parenthèses enchevêtrées", sont interdits (comme bien sûr les ordres dans lesquels x est fermé avant d'être ouvert).

Exercice :

Les propositions suivantes sont-elles correctes ? Pour chacune d'entre elle, donner une démonstration ou un contre-exemple.

1. Si y est accessible depuis x , alors y sera ouvert pendant l'exploration de x .
2. Si y est accessible depuis x et si $\text{pre}(x) < \text{pre}(y)$, alors $\text{pre}(y) < \text{post}(x)$.
3. S'il y a un arc xy et si $\text{pre}(x) < \text{pre}(y)$, alors $\text{pre}(y) < \text{post}(x)$.

À faire sur feuille !

Classification des arcs :

Un parcours en profondeur récursif d'un graphe $G = (S, A)$ donne une classification des arêtes de G en quatre catégories. Un arc (u, v) est dit :

- **arc arbre** si l'exploration de u a directement entraîné l'exploration de v voisin de u (cf ligne 14 de PARCOURS_PROF_AVEC_STOCKAGE_DES_TEMPS).

Les arcs arbre définissent une **forêt**ⁱ F (dont l'ensemble des sommets est S).

- **arc avant** si v est un descendant de u dans cette forêt F .
- **arc arrière** si u est un descendant de v dans F .
- **arc latéral** si les (sous-)arbres enracinés en u et en v sont disjoints dans F .

i. On peut montrer que les *arcs arbres* forment un sous-graphe acyclique : c'est un invariant de l'algorithme. Lorsqu'on ajoute un arc arbre, c'est vers un sommet qui n'avait encore jamais été vu, donc sans arête adjacente dans le sous-graphe. Ce nouvel arc ne forme donc pas un cycle.

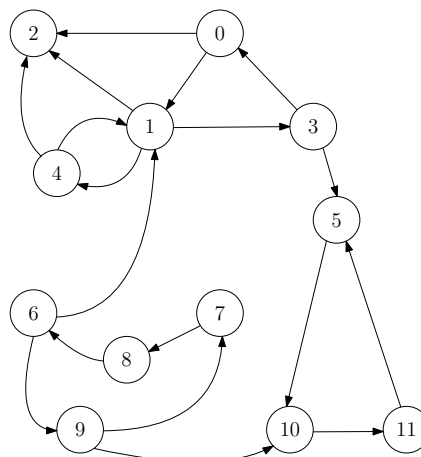
Remarque :

- Un graphe est acyclique si et seulement si son parcours en profondeur ne produit pas d'arc arrière. C'est ce que teste l'algorithme de détection de cycle / tri topologique rappelé en première partie de ce cours. (Il n'y a pas besoin de **pre** et **post** pour ce test.)

Exemple :

Avec l'algorithme PARCOURS_PROF_AVEC_STOCKAGE_DES_TEMPS sur le graphe G_1 ci-dessous, on obtient les classifications des arêtes suivantes :

FIGURE 5 – Le graphe G_1 .



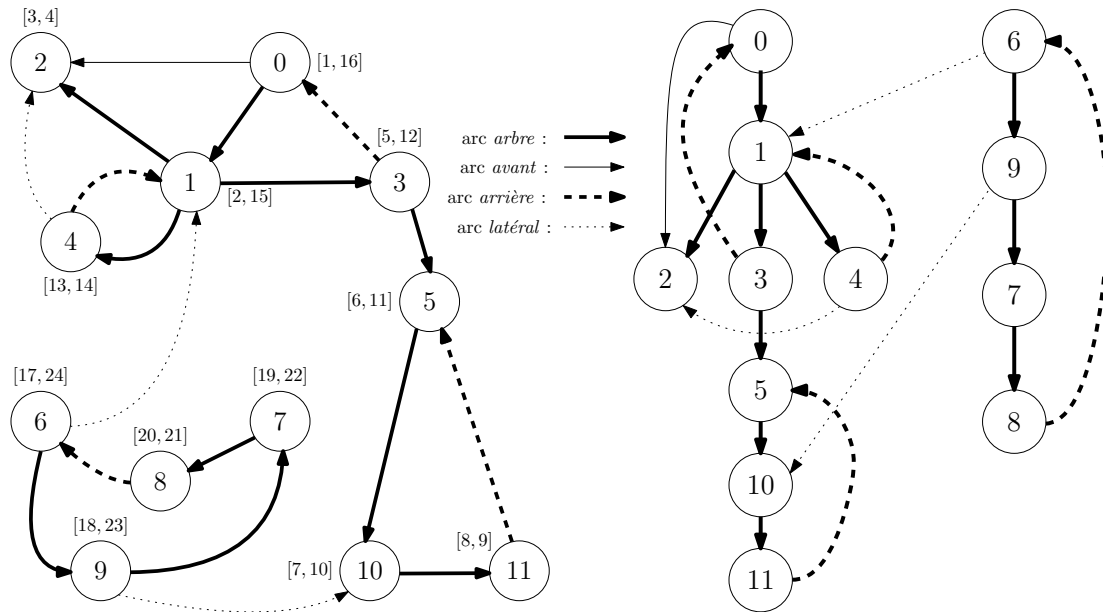
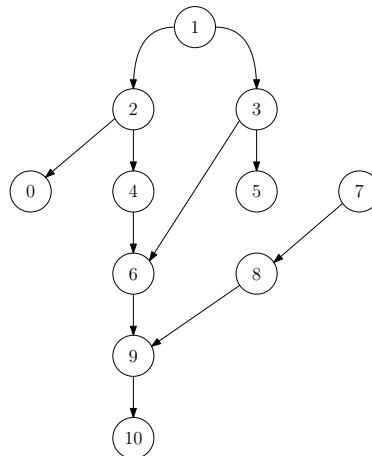


FIGURE 6 – Exemple de classification des arcs de G_1 , avec la forêt correspondante. Pour chaque nœud v , on a indiqué l'intervalle $[pre(v), post(v)]$.

Exercice :

1. Les temps **pre** et **post** et la classification des arcs obtenus à la figure 6 correspondent à un traitement des nœuds dans l'ordre croissant des étiquettes, autant dans la boucle principale que pour chaque parcours d'une liste de successeurs. Simuler l'exécution de l'algorithme et faire la figure correspondante si le traitement se fait dans l'ordre décroissant des étiquettes.
2. Appliquer l'algorithme PARCOURS_PROF_AVEC_STOCKAGE_DES_TEMPS au graphe G_0 suivant, en traitant les sommets par étiquettes croissantes, et classifier ses arêtes :

FIGURE 7 – Le graphe G_0



À faire sur feuille !

On peut classifier les arêtes algorithmiquement, pendant le parcours en profondeur lui-même. Il suffit de modifier légèrement le programme PARCOURS_PROF_AVEC_STOCKAGE_DES_TEMPS de la manière suivante :

Pseudo-code

```

1 Entrées :  $G = (S, A)$  un graphe orienté
2 Sortie : Deux tableaux pre et post indexés par  $S$ , une partition des arcs de  $G$ 
3
4 CLASSIFICATION( $G$ ) :
5     actifs  $\leftarrow \emptyset$  //contiendra la branche de l'arbre explorée actuellement
6     pre  $\leftarrow [0, \dots, 0]$ 
7     post  $\leftarrow [0, \dots, 0]$ 
8     t  $\leftarrow 1$ 

```

```

9
10 fonction EXPLORER(u) :
11     pre[u] ← t
12     t ← t+1
13     actifs ← actifs ∪ {u}
14     Pour tout v ∈ successeurs(u) Faire :
15         Si pre[v] = 0 Alors :
16             (u, v) est un arc arbre
17             EXPLORER(v)
18         // Sinon on distingue les arcs avant, arrières et latéraux :
19         Sinon si pre(v) > pre(u) Alors : //v est un descendant de u
20             (u, v) est un arc avant
21         Sinon si v ∉ actifs Alors : //u n'est pas un descendant de v,
22             //ils ne sont pas dans la même branche (v a été enlevé de "actifs")
23             (u, v) est un arc latéral
24         Sinon :
25             (u, v) est un arc arrière
26     post[u] ← t
27     t ← t+1
28     actifs ← actifs \ {u}
29
30 Pour chaque s ∈ S faire
31     Si pre[s] = 0 Alors :
32         EXPLORER(s)

```

Quelle est la complexité de cet algorithme (et du précédent)?

III.2 - Algorithme de Kosaraju

Introduction au problème

Lors d'un parcours (quelconque) depuis un sommet x , on explore tous les sommets accessibles depuis x . Dans un graphe non orienté, il s'agit exactement de la composante connexe de x . En revanche, dans le cas d'un graphe orienté, la composante fortement connexe de x est incluse dans cet ensemble, mais l'inclusion est stricte en général. C'est cela qui rend le calcul des composantes fortement connexes plus délicat que celui des composantes connexes dans un graphe non orienté.

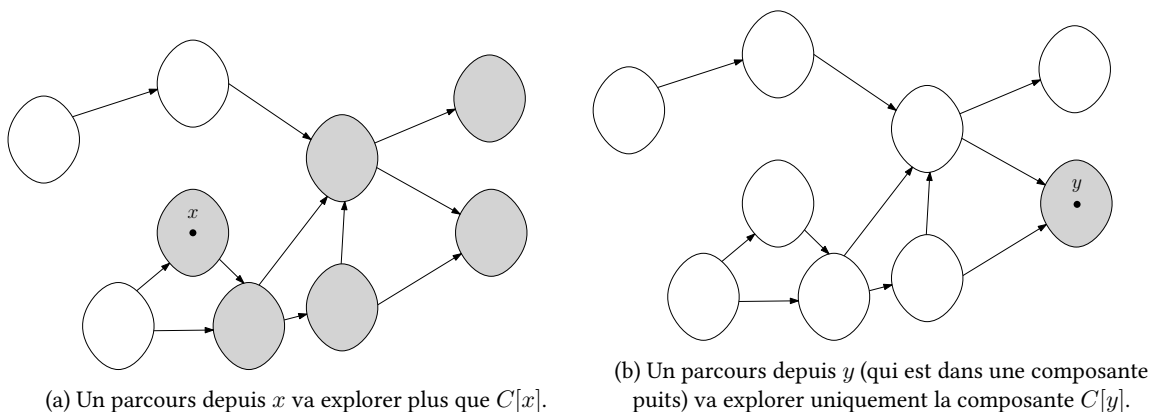


FIGURE 8 – Le graphe G_{CFC} pour un certain graphe orienté G .

Cependant, le graphe G_{CFC} ³ des composantes fortement connexes de G est acyclique, et contient donc au moins un sommet puits (propositions 4 et 6). Si la composante fortement connexe $C[x]$ de x est un puits, cela signifie qu'il n'y a aucun arc de $C[x]$ vers une autre CFC, et donc que les sommets accessibles depuis x sont exactement ceux de $C[x]$.

3. NB : la présence d'un arc d'une composante C vers une composante C' indique qu'il existe au moins un arc $x \rightarrow x'$ avec $x \in C$ et $x' \in C'$ (mais il peut en exister plusieurs).

⚠ **Attention!** Cette section est à copier sur feuille depuis le tableau. Ceci n'est qu'une annexe de la section. ⚠

Proposition 13

Soit x le premier sommet d'une CFC C à être ouvert (i.e. $\mathbf{pre}(x) = \mathbf{pre}(C)$). On a :

$$\mathbf{post}(x) = \mathbf{post}(C)$$

Démonstration. Les sommets y vérifiant $\mathbf{pre}(x) < \mathbf{post}(y) < \mathbf{post}(x)$ sont ceux découverts pendant l'exploration de y . Or ces sommets sont exactement ceux qui sont accessibles depuis x dans le graphe dans lequel on a supprimé les sommets déjà vus (i.e. les z tels que $\mathbf{pre}(z) < \mathbf{pre}(x)$). Comme tous les sommets de C sont accessibles depuis x sans sortir de C , et comme aucun sommet de C n'est déjà vu au moment de l'ouverture de x , on explorera tous les sommets de C (plus éventuellement d'autres sommets). Donc $\mathbf{post}(x) = \mathbf{post}(C)$. \square

Proposition 14

Notons G^{\leftarrow} le graphe miroir de G . On a $(G^{\leftarrow})_{CFC} = (G_{CFC})^{\leftarrow}$.

En particulier, les classes sources de G_{CFC} sont des classes puits dans $(G^{\leftarrow})_{CFC}$, et inversement.

Démonstration. Il suffit de déplier les définitions :

Les composantes connexes de G et G^{\leftarrow} sont les mêmes. En effet, soient $x, y \in S$ deux sommets de G . Alors il existe un chemin de x vers y et un chemin de y vers x dans G si et seulement si il existe un chemin de y vers x et un chemin de x vers y dans G^{\leftarrow} . Donc les sommets de $(G^{\leftarrow})_{CFC}$ et $(G_{CFC})^{\leftarrow}$ sont les mêmes.

De plus, il existe un arc $C \rightarrow C'$ dans G_{CFC} si et seulement s'il existe un arc (x, y) avec $x \in C$ et $y \in C'$ dans G , c'est à dire de manière équivalente qu'il existe un arc (y, x) dans G^{\leftarrow} avec $y \in C'$ et $x \in C$, d'où $C' \rightarrow C$ dans $(G^{\leftarrow})_{CFC}$.

$$\text{Donc } (G^{\leftarrow})_{CFC} = (G_{CFC})^{\leftarrow}.$$

\square

Remarques :

- En particulier, les composantes fortement connexes de G^{\leftarrow} sont les mêmes que celles de G : on pourra donc parler de $C[x]$ sans préciser si l'on se place dans G ou G^{\leftarrow} .

Principe de l'algorithme de Kosaraju :

On commence par effectuer un parcours en profondeur (dans un ordre quelconque) de G pour obtenir la liste x_1, \dots, x_n des sommets dans l'ordre décroissant des dates de fin de parcours : $\mathbf{post}(x_1) > \mathbf{post}(x_2) > \dots > \mathbf{post}(x_n)$.

Ensuite, on construit G^{\leftarrow} et on procède ainsi :

- on effectue un parcours depuis x_1 dans G^{\leftarrow} en marquant les sommets rencontrés pour les éliminer des parcours ultérieurs
- comme la classe de x_1 est une classe puits dans G_{CFC}^{\leftarrow} , on obtient exactement la CFC de x_1 dans G^{\leftarrow} (qui est égale à la CFC de x_1 dans G);
- on continue ensuite à parcourir la liste des x_i , en sautant les sommets déjà vus;
- quand on tombe sur un sommet non marqué, on sait que sa CFC est un puits dans ce qui reste de G_{CFC}^{\leftarrow} (puisque son instant \mathbf{post} est maximal parmi les sommets non marqués), elle correspond donc exactement aux sommets (non marqués) accessibles depuis x_i ;
- on continue ainsi jusqu'à avoir marqué tous les sommets.

Pseudo-code détaillé :

Pseudo-code



```
1  Entrées :  $G = (S, A)$  un graphe orienté
2  Sortie : La liste  $C_1, \dots, C_k$  des composantes fortement connexes de  $G$ 
3  (chaque  $C_i$  étant une liste de sommets).
4
5  //parcours en profondeur récursif : énumération
6  //des sommets dans l'ordre décroissant des dates de fin de parcours.
7  DFS_POST( $G$ ) :
8       $L \leftarrow []$ 
9       $vus \leftarrow \emptyset$ 
10     fonction EXPLORER( $x$ ) :
11         Si  $x \notin vus$  Alors :
12              $vus \leftarrow vus \cup \{x\}$ 
13             Pour  $y$  successeur de  $x$  Faire :
14                 EXPLORER( $y$ )
15             Insérer  $x$  en tête de liste dans  $L$  //ordre décroissant
16     pour  $x \in S$  faire
17         EXPLORER( $x$ )
18     renvoyer  $L$ 
19
20 fonction KOSARAJU( $G$ ) :
21     //Initialisation :
22      $L \leftarrow \text{DFS\_POST}(G)$ 
23     composantes  $\leftarrow []$ 
24      $vus \leftarrow \emptyset$ 
25      $G^{\leftarrow} \leftarrow \text{MIROIR}(G)$ 
26
27     fonction AJOUTER( $x, C$ ) : //parcours DFS récursif de la CFC de  $x$  dans  $G^{\leftarrow}$ 
28         Si  $x \notin vus$  Alors :
29              $vus \leftarrow vus \cup \{x\}$ 
30             Pour  $y$  successeur de  $x$  dans  $G^{\leftarrow}$  Faire :
31                 AJOUTER( $y, C$ )
32             Insérer  $x$  dans  $C$ 
33
34     Pour  $x \in L$ , dans l'ordre Faire :
35         Si  $x \notin vus$  Alors :
36              $C \leftarrow []$ 
37             AJOUTER( $x, C$ ) //marquages "vus" partagés entre tous les parcours
38             Insérer  $C$  dans composantes
39
40     Renvoyer composantes
```

III.3 - Application

Une utilisation explicitement au programme des composantes fortement connexes et de Kosaraju est celle de la résolution **en temps linéaire** du problème 2-SAT, que nous verrons en TD-cours.

IV - Couplages

IV.1 - Cas général

 **Attention!** Cette section est à copier sur feuille depuis le tableau. Ceci n'est qu'une annexe de la section. 

Définition 16

Couplage :

Un sommet est dit **couvert** par un couplage C s'il est incident à l'une des arêtes de C , **libre pour** C sinon.

Remarque :

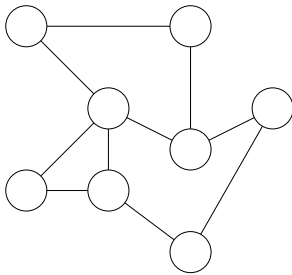
- Un couplage est un appariement (a priori partiel) des sommets du graphe : chaque sommet est associé à zéro ou un autre sommet.

Un couplage C est dit :

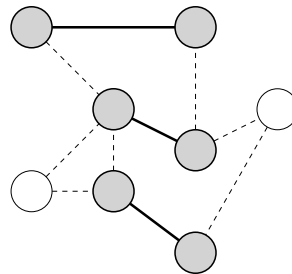
- **maximal** si
- **de cardinalité maximale** si
- **parfait** si

Exemple :

On propose le couplage suivant (à droite) sur le graphe ci-dessous. Est-il maximal ? De cardinalité maximale ? Parfait ? Proposer un couplage parfait de ce graphe. Dessinez-le sur le graphe, à gauche.



(a) Couplage parfait (à dessiner).



(b) Un couplage _____ du graphe.

Remarques :

- Un couplage *parfait* est *de cardinalité maximale*, mais la réciproque est fausse.
- On parle aussi de **couplage maximum** pour un couplage de cardinalité maximale (vocabulaire utilisé par le programme). ⚠ Ne pas confondre avec couplage **couplage maximal**. Un couplage maximum est maximal, mais la réciproque est fausse en général, comme ci-dessus.
- Un couplage est maximal si et seulement si il n'existe pas deux sommets adjacents du graphe non couverts par le couplage.

Exercice :

1. Donner une condition nécessaire très simple pour qu'un graphe admette un couplage parfait. En déduire un graphe admettant un couplage de cardinalité maximale non parfait.
2. Montrer que pour $n \geq 2$, il existe un graphe connexe à n sommets pour lequel la cardinalité maximale d'un couplage est de un.
3. Soit C et C' deux couplages sur un même graphe, avec C maximal. Montrer que $|C'| \leq 2|C|$.
4. En déduire une $\frac{1}{2}$ -approximation très simple pour le problème d'optimisation COUPLAGE MAX défini par :

Instance : un graphe $G = (S, A)$

Solution : un couplage C de G

Mesure à maximiser : $|C|$

On la décrira en Français ou en pseudo-code.

À faire sur feuille !

Soient $G = (S, A)$ un graphe non orienté et C un couplage de G .

- Un **chemin alternant** pour C est
- Un **chemin augmentant** pour C est

Exemple :

Donner les deux chemins augmentants pour le couplage de droite de la figure précédente (les entourer de deux couleurs différentes).

Remarque :

- Un chemin de longueur 1 reliant deux sommets libres est augmentant.

Soient A et B deux ensembles. La **différence symétrique** de A et de B , notée $A \Delta B$ est définie par :

$$A \Delta B = (A \cup B) \setminus (A \cap B)$$

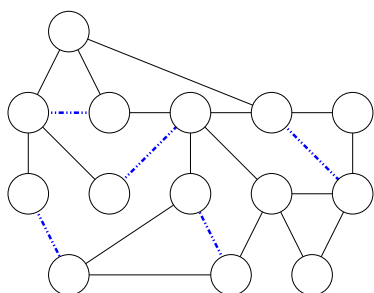
Schéma :

⚠ **Attention!** Partie sur papier. ⚠

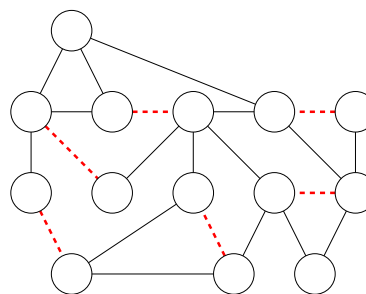
Lemme de Berge :

Soient $G = (S, A)$ un graphe non orienté et C un couplage de G . C est de cardinalité maximale si et seulement s'il n'existe pas de chemin augmentant pour C .

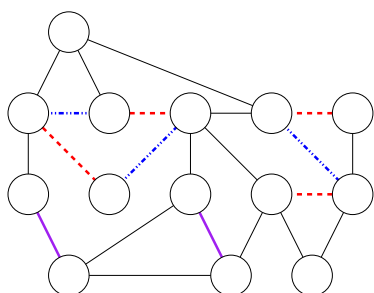
Démonstration. Sur papier! Schéma illustratif de la preuve :



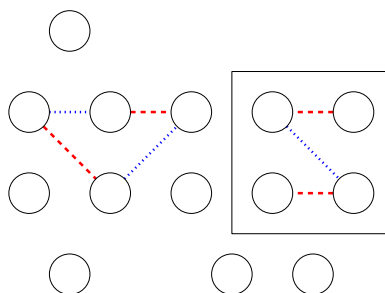
(a) Le couplage C .



(b) Le couplage C' avec $|C'| > |C|$.



(c) On combine C et C' .



(d) Les composantes connexes de $C \Delta C'$. La composante entourée fournit un chemin augmentant pour C .

□

On en déduit un algorithme permettant de calculer un couplage de cardinalité maximale :

Algorithme 4 : Couplage maximum dans un graphe

Entrées : Un graphe $G = (S, A)$ non orienté.

Sorties : Un couplage $C \subseteq A$ de cardinal maximal.

```

1  $C \leftarrow \emptyset$ 
2 tant que il existe un chemin  $c$  augmentant pour  $C$  faire
3   |  $C \leftarrow C \Delta c$ 
4 renvoyer  $C$ 
```

Remarque :

- Il reste bien sûr à régler le problème de la recherche *efficace* d'un chemin augmentant. Des algorithmes existent pour des graphes quelconques, avec une complexité totale en $O(|A||S|^2)$. Ils ne sont pas un objectif d'étude de ce chapitre, mais nous allons voir une méthode efficace dans le cas des graphes bipartis.

IV.2 - Cas des graphes bipartis

Proposition 15

Lemme d'extraction de cycle :

Soit $G = (S, A)$ un graphe. Soient $u, v \in S$ deux sommets de G . Si $uv \in A$ et si il existe un chemin reliant u à v , noté $u = x_0, x_1, \dots, x_k = v$ ne contenant pas l'arête uv , alors le graphe G possède un cycle contenant u et v .

Démonstration. Sur papier. □

Proposition 16 Définition 20

Rappel : graphe biparti.

Un graphe $G = (S, A)$ est dit **biparti** si l'on peut partitionner S en $X \sqcup Y$ de manière à ce que toutes les arêtes du graphe relient un sommet de X à un sommet de Y .

Exemple :

Un arbre est un graphe biparti.

Démonstration. Sur papier. □

Remarques :

- **Rappel** : un graphe est biparti si et seulement si il est 2-colorable (cf 1ère année).
- On peut étendre la définition à des graphes k -partis (où $V = V_1 \sqcup \dots \sqcup V_k$ et aucun sommet de V_i n'est voisin d'un autre sommet de V_i), mais l'on parle plutôt de graphe k -colorable dans ce cas.

Le cas des graphes bipartis est celui dans lequel la notion de couplage est la plus naturelle.

Exemples :

- Graphe biparti $S = X \cup Y$ avec X des candidats et Y des postes. Par exemple, un ensemble d'arêtes peut indiquer quels candidats sont intéressés par quel poste. Deux candidats ne peuvent obtenir le même poste, ni un candidat occuper deux postes différents, ce qui est exactement la définition d'un couplage.
- X des tâches et Y des processeurs (ou des personnes) susceptibles de les réaliser.

⚠ **Attention!** Suite de la partie sur papier. ⚠

V - Algorithme A*

V.1 - Introduction au problème

Dans cette partie, on considère des graphes pondérés et (a priori) orientés. On notera $\rho(e)$ le poids d'un arc e . On va présenter l'algorithme A^* , un algorithme de recherche de plus court chemin plus efficace que l'algorithme de Dijkstra dans le cadre de la recherche de distance entre une source et une destination fixées.

En effet, l'algorithme de Dijkstra recherche des plus courts chemins depuis **un sommet source** vers **tous les sommets du graphe**. Il arrive souvent qu'on ne soit intéressé que par un plus court chemin depuis **un sommet source** vers **un sommet destination**.

Exemples :

- recherche d'itinéraire dans un réseau de transport
- planification de mouvement en robotique ou dans le jeu vidéo

On peut bien sûr modifier l'algorithme de Dijkstra pour qu'il s'arrête dès qu'un plus court chemin vers un certain sommet but est trouvé : il suffit d'ajouter deux lignes dans le pseudo-code.

Il suffit d'ajouter les deux lignes suivantes dans le code de l'algorithme de Dijkstra (sous l'extraction du minimum) :

Pseudo-code

```
1  $u \leftarrow$  Extraire de  $F$  le sommet dont la valeur dans  $dist$  est minimale
2 Si  $u = destination$  Alors
3   Renvoyer  $dist[destination]$ 
```

Cependant, cette modification ne change au caractère omni-directionnel de la recherche : on examine tous les sommets par distance croissante à l'origine, et l'on s'arrête dès que l'on atteint notre but.

V.2 - Présentation de l'algorithme

Principe de l'algorithme A^* :

À tout moment, pour un sommet u donné du graphe, on a les estimations suivantes :

- le plus court chemin trouvé jusqu'à maintenant entre la source et u , ainsi que sa longueur $d'(source, u)$. Cette longueur est donc une **majoration** de la distance entre la source et u (longueur du plus court chemin existant dans le graphe) : $d(source, u) \leq d'(source, u)$.
- le terme $h(u)$ qui estime la distance restant à parcourir entre u et la destination. L'algorithme ne sera **correct** que si $h(u)$ est une **minoration** de la distance $d(u, destination)$: $h(u) \leq d(u, destination)$

Exemple : Graphe euclidien :

Dans le cas d'un graphe euclidien, chaque sommet correspond à un point du plan, et le poids de l'arc $u \rightarrow v$, s'il existe, vaut $\|v' - v\|_2$. C'est le cas, par exemple, d'une ville dans laquelle la portion d'une rue comprise entre deux intersections est un segment de droite.

On prend alors comme heuristique $h(u)$ la distance « à vol d'oiseau » de u à la destination.

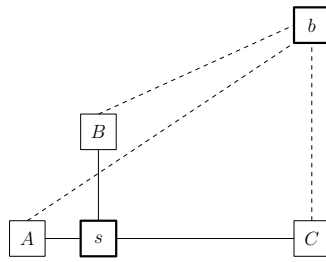


FIGURE 11 – Extrait de graphe euclidien. Les arêtes en trait plein existent, celles en pointillé peuvent ou non exister.

Exercice :

1. Dans quel ordre l'algorithme de Dijkstra explore-t-il les sommets ?
2. Quel est le sommet v voisin de s qui minimise la distance estimée vers b , i.e. ici $d(s, v) + h(v)$?

L'algorithme A^* va donc commencer par explorer ce sommet, plutôt que celui privilégié par Dijkstra.

Exemple de déroulement : Plus court chemin du Lycée du Parc (à Lyon) à l'ENS Lyon, algorithme de Dijkstra à gauche et A^* à droite. Les sommets sont coloriés en fonction de leur instant d'exploration.



Algorithme A^* :

Les structures de données utilisées dans l'algorithme A^* décrit ci-dessous sont les suivantes :

- tableau `dist` : indique pour chaque sommet u le poids du plus court chemin trouvé jusqu'à présent de source à u .
- file de priorité `ouverts` : contient les sommets u à explorer, avec comme priorité les valeurs de $d'(\text{source}, u) +$

$h(u)$.

- tableau **parents** : code un arbre permettant de reconstruire un chemin depuis la source jusqu'à chaque sommet exploré.

Pseudo-code

```
1  Entrées :  $G = (S, A, \rho)$  un graphe pondéré, une heuristique  $h$ ,  
2  un sommet source et un sommet destination.  
3  Sortie : Un chemin (de poids minimal ?) de source à but, si il en existe un.  
4   $A^*(G, s)$ :  
5       $\text{dist}[t] \leftarrow [\infty, \dots, \infty]$   
6       $\text{dist}[\text{source}] \leftarrow 0$   
7       $\text{parents} \leftarrow (\text{Nil}, \text{Nil}, \dots \text{Nil})$   
8       $\text{ouverts} \leftarrow \text{FILEPRIORITE}()$   
9      Insérer source dans ouverts avec la priorité  $h(\text{source})$   
10     Tantque  $\text{ouverts} \neq \emptyset$  Faire :  
11          $(u, \_) \leftarrow \text{EXTRAIREMIN}(\text{ouverts})$   
12         Si  $u = \text{destination}$  alors :  
13             Renvoyer le chemin trouvé en remontant les pères de destination à source.  
14         Pour chaque  $v$  successeur de  $u$  Faire :  
15              $d \leftarrow \text{dist}[u] + \rho(u \rightarrow v)$   
16             Si  $d < \text{dist}[v]$  Alors :  
17                  $\text{parents}[v] \leftarrow u$   
18                  $\text{dist}[v] \leftarrow d$   
19             Si  $v \in \text{ouverts}$  Alors :  
20                 Remplacer la priorité de  $v$  dans ouverts par  $d + h(v)$   
21             Sinon :  
22                 Insérer  $v$  dans ouverts avec la priorité  $d + h(v)$   
23     Renvoyer  $\emptyset$ 
```

N'hésitez pas à comparer ce pseudo-code à celui fourni par l'algorithme de Dijkstra (avec construction du plus court chemin). Là encore, on pourrait donner des versions de A^* qui se contentent de renvoyer la distance entre la source et la destination, ou qui insèrent plusieurs fois dans la file plutôt que de diminuer la priorité.

Remarques :

- En pratique, on utiliserait plutôt des dictionnaires que des tableaux pour dist et parents . En effet, l'intérêt de A^* est qu'on peut le plus souvent n'explorer qu'une petite partie du graphe, et il est donc dommage de réserver une quantité de mémoire proportionnelle au nombre total de sommets. Dans le cas assez fréquent où le graphe n'est connu que de manière implicite, cela peut même s'avérer impossible.
- On peut voir l'algorithme A^* comme un algorithme de *branch and bound* (séparation et évaluation). En effet, l'algorithme A^* explore toutes les solutions admissibles sous forme arborescente (les plus courts chemins depuis la source dans un graphe) et élimine des solutions potentielles non pertinentes : celles qui passent par des sommets n'ayant aucune chance de participer à un plus court chemin entre la source et la destination. Cet élagage de l'exploration se fait grâce à l'heuristique h , qui doit être admissible.

Exercice : Que devient l'algorithme A^* si l'heuristique h est :

1. constante égale à zéro ?
2. **parfaite**, c'est-à-dire telle que $h(u) = d(u, \text{destination})$ pour tout sommet u ?

⚠ En l'absence d'hypothèse supplémentaire sur l'heuristique h , dans le cadre de l'algorithme A^* , il est possible qu'un sommet soit inséré et extrait plusieurs fois. Lorsqu'on extrait de la file un sommet qui n'est pas le sommet destination, on n'a pas encore forcément trouvé sa distance à la source et un chemin le plus court depuis la source. On pourra trouver plus court plus tard.

Exemple :

Dans le graphe ci-dessous, le sommet destination est v_0 , le sommet source v_5 et les valeurs de $h(v)$ sont notées à l'intérieur des sommets.

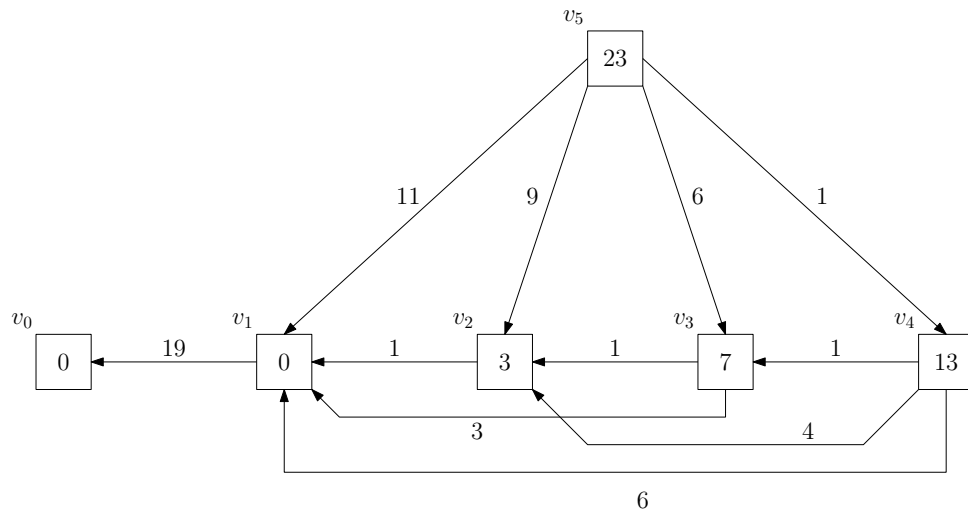


FIGURE 13 – Graphe G_e

Exercice :

1. Analyser le déroulement de l'algorithme A^* sur ce graphe.
2. Que remarquez-vous? Combien de fois passe-t-on par chaque sommet, dans le pire cas?

À faire sur feuille!

V.3 - Heuristique admissible, heuristique cohérente

Sans hypothèse sur l'heuristique h , il n'y a aucune raison que l'algorithme soit correct et que le chemin renvoyé soit de poids minimal! On a seulement la garantie que si un chemin de la source vers la destination existe, la fonction en renverra bien un (et elle détectera l'absence de chemin le cas échéant).

Exemple :

Considérons le graphe (non orienté) suivant (on a représenté les poids sur les arêtes et les valeurs de h à l'intérieur des sommets) :

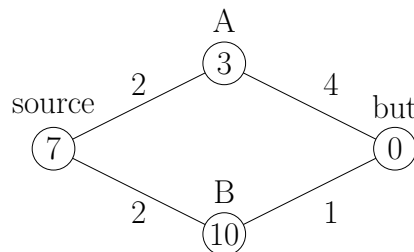


FIGURE 14 – Un graphe G_p problématique pour A^* .

Exercice :

1. Simuler l'exécution de l'algorithme A^* sur ce graphe, et déterminer le chemin renvoyé.
2. D'où vient le problème?

Heuristique admissible, heuristique cohérente :

Une heuristique h est dite :



- **admissible** si :
- **cohérente** (ou **monotone**) si :

Remarque :

-
- Une heuristique est donc admissible si elle est « optimiste », c'est-à-dire si elle ne surestime jamais la distance d'un sommet à la destination.

Exemple :

Dans le graphe G_e plus haut, l'heuristique h est-elle admissible ? cohérente ? Et dans le graphe G_p précédent ?

 Attention! Suite de la partie sur papier. 
--

Remarque (théorème de correction de l'algorithme A^*) :

Si l'heuristique est seulement supposée admissible, un sommet (autre que le sommet destination) peut être extrait à plusieurs reprises de la file, ce qui peut dans certains cas pathologiques résulter en une complexité exponentielle en le nombre de sommets du graphe (ex : graphe G_e). On peut faire mieux en imposant que l'heuristique soit cohérente.

Remarque (complexité avec heuristique cohérente) :

Ainsi la complexité obtenue en pire cas est encore $O((|S| + |A|) \log |S|)$, et en pratique bien meilleure que celle de Dijkstra car on ne visitera qu'un (petit) sous-ensemble de sommets (et d'arêtes).