

## - Algorithmes du cours -

### Chapitre 1 : Définitions de base, connexité

**Algorithme :** COMPOSANTE

**Données :** Un graphe  $G = (V, E)$  donné par liste d'arêtes.

**Résultat :** Le tableau  $comp$  vérifiant  $comp(x) = comp(y)$  ssi il existe un chemin de  $x$  à  $y$  (ou autrement dit ssi  $x$  et  $y$  appartiennent à la même composante connexe de  $G$ ).

```

1 début
2   pour tous les  $x \in V$  faire  $comp(x) \leftarrow x$ ;
3   pour tous les  $xy \in E$  faire
4     si  $comp(x) \neq comp(y)$  alors
5        $aux \leftarrow comp(x)$ ;
6       pour tous les  $z \in V$  faire
7         si  $comp(z) = aux$  alors  $comp(z) \leftarrow comp(y)$ ;

```

**Analyse de COMPOSANTE :**

- **Terminaison :** L'algorithme ne contient que des boucles 'pour' et des tests, il termine donc.
- **Preuve :** Montrons par récurrence sur le nombre  $i$  d'arêtes traitées **ligne 3** qu'à chaque étape on a la propriété  $\mathcal{P}_i$  : ' $comp(x) = comp(y)$  ssi il existe une  $xy$ -marche dans le graphe  $G_i$  formé par les arêtes précédemment traitées'.

- Si on a traité aucune arête,  $\mathcal{P}_0$  est vérifiée.

- Supposons  $\mathcal{P}_i$  vraie et notons  $xy$  la  $i+1$ -ème arête traitée **ligne 3**. Regardons le résultat du test effectué **ligne 4** à la  $i+1$ -ème étape :

- Si  $comp(x) = comp(y)$  : par  $\mathcal{P}_i$ ,  $x$  et  $y$  sont déjà reliés par une  $xy$ -marche. Le tableau  $comp$  est inchangé et la propriété  $\mathcal{P}_{i+1}$  est donc vraie.

- Si  $comp(x) \neq comp(y)$  : dans  $G_i$  (avant le traitement de  $xy$  donc), notons  $X$  l'ensemble des sommets ayant  $comp(x)$  comme valeur de  $comp$  et  $Y$  l'ensemble des sommets ayant  $comp(y)$  comme valeur de  $comp$ . On peut remarquer que par  $\mathcal{P}_i$ ,  $X$  (resp.  $Y$ ) est la composante connexe de  $G_i$  contenant  $x$  (resp.  $y$ ). On traite donc  $xy$  et les **ligne 5** à **ligne 7** affectent  $comp(y)$  à la valeur de  $comp$  de tous les sommets de  $X$ . Il faut s'assurer maintenant que  $\mathcal{P}_{i+1}$  est vraie (on peut noter que  $G_{i+1}$  est obtenu en ajoutant  $xy$  à  $G_i$ ). Par  $\mathcal{P}_i$ , deux sommets  $u$  et  $v$  dont les valeurs de  $comp$  n'ont pas changé (c'est-à-dire  $u \notin X$  et  $v \notin X$ ) restent reliés par une marche ssi leur valeur  $comp$  est égale. Si  $u \in X$  et  $v \in X$ , avant le traitement de  $xy$ , on avait  $comp(u) = comp(v) = comp(x)$  et  $u$  et  $v$  étaient reliés par une marche. Après le traitement de  $xy$ , on a  $comp(u) = comp(v) = comp(y)$  et  $u$  et  $v$  sont toujours reliés par une marche. Finalement, si  $u \in X$  et  $v \notin X$ , soit  $v \notin Y$  et il n'existe toujours pas de marche de  $x$  à  $y$  dans  $G_{i+1}$  et on a bien  $comp(u) \neq comp(v)$ , soit  $v \in Y$ . Dans ce cas, après le traitement de  $xy$ , on a  $comp(u) = comp(v)$ . Voyons qu'il existe une marche de  $u$  à  $v$  dans  $G_{i+1}$ . Avant le traitement de  $xy$  on avait  $comp(u) = comp(x)$  et par  $\mathcal{P}_i$  il existait une marche  $M$  de  $u$  à  $x$  dans  $G_i$ . De même, on avait  $comp(v) = comp(y)$  et il existait une marche  $M'$  de  $y$  à  $v$  dans  $G_i$ . Comme on ajoute l'arête  $xy$  à  $G_i$  pour obtenir  $G_{i+1}$ ,  $MM'$  forme une marche de  $u$  à  $v$  dans  $G_{i+1}$ . Le cas  $v \in X$  et  $u \notin X$  se traite symétriquement.

- **Complexité :** COMPOSANTE a un temps d'exécution  $O(n^2)$  dans le pire des cas.

En effet, la **ligne 2** effectue  $n$  opérations. La boucle de la **ligne 3** s'exécute  $m$  fois. À chaque fois que le test de la **ligne 4** est vrai on fusionne deux composantes et un numéro de composante disparaît. Ce test est donc vrai au plus  $n-1$  fois. L'exécution des **lignes 6** et **7** demande au plus  $n$  opérations et elles sont exécutées au plus  $n-1$  fois. Finalement, la boucle **ligne 3** demande un temps total  $O(m+n^2) = O(n^2)$  (car  $m \leq n^2/2$ ) et l'algorithme en entier, un temps  $O(n+n^2) = O(n^2)$ .

## Chapitre 2 : Arbres couvrants

**Algorithme :** ARBRE-COUVRANT

**Données :** Un graphe connexe  $G = (V, E)$  donné par liste d'arêtes.

**Résultat :** Un ensemble  $A \subseteq E$  d'arêtes tel que le graphe  $(V, A)$  soit un arbre couvrant de  $G$ .

```

1 début
2    $A \leftarrow \emptyset$ ;
3   pour tous les  $x \in V$  faire  $comp(x) \leftarrow x$ ;
4   pour tous les  $xy \in E$  faire
5     si  $comp(x) \neq comp(y)$  alors
6        $aux \leftarrow comp(x)$ ;
7        $A \leftarrow A \cup \{xy\}$ ;
8       pour tous les  $z \in V$  faire
9         si  $comp(z) = aux$  alors  $comp(z) \leftarrow comp(y)$ ;
10  Retourner  $A$ ;

```

(en gris, est noté ce qui est rajouté par rapport à l'algo COMPOSANTE)

**Analyse de ARBRE-COUVRANT :**

- **Terminaison :** L'algorithme ne contient que des boucles 'pour' et des tests, il termine donc.
- **Preuve :** On note  $A_i$  l'ensemble  $A$  formé après  $i$  tours de la boucle de la ligne 4. On reprend les notations de la preuve de l'algorithme COMPOSANTE pour montrer qu'à chaque étape,  $A_i$  induit un arbre couvrant sur chaque composante connexe du graphe  $G_i$ . C'est clair avant la première exécution de la ligne 4. À l'étape  $i + 1$  lorsqu'on traite une arête  $xy$  ligne 4, si  $comp(x) = comp(y)$  alors rien ne change, ni les valeurs de  $comp$ , ni  $A_i$ , ni les composantes connexes de  $G_i$ . Si  $comp(x) \neq comp(y)$ , alors on a vu dans la preuve de COMPOSANTE qu'en ajoutant l'arête  $xy$  à  $G_i$ , on fusionne les composantes connexes de  $x$  et  $y$  pour obtenir une seule composante connexe de  $G_{i+1}$ . De même, ligne 7, on va lier les arbres couvrants les composantes connexes de  $x$  et  $y$  dans  $G_i$  par l'arête  $xy$ . L'ensemble  $A_{i+1}$  induit ainsi un arbre couvrant de la composante connexe de  $x$  (et  $y$ ) dans  $G_{i+1}$ .  
À la fin de l'algorithme,  $G$  étant connexe,  $A$  formera un arbre couvrant de l'unique composante connexe de  $G$ .
- **Complexité :** ARBRE-COUVRANT a un temps d'exécution  $O(n^2)$  dans le pire des cas, comme COMPOSANTE.

**Algorithme :** ALGO DE KRUSKAL

**Données :** Un graphe connexe  $G = (V, E)$  donné par liste d'arêtes avec une fonction de poids  $w$  sur les arêtes de  $G$ .

**Résultat :** Un ensemble  $A$  d'arêtes de  $G$  formant un arbre couvrant de poids minimum (ACPM) de  $G$ .

```

1 début
2   Trier les arêtes de  $G$  par ordre de poids  $w$  croissant ;
3    $A \leftarrow \emptyset$  ;
4   pour tous les  $x \in V$  faire  $comp(x) \leftarrow x$  ;
5   pour tous les  $xy \in E$ , en suivant l'ordre précédemment calculé faire
6     si  $comp(x) \neq comp(y)$  alors
7        $aux \leftarrow comp(x)$  ;
8        $A \leftarrow A \cup \{xy\}$  ;
9       pour tous les  $z \in V$  faire
10        si  $comp(z) = aux$  alors  $comp(z) \leftarrow comp(y)$  ;
11  retourner  $A$  ;

```

(en gris, est noté ce qui est rajouté par rapport à l'algo ARBRE-COUVRANT)

### Analyse de KRUSKAL :

- **Terminaison** : L'algorithme ne contient que des boucles 'pour' et des tests, il termine donc.
- **Preuve** : Cet algorithme est une implémentation particulière de l'algorithme ARBRE-COUVRANT. Le graphe  $G$  étant connexe, on sait donc que KRUSKAL va retourner un arbre couvrant de  $G$ , reste à montrer que celui-ci est de poids minimal. Pour cela, on note  $T$  l'arbre couvrant retourné par KRUSKAL et on considère  $A$  un ACPM de  $G$  ayant un nombre maximal d'arêtes en commun avec  $T$ . Si  $T = A$ , il n'y a rien à faire,  $T$  est bien un ACPM de  $G$ . Supposons que  $T \neq A$  et considérons  $xy$  une arête de  $A$  n'appartenant pas à  $T$ . Comme  $xy$  n'a pas été choisi par l'algo,  $x$  et  $y$  étaient déjà reliés par un chemin  $P$  formée d'arêtes  $e_1, \dots, e_\ell$  précédemment traitées et ajoutées dans  $T$  par l'algo. En particulier, comme les arêtes ont été triées par poids croissant, on a pour tout  $i = 1, \dots, \ell$ ,  $w(e_i) \leq w(xy)$ . On va modifier  $A$  :  $A \setminus xy$  contient deux composantes connexes :  $C_1$  contenant  $x$  et  $C_2$  contenant  $y$ . Comme  $P$  relie  $x$  à  $y$  une (au moins) arête de  $P$  a une extrémité dans  $C_1$  et l'autre dans  $C_2$ . On note  $e_k$  cette arête et on peut remarquer que  $e_k \notin A$  (sinon  $xy$  et  $e_k$  seraient deux arêtes de  $A$  reliant les composantes connexes pour  $A$ ,  $C_1$  et  $C_2$ ;  $A$  contiendrait un cycle). On va 'modifier'  $A$  : l'ensemble  $A' = (A \setminus xy) \cup e_k$  forme un arbre couvrant de  $G$  de poids  $w(A) - w(xy) + w(e_k)$ . Si  $w(e_k) < w(xy)$ ,  $A$  ne serait pas un ACPM de  $G$ , puisque  $A'$  serait un arbre couvrant de  $G$  de poids strictement inférieur à  $A$ . On a donc  $w(xy) = w(e_k)$  (on avait déjà remarqué  $w(e_k) \leq w(xy)$ ). Ainsi,  $A'$  est aussi un ACPM de  $G$ , mais a une arête de plus en commun avec  $T$  que  $A$ , ce qui contredit le choix de  $A$ . On avait donc  $T = A$  et KRUSKAL revoit bien l'ensemble d'arêtes d'un ACPM.
- **Complexité** : Le tri des arêtes demande un temps en  $O(m \log m) = O(m \log n)$ . Le reste de l'algo a le même temps que composante, donc  $O(n^2)$  en version simple, ou  $O(m + n \log n)$  en version optimisée. Le temps d'exécution de KRUSKAL dans le pire des cas est donc en  $O((m + n) \log n) = O(m \log n)$  (ou  $O(n^2 + m \log n)$  en version simple).

**Algorithme** : COMPOSANTE-OPTIMISÉ

**Données** : Un graphe  $G = (V, E)$  donné par liste d'arêtes.

**Résultat** : Une fonction  $comp : V \rightarrow V$  telle que  $comp(x) = comp(y)$  si, et seulement si,  $x$  et  $y$  appartiennent à la même composante connexe.

```

1 début
2   pour tous les  $x \in V$  faire
3      $comp(x) \leftarrow x$ ;
4      $L(comp(x)) \leftarrow \{x\}$ ; // liste des sommets de  $comp(x)$ , gérée par une pile
5      $t(comp(x)) \leftarrow 1$ ; // taille de  $comp(x)$ 
6   pour tous les  $xy \in E$  faire
7     si  $comp(x) \neq comp(y)$  alors
8       si  $t(comp(x)) > t(comp(y))$  alors échanger  $x$  et  $y$ ;
9        $aux \leftarrow comp(x)$ ;
10       $t(comp(y)) \leftarrow t(comp(y)) + t(aux)$ ;
11      pour tous les  $z \in L(aux)$  faire
12         $comp(z) \leftarrow comp(y)$ ;
13        Empiler  $z$  sur  $L(comp(y))$ ;
14        Dépiler  $z$  de  $L(aux)$ ;

```

#### Analyse de COMPOSANTE-OPTIMISÉ :

- **Preuve** : C'est une autre implémentation de l'algo COMPOSANTE, on admet sa validité.
- **Complexité** : Les lignes 2 à 5 demandent un temps  $O(n)$ . La boucle de la ligne 6 s'exécute  $m$  fois et les lignes 7 à 10 et 12 à 14 demandent chacune un temps en  $O(1)$ . Les lignes 12 à 14 s'exécutent autant de fois qu'un sommet change de numéro de composante. Mais à chaque fois que cela se produit pour un sommet  $x$ , le nombre de sommets ayant même numéro de composante que  $x$  double au moins (grâce à la ligne 8). Un sommet change donc au plus  $\log n$  fois de numéro de composante. Ainsi toutes les exécutions des lignes 7 à 10 demandent un temps en  $O(m)$  et toutes les exécutions des lignes 12 à 14 demandent un temps en  $O(n \log n)$ . Finalement, COMPOSANTE-OPTIMISÉ fonctionne en temps  $O(m + n \log n)$ .

## Chapitre 3 : parcours

#### Algorithme : PARCOURS-EN-LARGEUR

**Données** : Un graphe  $G = (V, E)$  donné par liste de voisins, et  $r$  un sommet de  $G$ , la racine.

**Résultat** : Trois fonctions :  $ordre : V \rightarrow \{1, \dots, n\}$  (position dans le parcours),  $pere : V \rightarrow V$  (père dans le parcours) et  $niv : V \rightarrow \mathbb{N}$  (niveau : distance à la racine).

```

1 début
2   pour tous les  $v \in V$  faire  $dv(v) \leftarrow 0$ ; // sommets déjà vus;
3    $dv(r) \leftarrow 1$ ;  $ordre(r) \leftarrow 1$ ;  $pere(r) \leftarrow r$ ;  $niv(r) \leftarrow 0$ ; // la racine
4   Enfiler  $r$  dans  $AT$ ; // sommets à traiter,  $AT$  gérée comme une file
5    $t \leftarrow 2$ ; // le temps
6   tant que  $AT \neq \emptyset$  faire
7     Prendre  $v$  le premier sommet de  $AT$  l'enlever de  $AT$ ;
8     pour tous les  $x \in Vois(v)$  faire
9       si  $dv(x) = 0$  alors
10         $dv(x) \leftarrow 1$ ; // on traite  $x$  pour la première fois
11        Enfiler  $x$  dans  $AT$ , en dernière position;
12         $ordre(x) \leftarrow t$ ;  $t \leftarrow t + 1$ ;
13         $pere(x) \leftarrow v$ ;
14         $niv(x) \leftarrow niv(v) + 1$ ;

```

#### Analyse de PARCOURS-EN-LARGEUR :

Algos

L3 Info, L3 Math-Info.

• **Terminaison/complexité** : Un sommet apparaît une fois au plus (et même exactement) dans  $AT$ . En effet lorsqu'on place un sommet  $x$  dans  $AT$ , il faut que  $dv(x) = 0$  (**ligne 9**) et on affecte alors  $dv(x)$  à 1 ((**ligne 10**),  $x$  ne pourra pas être replacé dans  $AT$ . La **ligne 7** s'exécute donc au pire  $n$  fois. Ensuite, **ligne 8**, on explore tous les voisins de chaque sommet placé dans  $AT$ . Ainsi les instructions des lignes 9 à 14 seront effectuées au plus  $\sum\{deg(x) : x \in G\} = 2m$  fois. Toutes les instructions élémentaires ayant un temps d'exécution en  $O(1)$ , l'algorithme termine en temps  $O(n + m)$ .

• **Preuve** : On va montrer que l'arbre de parcours est un arbre des plus courts chemins (a.p.c.c.).

- *Fait 1* : Si  $ordre(x) < ordre(y)$  alors  $ordre(pere(x)) \leq ordre(pere(y))$ .

En effet, sinon, on aurait  $ordre(pere(y)) < ordre(pere(x))$ , mais on défilerait d'abord  $pere(y)$  et placerait  $y$  dans la file puis plus tard, on défilerait  $pere(x)$  et placerait  $x$  dans la file. Donc,  $y$  serait traité avant  $x$ , ce qui contredit  $ordre(x) < ordre(y)$ .

- *Fait 2* : Si  $ordre(x) < ordre(y)$  alors  $niv(x) \leq niv(y)$ .

On procède par récurrence sur la valeur du niveau. Plus précisément, on définit l'hypothèse  $\mathcal{H}_k$  : 'Quelques soient  $x$  et  $y$  avec  $niv(x) \leq k$  et  $niv(y) \leq k$  on a si  $ordre(x) < ordre(y)$  alors  $niv(x) \leq niv(y)$ '.  $\mathcal{H}_0$  est vraie puisque  $r$  est le seul sommet de niveau 0. Supposons que  $\mathcal{H}_k$  soit vraie et considérons deux sommets  $x$  et  $y$  avec  $niv(x) \leq k+1$  et  $niv(y) \leq k+1$  et disons  $ordre(x) < ordre(y)$ . On veut montrer que  $niv(x) \leq niv(y)$ . Si  $x = r$  alors  $niv(x) \leq niv(y)$  est clair. Sinon,  $x$  et  $y$  ont tous les deux un père (différent d'eux-mêmes). Par construction du niveau, **ligne 14**, on a  $niv(pere(x)) = niv(x) - 1$  et  $niv(pere(y)) = niv(y) - 1$  et en particulier,  $niv(pere(x)) \leq k$  et  $niv(pere(y)) \leq k$ . Comme  $ordre(x) < ordre(y)$ , par le Fait 1, on a  $ordre(pere(x)) \leq ordre(pere(y))$ . Si  $ordre(pere(x)) = ordre(pere(y))$  alors  $pere(x) = pere(y)$  et  $niv(x) = niv(pere(x)) + 1 = niv(pere(y)) + 1 = niv(y)$ . Si  $ordre(pere(x)) < ordre(pere(y))$  alors, par  $\mathcal{H}_k$ , comme  $niv(pere(x)) \leq k$  et  $niv(pere(y)) \leq k$ , on a  $niv(pere(x)) \leq niv(pere(y))$ . On en déduit donc  $niv(x) = niv(pere(x)) + 1 \leq niv(pere(y)) + 1 = niv(y)$ . Ayant choisi  $x$  et  $y$  quelconques, de niveau au plus  $k+1$ , on a prouvé  $\mathcal{H}_{k+1}$ .

- *Fait 3* : Pour toute arête  $xy$  de  $G$ , on a  $|niv(x) - niv(y)| \leq 1$  (autrement dit  $T$  est bien un APCC).

En effet, soit  $xy$  une arête de  $G$  avec (en toute généralité)  $ordre(x) < ordre(y)$ . Lorsqu'on défile  $x$ , si  $dv(y) = 0$  alors  $pere(y) = x$  et  $niv(y) = niv(x) + 1$ . Si, par contre,  $dv(y) = 1$ , c'est que le père de  $y$  a déjà été traité **ligne 7** et défilé et que  $y$  est encore dans la file. On a donc  $ordre(pere(y)) < ordre(x) < ordre(y)$ . Par le Fait 2, on a ainsi  $niv(pere(y)) \leq niv(x) \leq niv(y)$ , soit  $niv(y) - 1 \leq niv(x) \leq niv(y)$ .

**Algorithme** : PARCOURS-EN-PROFONDEUR

**Données** : Un graphe  $G = (V, E)$  donné par listes de voisins, gérées comme des piles, et  $r$  un sommet de  $G$ .

**Résultat** : Deux fonctions  $debut : V \rightarrow \{1, \dots, 2n\}$  et  $fin : V \rightarrow \{1, \dots, 2n\}$ , dates de début et fin de traitement, et une fonction  $pere : V \rightarrow V$ .

```

1  début
2  |   pour tous les  $x \in V$  faire  $dv(x) \leftarrow 0$ ; // sommets déjà vus;
3  |    $dv(r) \leftarrow 1$ ;  $debut(r) \leftarrow 1$ ;  $pere(r) \leftarrow r$ ; // la racine
4  |   Empiler  $r$  sur  $AT$ ; // sommets à traiter,  $AT$  gérée comme une pile
5  |    $t \leftarrow 2$ ; // le temps
6  |   tant que  $AT \neq \emptyset$  faire
7  |       |   Noter  $x$  le sommet en haut de  $AT$ ;
8  |       |   si  $vois(x) = \emptyset$  alors
9  |       |       |   Dépiler  $AT$ ;
10 |       |       |    $fin(x) \leftarrow t$ ;  $t \leftarrow t + 1$ ; // fin de traitement pour  $x$ 
11 |       |   sinon
12 |       |       |   Noter  $y$  le sommet en haut de  $vois(x)$  et dépiler  $vois(x)$ ;
13 |       |       |   si  $dv(y) = 0$  alors
14 |       |       |       |    $dv(y) \leftarrow 1$ ; // on traite  $y$  pour la première fois
15 |       |       |       |   Empiler  $y$  sur  $AT$ ;
16 |       |       |       |    $debut(y) \leftarrow t$ ;  $t \leftarrow t + 1$ ;
17 |       |       |       |    $pere(y) \leftarrow x$ ;

```

**Analyse de PARCOURS-EN-PROFONDEUR :**

• **Terminaison/complexité** : comme pour PARCOURS-EN-LARGEUR, un sommet  $x$  est placé dans  $AT$  si et seulement si il vérifie  $dv(x) = 0$ . Comme on affecte  $dv(x)$  à 1 dès qu'il est placé dans  $AT$ ,  $x$  sera placé au plus une fois dans  $AT$ . La suite du raisonnement est identique et, sous l'hypothèse que l'empilement et le dépilement se font en  $O(1)$ , la complexité de PARCOURS-EN-PROFONDEUR est en  $O(n + m)$ .

• **Preuve** : On se sert des intervalles de présence dans la pile pour voir qu'on a un arbre normal.

- *Fait 1* : On n'a pas d'intervalles qui se chevauchent, c'est-à-dire que dans le mot  $M$  décrivant la présence dans la pile, on n'a pas  $\dots x \dots y \dots x \dots y \dots$  quelques soient les sommets  $x$  et  $y$ .

En effet, cela vient du fonctionnement de la pile. Si on empile  $x$  avant  $y$ , on doit dépiler  $x$  avant  $y$ .

- *Fait 2* : Si dans  $M$  on a  $\dots x f_1 \dots f_1 f_2 \dots f_2 \dots f_k \dots f_k x \dots$  alors  $f_1, \dots, f_k$  sont exactement les fils de  $x$  dans le parcours.

Voyons d'abord que les  $f_i$  sont bien des fils de  $x$ . Le sommet  $f_1$  est empilé juste après  $x$  donc  $pere(f_1) = x$ . À la fin de l'intervalle de  $f_1$ ,  $f_1$  est dépilé de  $AT$  ligne 9 et  $x$  se retrouve en haut de  $AT$ . Le sommet  $f_2$  est alors empilé et on a donc  $pere(f_2) = x$  et ainsi de suite jusqu'à  $f_k$ . Montrons maintenant que les fils de  $x$  sont tous des  $f_i$ . Si  $y$  est un fils de  $x$ , alors  $y$  a été traité lorsque  $x$  était en haut de la pile, donc  $y$  est bien l'un des  $f_i$ .

En particulier,  $x$  est descendant de  $y$  ssi on a  $\dots y \dots x \dots x \dots y \dots$  dans le mot  $M$ .

- *Fait 3* :  $T$  est un arbre normal.

Prenons  $xy$  une arête de  $G$  avec disons  $debut(x) < debut(y)$ . Dans le mot  $M$ , on ne peut pas avoir  $\dots x \dots x \dots y \dots y \dots$  puisqu'au moment où on dépile  $x$ , tous ses voisins ont été visités. Dans  $M$ , on a donc  $\dots x \dots y \dots y \dots x \dots$ , et par le Fait 2,  $y$  est un descendant de  $x$ .

## Chapitre 5 : plus court chemins

**Algorithme** : ALGO DE DIJKSTRA

**Données** : Un graphe  $G = (V, E)$  donné par liste de voisin avec  $l$  une fonction de longueur positive sur les arêtes, et  $r$  un sommet de  $G$ , la racine.

**Résultat** : Une fonction  $d : V \rightarrow \mathbb{R}^+$  donnant la distance à la racine  $r$  et une fonction  $pere : V \rightarrow V$ .

```

1 début
2   pour tous les  $v \in V$  faire
3      $d(v) \leftarrow +\infty$ ;
4      $traite(v) \leftarrow 0$ ; // pour marquer les sommets traités
5    $pere(r) \leftarrow r$ ;  $d(r) \leftarrow 0$ ; // la racine
6   tant que il existe  $x$  avec  $traite(x) = 0$  faire
7     Choisir un tel  $x$  avec  $d(x)$  minimum;
8      $traite(x) \leftarrow 1$ ;
9     pour tous les  $y \in Vois(x)$  faire
10      si  $traite(y) = 0$  et  $d(y) > d(x) + l(xy)$  alors
11         $d(y) \leftarrow d(x) + l(xy)$ ; //  $x$  est un raccourci pour atteindre  $y$ 
12         $pere(y) \leftarrow x$ ;

```

**Analyse de DIJKSTRA :**

• **Terminaison** : à chaque passage dans le 'tant que' de la ligne 6, le nombre de sommets  $x$  avec  $traite(x) = 1$  croît strictement. L'algorithme termine donc.

• **Preuve** : L'algorithme de DIJKSTRA est un algorithme de parcours prenant en entrée un graphe connexe. La fonction  $pere$  retournée correspond à l'arbre de parcours et les valeurs de  $d$  aux distances des chemins issus de  $r$  dans l'arbre. On va montrer que ces chemins sont bien des p.c.c. Pour cela, prouvons par

récurrance sur  $k$ , le nombre de sommets traités, que  $\mathcal{H}_k$  : 'les distances sont calculées correctement pour les sommets traités à l'étape au plus  $k$ ' est vraie.

Lorsque seule la racine est traitée, sa distance à elle-même est bien 0. Supposons maintenant que à une certaine étape  $k < n$  de l'algorithme,  $\mathcal{H}_k$  soit vraie et considérons  $x$  le sommet de  $G$  traité à l'étape  $k + 1$ . Par l'absurde, supposons qu'il existe un p.c.c.  $P$  de  $r$  à  $x$  dans  $G$  de longueur  $l < d(x)$ . Comme  $r$  a été traité avant l'étape  $k$  et pas  $x$ , il existe un arc  $vu$  de  $P$  tel que  $u$  a été traité avant l'étape  $k$  et pas  $v$ . Comme, par  $\mathcal{H}_k$ , la distance de  $r$  à  $u$  est bien calculée et que l'arc  $vu$  a été possiblement relaxé au moment du traitement de  $u$ , la distance de  $r$  à  $u$  est bien calculée et vaut au plus  $l$ . À l'étape  $k + 1$  on aurait donc  $d(v) = l < d(x)$ , ce qui contredit le choix de  $x$  ligne 7 à l'étape  $k + 1$ .

• **Complexité** : L'initialisation des lignes 2 à 5 demande un temps en  $O(n)$ . La ligne 8 s'exécute  $n$  fois et les lignes 10 à 13 demandent un temps de traitement en  $O(\deg(x)) \leq O(n)$ . Finalement DIJKSTRA prend un temps en  $O(n^2)$ . On peut cependant avoir une implémentation plus fine. Si on gère  $d$  par un tas, alors chaque extraction du minimum ligne 7 et chaque mise-à-jour ligne 11 prend un temps en  $O(\log n)$ . Comme le nombre total de tours de boucle de la ligne 10 est  $\sum \{\deg(v) : v \in G\} = 2m$ , on a un temps d'exécution total en  $O(m \log n)$ .

**Algorithme** : ALGO DE BELLMAN-FORD

**Données** : Un graphe  $D = (V, A)$  orienté donné par liste d'arcs avec une fonction de longueur  $l$  sur les arcs, et  $r$  un sommet de  $D$ .

**Résultat** : Un signalement si  $D$  contient un cycle orienté de longueur totale négative, et sinon une fonction  $d : V \rightarrow \mathbb{R}$  donnant la distance à la racine  $r$  et une fonction  $pere : V \rightarrow V$ .

```

1 début
2   pour tous les  $v \in V$  faire
3      $d(v) \leftarrow +\infty$ ;
4    $pere(r) \leftarrow r$ ;  $d(r) \leftarrow 0$ ; // la racine
5   pour tous les  $i$  de 1 à  $n - 1$  faire
6     pour tous les  $uv \in A$  faire
7       si  $d(v) > d(u) + l(uv)$  alors
8          $d(v) \leftarrow d(u) + l(uv)$ ; //  $u$  est un raccourci pour atteindre  $v$ 
9          $pere(v) \leftarrow u$ ;
10  pour tous les  $uv \in A$  faire
11  si  $d(v) > d(u) + l(uv)$  alors
12   $\leftarrow$  Retourner 'Il existe un cycle orienté de poids  $< 0$ ';
```

**Analyse de BELLMAN-FORD** :

- **Terminaison** : L'algorithme ne contient que des boucles 'pour' et des tests, il termine donc.
- **Preuve** : On rappelle que  $dist_D(r, x)$  désigne la longueur d'un p.c.c. de  $r$  à  $x$  dans  $D$ . Le but de la preuve est de montrer qu'à la fin de l'algorithme, on a pour tout sommet  $x$  de  $D$ ,  $d(x) = dist_D(r, x)$ .

Commençons par prouver par récurrence sur  $p$  la propriété  $\mathcal{H}_p$  : 'Après  $p$  passages dans la boucle ligne 5 à 9, les sommets qui ont un plus court chemin (p.c.c.) depuis  $r$  contenant au plus  $p$  arcs vérifient  $d(x) \leq dist_D(r, x)$ '.

La propriété  $\mathcal{H}_0$  est clairement vraie.

Supposons que  $\mathcal{H}_p$  soit vraie et prenons un sommet  $x$  qui admet un p.c.c.  $P$  depuis  $r$  contenant  $p + 1$  arcs. Notons  $y$  le prédécesseur de  $x$  le long de  $P$ . Le chemin  $P \setminus x$  est un p.c.c. de  $r$  à  $y$  (sinon, on pourrait raccourcir  $P$ ) qui contient  $p$  arcs. Par  $\mathcal{H}_p$ , on a donc  $d(y) \leq dist_D(r, y)$  après  $p$  passages dans la boucle lignes 5 à 9. Au  $p + 1$ -ème passage, l'algorithme met à jour si besoin  $d(x)$  et  $d(x) \leq d(y) + l(yx) \leq dist_D(r, y) + l(yx) = dist_D(r, x)$ . Ainsi,  $\mathcal{H}_{p+1}$  est prouvé, et à la fin de l'algorithme, on a bien  $d(x) \leq dist_D(r, x)$  pour tout sommet  $x$  de  $D$ .

On montre l'inégalité inverse par contradiction : supposons qu'il existe un sommet  $x$  de  $D$  tel qu'à la fin de l'algorithme on ait  $d(x) < dist_D(r, x)$ . Plus précisément, on choisit pour  $x$  le premier sommet

au cours de l'algorithme qui vérifie  $d(x) < \text{dist}_D(r, x)$  et on stoppe l'algo avant la relaxation de l'arc  $yx$  qui amène à cette inégalité. On note  $P$  le chemin  $r, \dots, \text{pere}(\text{pere}(y)), \text{pere}(y), y$ . Si  $x \notin P$ ,  $Px$  est un chemin de  $D$  de longueur  $d(x)$  après la relaxation de  $yx$  et on aurait  $d(x) \leq \text{dist}_D(r, x)$  par la première partie de la preuve, une contradiction. Donc, on a  $x \in P$  et comme il y a eu relaxation de  $yx$ , on a aussi  $d(y) + l(yx) < d(x)$ . Or le long de  $P$ , on a  $d(y) = d(x) + \text{dist}_P(x, y)$  et on obtient  $\text{dist}_P(x, y) + l(yx) < 0$ . Autrement dit, le cycle orienté  $P[x, y] \cup yx$  serait de longueur totale strictement négative, ce qui est exclu.

La preuve de détection de cycle orienté de longueur négative sera vue en cours.

- **Complexité** : Rien de mystérieux, BELLMAN-FORD s'exécute en temps  $O(nm)$ .

**Algorithme** : ALGO DE FLOYD-WARSHALL

**Données** : Un graphe  $D = (V, A)$  orienté donné par liste d'arcs avec une fonction de longueur  $l$  sur les arcs. On prend  $V = \{1, \dots, n\}$ .

**Résultat** : Un signalement si  $D$  contient un cycle orienté de longueur totale négative, et sinon une matrice  $d : V \times V \rightarrow \mathbb{R}$  donnant la distance entre toutes paires de sommets de  $V$  et une matrice  $P : V \times V \rightarrow V$ ,  $P[i][j]$  contenant le début d'un p.c.c. de  $i$  à  $j$ .

```

1 début
2   pour tous les i de 1 à n faire
3     pour tous les j de 1 à n faire
4       si ij ∈ A alors
5         | d[i][j] ← l(ij); P[i][j] ← j;
6       sinon
7         | d[i][j] ← +∞; P[i][j] ← ∅;
8   pour tous les k de 1 à n faire
9     pour tous les i de 1 à n faire
10      pour tous les j de 1 à n faire
11        si d[i][j] > d[i][k] + d[k][j] alors
12          | d[i][j] ← d[i][k] + d[k][j]; // k est un raccourci pour aller de i à j
13          | P[i][j] ← P[i][k];
14   pour tous les i de 1 à n faire
15     si d[i][i] < 0 alors
16       | Retourner 'Il existe un cycle orienté de poids <0';

```

**Analyse de FLOYD-WARSHALL** :

- **Terminaison** : L'algorithme ne contient que des boucles 'pour' et des tests, il termine donc.
- **Preuve** : FLOYD-WARSHALL est un bel exemple de *programmation dynamique*. Montrons par récurrence sur  $k$  la propriété  $\mathcal{H}_k$  : 'À la  $k$ -ème étape, c-à-d après  $k$  tours de la boucle des lignes 8 à 13, les longueurs et débuts des p.c.c. dont les sommets internes sont dans l'ensemble  $\{v_1, \dots, v_k\}$  sont bien calculés'.

La propriété  $\mathcal{H}_0$  est clairement vraie.

Supposons  $\mathcal{H}_k$  vraie et considérons  $P$  un p.c.c. de  $x$  à  $y$  dont les sommets internes sont dans l'ensemble  $\{v_1, \dots, v_{k+1}\}$ . Si  $P$  ne contient pas  $v_{k+1}$  comme sommet interne, alors, par  $\mathcal{H}_k$ , il était bien calculé à l'étape  $k$ , il n'y a plus de relaxation possible le concernant et il reste bien calculé à l'étape  $k + 1$ . Si  $P$  contient  $v_{k+1}$  alors, par  $\mathcal{H}_k$  les longueurs et débuts des p.c.c.  $P[x, v_{k+1}]$  et  $P[v_{k+1}, y]$  étaient bien calculées à l'étape  $k$ . Les mises-à-jour de l'étape  $k + 1$  ligne 12 et 13 permettent de calculer correctement les longueurs et début de  $P$ .

La preuve de détection de cycle orienté de longueur négative sera vue en cours.

- **Complexité** : Rien de mystérieux non plus, FLOYD-WARSHALL s'exécute en temps  $O(n^3)$ .