Algorithmique des graphes

Carla Selmi, département d'informatique, université de Rouen

Sommaire

1. Le problème fondateur : les sept ponts de Könisberg.
2. Graphes : définitions et exemples.
3. Représentations des graphes.
4. Problèmes de cheminement dans un graphe orienté :
   - Accessibilité : l'algorithme de Roy-Warshall.
   - Plus courts chemins et plus courtes distance : l'algorithme de Floyd-Warshall.
   - Plus courts chemins depuis un sommet : les algorithmes de Dijkstra et de Bellman-Ford.
5. Graphes sans cycle : arbres et arborescences.
6. Gestion des partitions d'un ensemble : recherche des composantes connexes par la méthode de l'union et de la recherche.
7. Arbres couvrants minimum : les algorithmes de Kruskal et Prim.
8. Parcours de graphes orientés :
   - L'algorithme d'exploration des graphes.
   - Les différentes stratégies d'exploration.
   - Implantation du parcours en profondeur dans la version avec retour en arrière.
   - Propriétés du parcours en profondeur.
   - Les applications du parcours en profondeur.
9. Flux et réseaux de transport
   - La méthode de Ford-Fulferson.
   - L'algorithme de Ford-Fulkerson.
10. Introduction au langage de programmation Python.

Bibliographie

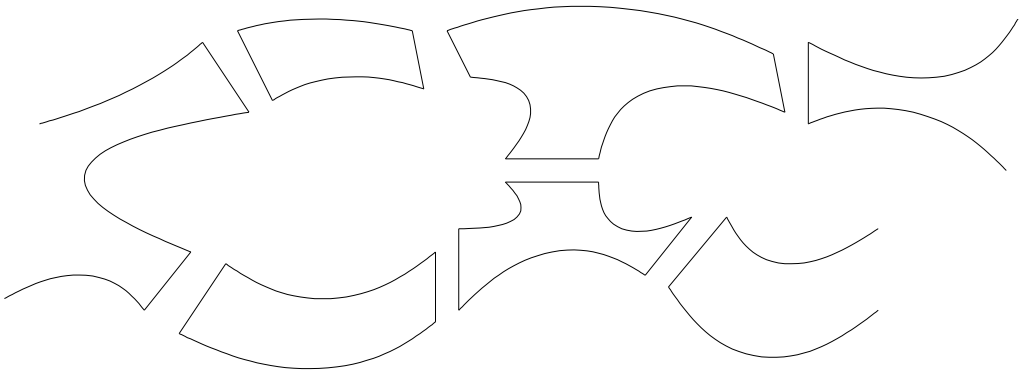
1. Aho A., Ullman J., *Concepts fondamentaux de l'informatique*, Dunod, 1993.
2. Beauquier D., Berstel J., Chretienne P., *Éléments d'algorithmique*, Masson, 1992.
3. Cardon A., Charras C., *Introduction à l'algorithmique*, Ellipses, 1996.
4. Bourgeois N., *Python*, Ellipses 2017.
5. , Cormen T., Leiserson C., Rivest R., *Introduction à l'algorithmique*, Dunon, Paris 1994.
6. La Recherche, *Jeux Mathématiques*, 2000, N. exceptionnel mai-juin 2000.
7. Tangente, L'aventure mathématique, *Les graphes, de la théorie des jeux à l'intelligence artificielle*, 2002-2, Hors Série n. 12.

Webgraphie

1. Tutoriel officiel python : https://docs.python.org/fr/3/tutorial/index.html
2. https://python.sdv.univ-paris-diderot.fr
3. http://www.courspython.com

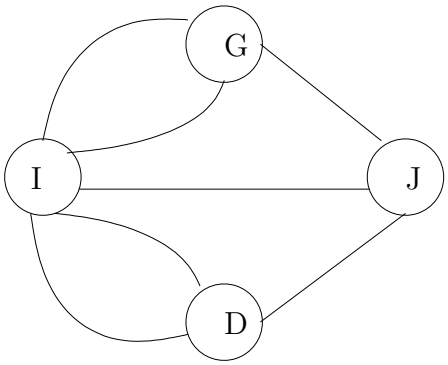
1 Le problème fondateur: le sept ponts de Könisberg

Au $XVII^e$ siècle, un problème se discutait beaucoup chez les habitants de la ville de Könisberg, aujourd'hui Kalingrad (en Russie, proche de la Pologne et de la Lituanie). La ville, construite autour de la rivière Pregel, possédait sept ponts, selon le plan ci-contre.



La question était de savoir comment un touriste pouvait-il visiter la ville en traversant tous les ponts, dans un ordre quelconque sans jamais passer deux fois sur le même pont.

Le mathématicien suisse Leonhard Euler résolut ce problème en utilisant un *modèle mathématique*, que nous appelons aujourd'hui *graphe*. L'idée est la suivante : la ville de Könisberg est la réunion de quatre parties, I, J (les deux îles), G et D (les rives gauche et droite de la rivière Pregel). Les sept ponts, $p_1, \ldots, p_7$, relient entre elles ces parties. La forme de différentes parties de la ville et leurs distances reciproques n'interviennent pas dans la résolution du problème. Nous pouvons donc représenter chaque partie de la ville par un point et chaque pont par une arête. Nous obtenons le représentation suivante :

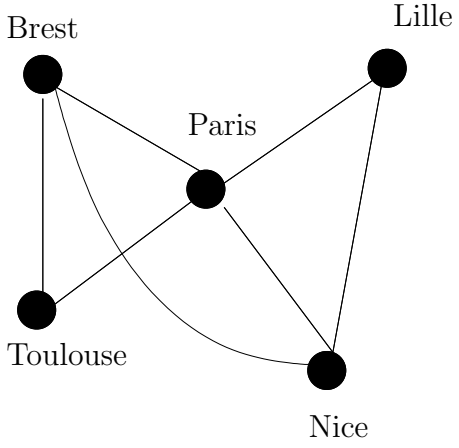


C'est le graphe de Könisberg. Le problème des sept ponts de Könisberg peut se reformuler de la manière suivante :
Comment, en partant de l'un des quatre sommets, peut-on colorier au crayon la totalité des arêtes du graphe sans passer deux fois par la même arête et sans lever

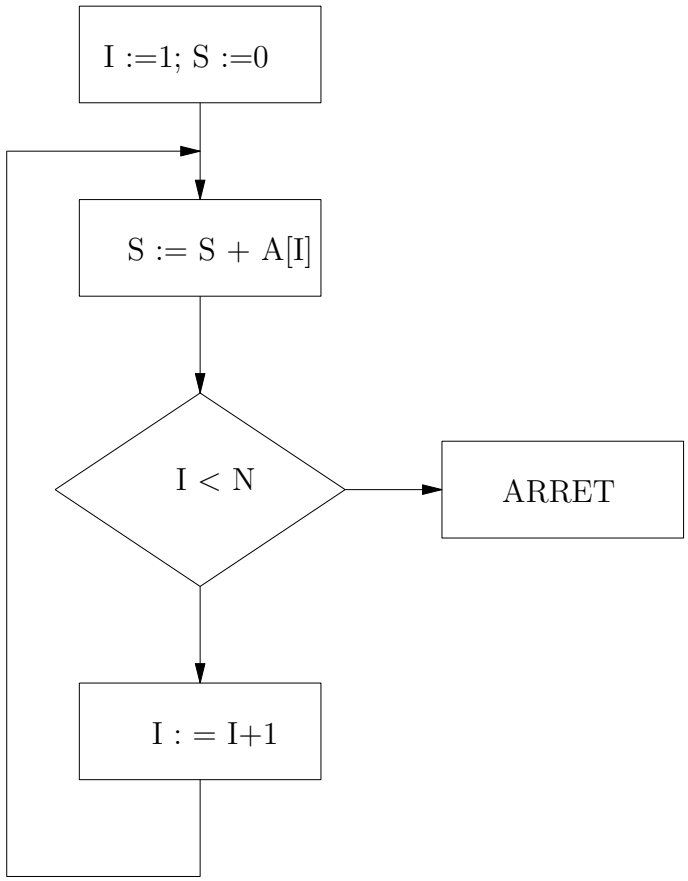
le crayon ?
La solution de ce problème est la caractérisation de l'existence d'un chemin eulérien (un chemin qui parcourt toutes les arêtes du graphe sans passer deux fois par la même arête). Euler a prouvé qu'il suffit de compter le nombre d'arêtes qui sortent de chaque sommet (le degré d'un sommet). S'il y a plus de deux sommets pour lesquels le degré est impair, alors il n'existe pas de chemin eulérien dans le graphe.
Revenons aux graphe de Könisberg. Les quatre sommets qui le composent ont tous degré impair. Il n'existe donc pas de chemin eulérien pour ce graphe. Une visite minimaliste de la ville de Könisberg est donc impossible.
Dans le graphe de Könisberg, les sommets I et G (comme les sommets I et D) sont reliés par plusieurs arêtes. On parle dans ce cas de *multigraphes*.

2 Graphes: définitions et exemples

Beaucoup de problèmes de la vie courante peuvent être représentés par un graphe. Prenons l'exemple d'un graphe de liaisons aériennes. Les villes sont les sommets du graphe. On définit une relation entre les sommets du graphe de la manière suivante : deux sommets A et B sont en relation s'il existe un vol permettant d'aller de la ville représentée par A vers la ville représentée par B. Un graphe permet aussi de représenter une relation binaire dans un ensemble. Les sommets de ce graphe sont les éléments de l'ensemble et une paire de sommets est une arête si les deux sommets qui la composent sont en relation. Deux sommets qui représentent deux villes qui sont en relation sont reliés par une arête. Nous pouvons supposer que la compagnie aérienne assure des vols aller-retour. La relation entre les sommets du graphe est donc symétrique. Dans ce cas, le graphe est dit *non orienté*.

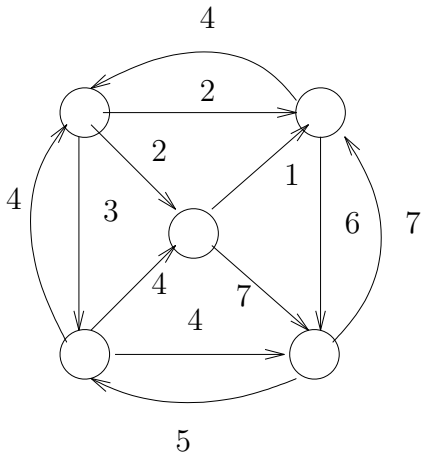


Considérons le graphe associé au programme suivant :



Les sommets de ce graphe sont les instructions du programme. Nous dirons que deux instructions A et B sont en relation si et seulement si A est exécutée immédiatement avant B. Si A et B sont en relation, elles seront reliées par un arc. Cette relation entre les instructions ou tests n'est pas symétrique. En effet, l'instruction $I := I + 1$ s'exécute immédiatement avant $S := S + A[I]$, mais pas l'inverse. Ce graphe est dit *orienté*.

Dans beaucoup de problèmes il est naturel d'associer à chaque arc (ou arête) une valeur (que nous appelerons coût ou poids). Comme dans l'exemple suivant de graphe qui modélise la situation du trafic dans une ville.



Dans cet exemple les coûts sur les arcs indiquent les temps de parcours.

- **Un graphe non orienté** est un couple $G = \langle S, A \rangle$, où S est un ensemble fini de sommets et A est un ensemble fini de paires de sommets, appelés **arêtes**.
- **Un graphe orienté** est un couple $G = \langle S, A \rangle$, où S est un ensemble fini de sommets et A est un ensemble fini de couples, c'est à dire de paires ordonnées, de sommets, appelés **arcs**.
- Un **graphe valué** est un triplet $G = \langle S, A, p \rangle$ où $\langle S, A \rangle$ est un graphe orienté ou non et p est une fonction de A dans $\mathbb{R}$.

3 Représentations des graphes

Dans la suite, un graphe (orienté ou non) sera noté $G = \langle S, A \rangle$ où $S = \{1, \ldots, n\}$ et $A = \{a_1, \ldots, a_m\}$.

Parmi les représentations d'un graphe $G = \langle S, A \rangle$, nous pouvons retenir essentiellement les suivantes :

- *Matrice d'adjacence* $M = M(G)$ est une matrice carrée de booléens de taille n, définie par :

$$M[i,j] = \begin{cases} 1 & \text{si } (i,j) \in A \ (resp. \{i,j\} \in A) \\ 0 & \text{sinon} \end{cases}$$

- *Matrice des distances (ou des poids)* $D = D(G)$ dans le cas d'un graphe G valué. C'est une matrice carrée de réels de taille n, définie par :

$$D[i,j] = \begin{cases} p(i,j) & \text{si } (i,j) \in A \ (resp. \{i,j\} \in A) \\ +\infty & \text{sinon} \end{cases}$$

- *La liste de successeurs* (appelée *liste des voisins* dans le cas d'un graphe non orienté) est un tableau ou une liste de longueur n :

$$Succ = (q_1, \ldots, q_n) \ (\ Vois = (q_1, \ldots, q_n))$$

où q_i est un pointeur sur la liste des successeurs (resp. voisins) du sommet i.

- *Liste d'arcs ou des arêtes* Un graphe orienté (non orienté) peut aussi être représenté par la liste des ses arcs (arêtes).

Notons que la taille de l'espace mémoire occupé par de ces représentations est différente. La taille de l'espace occupé par la représentation par matrice d'adjacence est de l'ordre de $\mathcal{O}(n^2)$, celle par liste des successeurs (des voisins) est de $\mathcal{O}(n+m)$ et celle par liste d'arcs ou (arêtes) (voisins) est de $\mathcal{O}(m)$. La représentation par liste des successeurs permet de traiter plus rapidement des graphes qui ont très peu d'arcs (arêtes) et des graphes dont le nombre de sommets est variable, par l'utilisation des listes chaîneés.

3.1 Structures de données

Les structures des données que nous utiliserons sont les suivantes :

- La matrice d'adjacence est un tableau à deux dimensions de booléens.

$$\begin{array}{lll} Type & GrapheM = & Enregistrement \\ & & n : Entier; \\ & & M : Tableau[1..n, 1..n] \ de \ Booléen \\ & & Fin; \end{array}$$

- La matrice des distances (ou des poids), dans le cas d'un graphe valué, est un tableau à deux dimensions de réels :

$$\begin{array}{lll} Type & GrapheMD = & Enregistrement \\ & & n : Entier; \\ & & D : Tableau[1..n, 1..n] \ de \ Réel \\ & & Fin; \end{array}$$

- La représentation par liste de successeurs est un tableau de listes chaînées.

$$\begin{array}{lll} Type & Cellule = & Enregistrement \\ & & sommet : Entier; \\ & & suiv : \uparrow Cellule \\ & & Fin; \\ & Acellule = & \uparrow Cellule; \\ & GrapheTL = & Enregistrement \\ & & n : Entier; \\ & & T : Tableau[1..n] \ de \ Acellule \\ & & Fin; \end{array}$$

- La représentation par liste d'arcs est une listes chaînées de couples de sommets.

$$
\begin{array}{lll}
Type & Arc = & Enregistrement \\
 & & sommeti : Entier; \\
 & & sommetf : Entier; \\
 & & suiv : \uparrow Arc \\
 & & Fin; \\
 & GrapheLArc = & Enregistrement \\
 & & n : Entier; \\
 & & LA : \uparrow Arc \\
 & & Fin;
\end{array}
$$

- La représentation par liste d'arêtes est une listes chaînées de paires de sommets.

$$
\begin{array}{lll}
Type & Arête = & Enregistrement \\
 & & s1 : Entier; \\
 & & s2 : Entier; \\
 & & suiv : \uparrow Arête \\
' & & Fin; \\
 & GrapheLArête = & Enregistrement \\
 & & n : Entier; \\
 & & LA : \uparrow LArête \\
 & & Fin;
\end{array}
$$

4 Problèmes de cheminement dans un graphe : accessibilité et plus court chemin

Parmi les problèmes de cheminement dans un graphe, nous nous intéresserons à l'existence d'un chemin entre deux sommets d'un graphe orienté ou non et au calcul d'un plus court chemin entre deux sommets, quand celui-ci existe, dans un graphe valué.

4.1 Chaînes, chemins, cycles et circuits

Nous introduisons, ici, les notions fondamentales liées au cheminement dans un graphe. Considérons un arc $a = (x, y) \in A$ (une arête $a = \{x, y\}$) d'un graphe orienté $G = \langle S, A \rangle$ (non orienté). Nous dirons que x est *l'extrémité initiale* de a et y son *extrémité finale* (x et y sont les extrémités de a) et que x et y sont *adjacents*. Nous dirons aussi que x est un *prédécesseur* de y et que y est un *successeur* de x (x et y sont des sommets *voisins*) ou que l'arc a est un *arc sortant* de x (*incident* à x), dit aussi *positif*, et un *arc entrant* en y (*incident* à y), dit aussi *négatif*, (en x et en y).
Deux arcs ou deux arêtes a_1 et a_2 sont *distincts* s'ils ont au moins une extrémité différente. Ils sont *adjacents* s'ils ont une extrémité en commun et ils sont *égaux* s'ils

ont les deux extrémités égales. Deux arcs sont *consécutifs* si l'extrémité finale de a_1 coïncide avec l'extrémité initiale de a_2.
Une *chaîne* de longueur $k \geq 0$ est une suite composée de k arcs ou de k arêtes, deux à deux adjacents. Par convention, une chaîne de longueur zéro est composée d'un seul sommet, nous l'appelerons chaîne triviale. Un *chemin* de longueur k est une suite de k arcs deux à deux consécutifs. Un chemin est une chaîne mais la propriété inverse n'est pas vérifiée. Un chemin ou une chaîne peut se décrire aussi par la suite des sommets par lesquels il passe, $c = (x_0, \ldots, x_k), x_i \in S, 0 \leq i \leq k$. Dans ce cas, nous dirons que c est un chemin de x_0 vers x_k ou une chaîne reliant x_0 et x_k.
Un chaîne $c = (a_1, \ldots, a_k), a_i \in A, 1 \leq i \leq k$, de longueur $k \geq 1$ est un *cycle* si les arcs ou arêtes a_1 et a_k ont une extrémité en commun. Un cycle de longueur 1 est appelée *boucle*. Un cycle qui est aussi un chemin est appelé *circuit*.
Une chaîne est *élémentaire* si les sommets qui la composent sont deux à deux distincts. Un cycle $(x_0, \ldots, x_k), x_i \in S, 0 \leq i \leq k$, avec $x_0 = x_k$, est élémentaire si la chaîne $(x_0, \ldots, x_{k-1})$ est élémentaire.

4.2 Le Lemme de König

Les chemins d'un graphe orienté constituent un ensemble en général infini (pour cela, il faut et il suffit que le graphe contienne un circuit). Parcontre, l'ensemble des chemins élémentaires est, parcontre, toujours fini. Le Lemme de König montre que l'existenxe d'un chemin d'extrémités x et y entraîne l'existence d'un chemin élémentaire ayant les mêmes extrémités.
Soit $c = (x_0, \ldots, x_k), x_i \in S, 0 \leq i \leq k$ un chemin de longueur k d'un graphe orienté $G = \langle S, A \rangle$. Un *chemin extrait* de c est un chemin qui a les mêmes extrémités de c et tel que la suite de sommets dont il est composés, est une sous-suite de la suite de sommets de c.

Exemple 4.1 *Du chemin* $c = (1, 2, 1, 3, 2, 4, 5, 6, 5, 7)$ *nous pouvons extraire le chemin élémentaire* $c' = (1, 3, 2, 4, 5, 7)$.

Le lemme suivant, dit de König, montre que de tout chemin de G,

Lemme 4.2 *Soit* $G = \langle S, A \rangle$ *un graphe orienté. S'il existe un chemin de* x *vers* y, $x, y \in S$, *alors il existe aussi un chemin élémentaire de* x *vers* y.

Preuve. Par récurrence sur la longueur k du chemin c. Si $k = 0$ alors le chemin $c = (x_0)$ est élémentaire. Si $k = 1$ alors $c = (x_0, x_1) \in A$ est élémentaire si $x_0 \neq x_1$. Dans le cas où $x_0 = x_1$, c est un cycle élémentaire car le chemin (x_0) est élémentaire. Considérons maintenant un chemin $c = (x_0, \ldots, x_k), k \geq 2$, de $x_0 = x$ vers $x_k = y$. Si c n'est pas élémentaire alors il existe un couple d'indices $0 \leq r < s \leq k$, tels que $x_s = x_r$. Sans perte de généralité, nous pouvons choisir le plus petit couple (r, s) qui vérifie la propriété $x_s = x_r$. Supposons $r \neq 0$ ou $s \neq k$. Le chemin $c' = (x_0, \ldots, x_r, x_{s+1}, \ldots, x_k)$ est extrait de c et de longueur strictement inférieure à k. Par hypothèse de récurrence, nous pouvons extraire de c', et donc de c, un

chemin élémentaire de x vers y. Dans le cas $r = 0$ et $s = k$, le chemin trivial (x_0) est un chemin élémentaire de x_0 vers x_0.

■

4.3 Accessibilité : l'algorithme de Roy-Warshall

Soit $G = \langle S, A \rangle$ un graphe orienté et soient x et y deux sommets de G. Nous dirons que y est *accessible* à partir de x s'il existe un chemin de x à y.
La relation d'accessibilité est décrite par une matrice carrée de booléens $\mathcal{AC}$ de taille n, dite *matrice d'accessibilité*, définie par

$$\mathcal{AC}[x, y] = \begin{cases} 1 & \text{s'il existe un chemin de x vers y} \\ 0 & \text{sinon} \end{cases}$$

Le sommet y est accessible à partir du sommet x si et seulement si $\mathcal{AC}[x, y] = 1$.
L'algorithme de Roy-Warshall permet de calculer la matrice d'accessibilité $\mathcal{AC}$ à partir de la matrice d'adjacence M de G.

4.3.1 Classement des chemins par niveau

Le niveau d'un chemin $c = (x_0, \ldots, x_k)$ est défini par :

$$Niveau(c) = \begin{cases} Maximum\{x_1, \ldots, x_{k-1}\} & \text{si k } \geq 2 \\ 0 & \text{sinon} \end{cases}$$

Nous appellerons *sommets intérieurs* du chemin c, les sommets $x_1, \ldots, x_{k-1}$.
Le niveau d'un chemin c est donc le maximum de l'ensemble de ses sommets intérieurs quand cet ensemble est non vide et zéro sinon. La notion de niveau permet de définir une suite de relations entre les sommets du graphe, et donc une suite de matrices booléennes.
Pour tout $0 \leq k \leq n$, nous définissons une relation, dans l'ensemble des sommets du graphe, par une matrice carrée des booléens de taille n, notée $R^{(k)}$ et définie par :

$$R^{(k)}[x, y] = \begin{cases} 1 & \text{s'il existe un chemin élémentaire de niveau} \leq \text{k de x vers y} \\ 0 & \text{sinon} \end{cases}$$

Notons que:

- Un chemin élémentaire c de niveau inférieur ou égal à zéro est un chemin qui ne possède pas de sommet intérieur. Un tel chemin est soit le chemin trivial (un sommet est accessible à partir de lui même) soit un arc. Nous avons que

  $$R^{(0)} = M \bigvee \Delta_n$$

  où Δ_n est la matrice identité de dimension $n \times n$ ($\Delta_n[x, y] = 1$ si $x = y$ et zéro sinon) et $\bigvee$ est l'opérateur booléen d'union. Par la suite, nous noterons $\bigwedge$ l'opérateur booléen d'intersection.

- L'ensemble des chemins élémentaires qui ont un niveau inférieur ou égale à n, est l'ensemble des tous les chemins élémentaires de G. La relation définie par $R^{(n)}$ est donc la relation d'accessibilité de G.

$$R^{(n)} = \mathcal{AC}.$$

4.3.2 Relation entre $R^{(k)}$ et $R^{(k+1)}$

Les matrices $R^{(k)}$ et $R^{(k+1)}$ sont reliées par une récurrence, comme le montre la proposition suivante.

Proposition 4.3 *Pour tous $x, y \in S$ et pour tout $0 \leq k \leq n-1$,*

$$R^{(k+1)}[x,y] \ = R^{(k)}[x,y] \bigvee (R^{(k)}[x,k+1] \bigwedge R^{(k)}[k+1,y])$$

Preuve. Tout chemin élémentaire c de x à y de $Niveau(c) \leq k+1$ est soit un chemin éleméntaire de niveau inférieur ou égal à k (cela signifie que $k+1$ n'est pas un sommet intérieur de c) soit il se décompose en un chemin élémentaire c_1 de x vers k de $Niveau(c_1) \leq k$ et d'un chemin élémentaire c_2 de k vers y de $Niveau(c_2) \leq k$ (cela signifie que $k+1$ est un sommet intérieur de c).

D'autre part, supposons qu'il existe un chemin élémentaire c_1 de x vers $k+1$ de $Niveau(c_1) \leq k$ et un chemin élémentaire c_2 de $k+1$ vers y de $Niveau(c_1) \leq k$. Le chemin c obtenu par concatenation de c_1 et de c_2 est un chemin (pas forcement élémentaire) de x vers y de $Niveau(c) \leq k+1$. En vertu du Lemme de König, nous pouvons affirmer qu'il existe un chemin élémentaire de x vers y de niveau inférieur ou égal à $k+1$.

■

Remarque 4.4 *Un chemin élémentaire c de x vers $k+1$ de $Niveau(c) \leq k+1$ est un chemin élémentaire de $Niveau(c) \leq k$ car $k+1$ ne peut pas être un sommet intérieur de c. Nous avons donc,*

$$R^{(k+1)}[x,k+1] \ = \ R^{(k)}[x,k+1]$$

et

$$R^{(k+1)}[k+1,y] \ = \ R^{(k)}[k+1,y].$$

Cela veut dire que la colonne et la ligne d'ordre $k+1$ des matrices $R^{(k)}$ et $R^{(k+1)}$ sont égales. Puisque, par la proposition précédente, seulement ces valers interviennent dans le calcul de $R^{(k+1)}$, une seule matrice de travail est utilisée pour le calcul de $\mathcal{AC}$.

Remarque 4.5 *L'algorithme de Roy-Warhall est basé sur programmation dynamique. Cette technique de programmation, parfois difficile ou impossible a appliquer à des problèmes pour lesquels une solution récursive existe, est la suivante :*

- *Trouver une relation de récurrence.*
- *Au lieu d'un algorithme récursif, construire, par ordre croissant, une table de valeurs en utilisant la relation de récurrence trouvée.*

4.4 L'algorithme de Roy-Warshall

En utilisant les résultats précédents, nous obtenons l'algorithme suivant :

Algorithme *Roy-Warshall*
{**Entrée** : Un graphe G de type GrapheM}
{**Sortie** : La matrice d'accessibilité AC de G}
Var i, j, k : Entier; AC : Tableau$[1 \dots n, 1 \dots n]$ de Booléen;
Début
$AC \leftarrow G.M \bigvee \Delta_{G.n}$;
Pour k **de** 1 **à** $G.n$ **faire**
 Pour i **de** 1 **à** $G.n$ **faire**
  Pour j **de** 1 **à** $G.n$ **faire**
   $AC[i,j] \leftarrow AC[i,j] \bigvee (AC[i,k] \bigwedge AC[k,j])$
Finalgo

L'algorithme de Roy-Warshall calcule la matrice d'accessibilité d'un graphe orienté $G = \langle S, A \rangle$ en $\mathcal{O}(n^3)$ d'opérations élémentaires.

4.4.1 Construction d'un chemin

Pour construire un chemin du sommet x au sommet y, dans le cas où y est accessible à partir de x, nous définissons une suite de matrices d'entiers de taille n de la manière suivante :

$$P^{(0)}[x,y] = \begin{cases} x & \text{si } R^{(0)}_{xy} = 1 \\ 0 & \text{sinon} \end{cases}$$

Pour tout $0 \leq k \leq n-1$:

$$P^{(k+1)}[x,y] = \begin{cases} P^{(k)}[x,y] & \text{si } R^{(k)}[x,y] = 1 \\ P^{(k)}[k+1,y] & \text{si } R^{(k)}[x,y] = 0 \bigwedge (R^{(k)}[x,k+1] \bigwedge R^{(k)}[k+1,y]) = 1 \\ 0 & \text{dans tous les autres cas} \end{cases}$$

L'élément $P^{(k)}[x,y]$ contient le prédécesseur du sommet y sur un chemin élémentaire de x vers y de niveau inférieur ou égale à k, si ce chemin existe et zéro sinon.

Pour repérer un chemin de x vers y, si ce chemin existe, il suffit d'analyser la ligne d'indice x de la matrice $P^{(n)}$.

Exercice 1 *Appliquer l'algorithme de Roy-Warshall sur le graphe orienté suivant, donné par sa matrice d'adjacence*

$$M = \begin{pmatrix} 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 \end{pmatrix}$$

4.5 Plus courts chemins et plus courtes distance: l'algorithme de Floyd-Warshall

Soit $G = \langle S, A, p\rangle$ un graphe orienté et valué par la fonction $p : A \longrightarrow \mathbb{R}$. La représentation d'un graphe valué se fera de la même manière que celle d'un graphe en remplacant les booléens par des réels. La matrice d'adjacence d'un graphe valué, dite aussi *matrice des poids ou des distances*, est définie par :

$$D[ij] = \begin{cases} p(x,y) & \text{si}(i,j) \in A \ (resp.\{i,j\} \in A) \\ +\infty & \text{sinon} \end{cases}$$

Le *poids* d'un chemin $c = (x_0, \ldots, x_k),\ x_i \in S,\ \forall 0 \leq i \leq k$, est défini par

$$p(c) = \begin{cases} 0 & \text{si } k = 0 \\ \sum_{i=1}^{k} p(x_{i-1}, x_i) & \text{sinon} \end{cases}$$

Dans le cas des graphes orientés et valués, le problème de l'accessibilité se transforme en la recherche d'un chemin d'évaluation minimale si ce chemin existe.
Nous appelerons *plus court chemin* de x vers y un chemin de x vers y de poids minimum si ce chemin existe, et son poids la *plus courte distance* entre x et y, noté $\delta(x, y)$.

4.5.1 Existence et propriétés des plus courts chemins

Nous appellerons *circuit absorbant* un circuit de G dont le poids est strictement négatif. La présence d'un circuit absorbant sur un chemin entre deux sommets x et y ne permet pas de définir la plus courte distance de x vers y. En effet, on peut prolonger ce chemin en circulant indéfiniment dans le circuit et ainsi diminuer arbitrairement le poids du chemin.
Pour le calcul des plus courtes distances, nous supposerons que G ne possède pas de circuit absorbant.
La proposition suivante contient des propriétés évidentes des plus courts chemins.

Proposition 4.6 *Soit $G = \langle S, A, p\rangle$ un graphe orienté et valué et soit $c = (x_0, \ldots, x_k),\ x_i \in S,\ \forall\ 0 \leq i \leq k$, un plus court chemin de x_0 vers x_k. Le chemin c vérifie les propriétés suivantes :*

- *On peut extraire de c un plus court chemin élémentaire de x_0 à x_k.*
- *Pour tous les indices $0 \leq i < j \leq k$, on peut extraire de c un plus court chemin élémentaire de x_i vers x_j.*

Pour le calcul des plus courtes distances, nous utilisons une méthode proche de celle utilisée pour le calcul de la matrice d'accessibilité.

Pour tout $0 \leq k \leq n$, notons $PCE_{xy}^{(k)}$ l'ensemble des plus courts chemins élémentaires

de x vers y de G de niveau inférieur ou égal à k et par $CD^{(k)}$ la matrice des plus courtes distances de niveau k.

$$CD^{(k)}[x,y] = \begin{cases} p(c) & \text{si } c \in PCE_{xy}^{(k)} \\ +\infty & \text{si } PCE_{xy}^{(k)} = \emptyset \end{cases}$$

Nous notons que :

- $CD^{(0)}[x,y] = \begin{cases} 0 & \text{si } x = y \\ p(x,y) & \text{si } x \neq y \text{ et } (x,y) \in A \\ +\infty & \text{dans tous les autres cas} \end{cases}$
  La matrice $CD^{(0)}$ est obtenue à partir de la matrice des poids D en remplacant les éléments de la diagonale par zéro.
- $CD^{(n)}[x,y] = \begin{cases} \delta(x,y) & \text{si il existe un chemin de x vers y} \\ +\infty & \text{sinon} \end{cases}$
  La matrice $CD^{(n)}$ est la matrice des plus courtes distances.

4.5.2 Relation entre $CD^{(k+1)}$ et $CD^{(k)}$

Comme pour l'algorithme de Roy-Warshall, nous obtenons une relation de récurrence entre $CD^{(k+1)}$ et $CD^{(k)}$.

Proposition 4.7 *Un plus court chemin élémentaire de niveau inférieur ou égal à $k+1$ de x vers y est soit un plus court chemin élémentaire de niveau k soit il se compose d'un plus court chemin élémentaire de x vers $k+1$ de niveau inférieur ou égal à k et d'un plus court chemin élémentaire de $k+1$ vers y de niveau inférieur ou égal à k. Par conséquent,*

$$CD^{(k+1)}[x,y] = Min\{CD^{(k)}[x,y], CD^{(k)}[x,k+1] + CD^{(k)}[k+1,y]\}.$$

4.6 L'algorithme de Floyd-Warshall

L'algorithme suivant, proposé par Floyd, calcule, en utilisant la proposition précédente, la matrice des plus courtes distances en $\mathcal{O}(n^3)$:

Algorithme *Floyd-Warshall*
{**Entrée** : Un graphe G de type GrapheMD}
{**Sortie** : La matrice CD des plus courtes distances de G }
Var i, j, k : Entier, CD : Tableau$[1 \ldots n, 1 \ldots n]$ de Réel;
Début
Pour i **de** 1 **à** $G.n$ **faire**
Pour j **de** 1 **à** $G.n$ **faire**
Si $i = j$ **alors** $CD[i,j] \leftarrow 0$
Sinon $CD[i,j] \leftarrow G.D[i,j]$;

Pour k **de** 1 **à** $G.n$ **faire**
 Pour i **de** 1 **à** $G.n$ **faire**
  Pour j **de** 1 **à** $G.n$ **faire**
   $CD[i,j] \leftarrow Minimum\{CD[i,j], (CD[i,k]+CD[k,j])\}$
Finalgo

Exercice 2 *Appliquer l'algorithme de Floyd-Warshall sur le graphe orienté et valué suivant, donné par sa matrice des distances*

$$D = \begin{pmatrix} +\infty & 1 & 8 & 3 & 10 \\ +\infty & +\infty & 5 & 2 & +\infty \\ 4 & +\infty & +\infty & +\infty & +\infty \\ +\infty & +\infty & 2 & +\infty & 6 \\ 8 & +\infty & 7 & +\infty & +\infty \end{pmatrix}$$

5 Plus courts chemins depuis un sommet : l'algorithme de Dijkstra

Dans le présent paragraphe, nous étudions le problème des plus courts chemins à origine unique : étant donné un graphe orienté et valué $G = \langle S, A, p\rangle$ et un sommet s dit *sommet source*, déterminer un plus court chemin depuis s vers n'importe quel sommet $v \in S$. Nous avons déjà vu que la présence dans G d'un circuit absorbant entre deux sommets x et y ne permet pas de déterminer la plus courte distance entre x et y. L'algorithme de Dijkstra résout le problème des plus courts chemins à origine unique dans le cas où tous les poids des arcs du graphe sont positifs ou nuls (le graphe ne possède donc pas de circuit absorbant). Il utilise la technique dite de *relâchement*. Pour tout sommet $v \in S$, on mantient à jour un attribut $d[v]$ qui est une borne supérieure pour la plus courte distance de s vers v. L'attribut $d[u]$ est appelé *estimation de la plus courte distance* de s vers u. Dans la suite, pour simplifier l'écriture des l'algorithmes, le sommet source sera $s = 1$.
Au début de l'algorithme nous avons :

$$d[v] = \begin{cases} +\infty & \text{si } v \neq 1 \\ 0 & \text{sinon} \end{cases}$$

Pour la représentation des plus courts chemins, on utilisera le tableau de prédécesseurs π défini par :

$$\pi[v] = \begin{cases} u & u \text{ est le prédécesseur de v sur un plus court chemin de 1 vers v} \\ 0 & \text{si ce prédécesseur n'existe pas} \end{cases}$$

A début de l'algorithme $\pi[v] = 0,\ \forall v \in S$.
Voici la procédure qui permet d'initialiser les tableaux d et π.

Procédure *InitDijkstra*(G : GrapheMD; **Var** d : Tableau$[1 \dots n]$ de Réel; **Var** π : Tableau$[1 \dots n]$ d'Entier) ;
Var u : Entier;
Début
Pour u de 1 à $G.n$ **faire**
Début
$d[u] \leftarrow +\infty$;
$\pi[u] \leftarrow 0$
Finpour
$d[1] \leftarrow 0$
Finprocédure

5.1 La procédure Relâcher et ses propriétés

Le processus de relâchement d'un arc (u, v) est un test permettant de savoir s'il est possible, en passant par u, d'améliorer l'estimation de la plus courte distance de 1 vers v et, si oui, de mettre à jour les tableaux d et π. On rappelle que la plus courte distance entre deux sommet u et v est notée $\delta(u, v)$.

Exemple 5.1 *Supposons que $d[u] = 5$, $d[v] = 9$ et que le poids de l'arc (u, v) soit $p(u, v) = 2$. Le chemin de 1 vers v qui passe par u améliore l'estimation de la plus courte distance de 1 vers v. Cette estimation passe de 9 à $d[u]+p(u, v) = 7$. Dans le cas $d[u] = 5, d[v] = 6$ et $p(u, v) = 2$, l'estimation de la pondération de la plus courte distance de 1 à v ne peut pas être améliorée en passant par u.*

Nous écrivons la procédure *Relâcher*.

Procédure *Relâcher*(G : GrapheMD; u, v : Entier; **Var** d : Tableau$[1 \dots n]$ de Réel **Var** π : Tableau$[1 \dots n]$ d'Entier);
Début
Si $(d[u] + G.D[u, v] < d[v])$ **alors**
Début
$d[v] \leftarrow d[u] + G.D[u, v]$;
$\pi[v] \leftarrow u$;
Finsi
Finprocédure
Les propriétés vérifiées par l'attribut d sont contenues dans la proposition suivante :

Proposition 5.2 *On suppose que les tableaux d et π soient initialisés par la procédure InitDijkstra. L'attribut d vérifie les propriétés suivantes :*

1. *Après l'application de la procédure Relâcher sur l'arc (u, v) de poids $p(u, v)$, on a*
$$d[v] \leq d[u] + p(u, v).$$

2. *Après toute séquence d'application de la procédure Relâcher sur les arcs du graphe, on a pour tout sommet v de S :*

$$d[v] \geq \delta(1, v).$$

De plus, une fois que $d[v]$ atteint sa borne inférieure $\delta(1, v)$, sa valeur n'est plus jamais modifiée par l'application de la procédure Relâcher.

3. *Soit $c = (1, \ldots, u, v)$ un plus court chemin de 1 vers v dont le dernier arc est (u, v). Supposons que $d[u] = \delta(1, u)$ avant l'appel de la procédure Relâcher sur l'arc (u, v). Alors, après l'appel de Relâcher sur l'arc (u, v), nous avons que $d[v] = \delta(1, v)$.*

Preuve.
1. Par définition de la procédure *Relâcher*.
2. Aprés la procédure *InitDijkstra* la propriété $d[v] \geq \delta(1, v)$ est vérifiée. Supposons par l'absurde que v soit le premier sommet pour lequel, après l'application de la procédure *Relâcher* sur l'arc (u, v), la propriété $d[v] \geq \delta(1, v)$ ne soit plus vérifiée. On a alors $d[v] < \delta(1, v)$. Puisque la valeur de $d[v]$ a été modifiée, nous avons, par définition de la procédure *Relâcher*,

$$d[v] = d[u] + p(u, v) < \delta(1, v).$$

C'est à dire, nous obtenons qu'il existe un chemin de 1 vers v dont le poids est strictement inférieur à la plus courte distance, ce qui est impossible. En outre, puisque le poids sont tous positifs ou nuls, une fois que $d[v]$ a atteint sa borne minimale $\delta(1, v)$, sa valeur ne sera plus changée par une application successive de la procédure *Relâcher*.
3. Par la propriété 2, si $d[u] = \delta(1, u)$ avant l'appel de la procédure *Relâcher* sur (u, v), cette valeur est conservée après l'appel. Par la propriété 1, nous avons $d|u] + p(u, v) \geq d[v]$ et par la propriété 2, $d[v] \geq \delta(1, v)$. Donc, après l'appel de la procédure *Relâcher* sur (u, v), les inégalités suivantes sont vérifiées :

$$\delta(1, v) \leq d[v] \leq d|u] + p(u, v) = \delta(1, u) + p(u, v).$$

Puisque $c = (1, \ldots, u, v)$ est un plus court chemin de 1 vers v, on a $\delta(1, u) + p(u, v) = \delta(1, v)$. On obtient, $\delta(1, v) \leq d[v] \leq \delta(1, v)$ ce qui implique $d[v] = \delta(1, v)$.

■

D'après le point 3 de la proposition précédente, si $c = (1, \ldots, u, v)$ est un plus court chemin de 1 vers v et $d[u] = \delta(1, u)$, alors il suffit d'appliquer la procédure *Relâcher* sur l'arc (u, v) pour obtenir $d[v] = \delta(1, v)$. Le problème est maintenant celui de savoir comment déterminer si $d[u] = \delta(1, u)$. La réponse est donnée par l'algorithme de Dijkstra, que nous allons décrire. Nous dirons qu'un sommet u est *ouvert* si $d|u] < +\infty$ et qu'il est *fermé* si la procédure *Relâcher* a été appliquée à tous ses successeurs. Au début, seul le sommet origine 1 est ouvert et aucun sommet n'est fermé. Nous utilisons, dans l'écriture suivante de l'algorithme de Dijkstra,

deux tableaux de booléens O et F pour décrire les sommets ouverts et ceux qui sont fermés, $O[u] = vrai$ si et seulement si u est ouvert et $F[u] = vrai$ si et seulement si u est fermé.

Algorithme *Dijkstra*;
{**Entrée** : Graphe orienté et valué G de type GrapheMD de sommet source $s = 1$ }
{**Sortie** : Tableaux d et π}
Var u, v, i : Entier; O, F : Tableau$[1 \dots n]$ de Booléen;
Début
Pour i de 1 à $G.n$ **faire**
Début
$F[i] \longleftarrow faux$;
$O[i] \longleftarrow faux$;
Finpour;
InitDijkstra(G, d, π);
$O[1] \longrightarrow vrai$;
Pour i de 1 à $G.n$ **faire**
Début
{Boucle principale}
Choisir un sommet u : $O[u]$ *et* $non(F[u])$ tel que $d[u]$ soit minimal;
Pour tout successeur v de u tel que $non(F[v])$ **faire**
Début
$O[v] \longleftarrow vrai$;
Relâcher(G, u, v, d, π);
Finpour;
$F[u] \longleftarrow vrai$;
Finpour
Finalgo

La proposition suivante prouve la validité de l'algorithme de Dijkstra.

Proposition 5.3 *Le sommet u choisi à chaque itération de la boucle principale de l'algorithme de Dijkstra, est tel que la valeur $d[u]$ est la plus courte distance entre le sommet origine 1 et u, c'est à dire $d[u] = \delta(1, v)$.*

Preuve. La preuve se fait par récurrence sur le nombre k d'itérations de la boucle principale.

- $k = 0$: Le seul sommet ouvert et pas fermé est le sommet origine 1 dont l'attribut vaut $d[1] = 0$. Puisque tout circuit est de poids positif ou nul, la plus courte distance de 1 vers 1 vaut zéro.

- Hyp. de récurrence : supposons la propriété vraie pour tout sommet choisi avant la k-ème itération de la boucle principale. Soit u le sommet choisi à la k-ème itération et soit F l'ensemble des sommets fermés avant la k-ème itération.

Comme tout sommet $t \in F$ a été choisi lors d'une itération précédente, par hypothèse de récurrence, on a que $d[t] = \delta(1,t)$. Par le point 2 de la Proposition 5.2, cette valeur ne changera plus. Soit $c = (1,\ldots,y,x\ldots,u)$ un chemin quelconque de 1 vers u et soit (y,x) le dernier arc de ce chemin tel que $y \in F$ et $x \notin F$. Comme y est fermé, la procédure Relâcher a été appliquée sur l'arc (y,x) et donc nous avons $d[x] \leq d[y] + p(y,x) \leq \delta(1,y) + p(y,x)$. Nous pouvons conclûre que $p(c) \geq \delta(1,y) + p(y,x) \geq d[x]$. D'autre part, le choix de u implique que $d[u] \leq d[x]$. On en déduit, $p(c) \geq d[u]$, pour tout chemin c de 1 vers u. Il en résulte que $d[u] = \delta(1,u)$.

■

On obtient une implantation efficace de l'algorithme de Dijkstra en utilisant, pour gérer les sommets ouverts et pas fermés, un tas dont la priorité est donnée par la valeur minimale de l'attribut d. Avec cette implantation, l'algorithme de Dijkstra a une complexité de l'ordre $\mathcal{O}(nlog_2(n))$.

Exercice 3 *Appliquer l'algorithme de Dijkstra sur le graphe de l'Exercice 2.*

Remarque 5.4 *L'algorithme de Dijkstra ne fonctionne pas dans le cas de présence des poids négatifs même si le graphes d'entrée G ne possède pas de circuits de poids absorbants. Pour vous en convaincre, appliquez l'algorithme de Dijkstra au graphe dont la matrice des poids est :*

$$D = \begin{pmatrix} +\infty & 3 & 4 \\ +\infty & +\infty & +\infty \\ +\infty & -2 & +\infty \end{pmatrix}$$

L'algorithme de Bellman-Ford utilise les mêmes structures des données et la procédure Relâcher utilisées dans l'algorithme de Dijkstra, pour calculer les plus courtes distances et les plus courts chemins d'un graphe orienté et valué G avec une fonction de poids quelconque.

Algorithme *Bellman-Ford*;
{**Entrée** : Graphe orienté et valué G de type GrapheMD de sommet source 1 }
{**Sortie** : Les tableaux d et π}
Var u, v, k : Entier;
Début
InitDijkstra(G, d, π);
Pour k de 1 à $G.n - 1$ **faire**
{Répéter $G.n - 1$ fois}
Pour u de 1 à $G.n$ **faire**
Pour v de 1 à $G.n$ **faire**
Si $G.D[u, v] \neq +\infty$ **alors**
{Pour tout arc (u, v) de G}
Relâcher(G, u, v, d, π);
Finalgo

Cet algorithme peut aussi tester la présence de circuits absorbants dans le graphe d'entrée G. Il suffit, à la sortie de la boucle principale, d'exécuter encore une fois l'instruction :
Pour tout arc $(u, v) \in A$ **faire** *Relâcher*(G, u, v, d, π);
Si un changement se produit, le graphe possède un circuit absorbant.

6 Le tris par tas

Le tas est une implantation efficace de la structure de données *file de priorité*. Une file de priorité généralise la structure de données *file d'attente*. Dans une file d'attente le premier arrivé est le premier servi ("first in, first out", en anglais). Dans une file de priorité, le premier servi est le premier arrivé parmi ceux qui ont la plus forte priorité. Un élément d'une file de priorité est représenté par un couple (p, t) où p est la priorité et t est le temps d'attente. Les éléments d'une file de priorité vérifient une relation d'ordre total, définie par :

$$(p, t) > (p', t') \text{ si } p > p' \text{ et } (p, t) \geq (p', t') \text{ si } p = p' \text{ et } t \geq t'.$$

6.1 La structure de données tas

Un tas est un tableau $T[1 \dots n], n \geq 1$, dont les éléments appartiennent à un ensemble totalement ordonné, par exemple les réels, vérifiant la propriété :

$$(*)(T[i] \geq T[2i]) \text{ et } (T[i] \geq T[2i+1]), \forall 1 \leq i \leq \lfloor n/2 \rfloor,$$

où $\lfloor n/2 \rfloor$ vaut la partie entière inférieure de $n/2$.

Remarque 6.1 *Dans cette implantation d'une file de priorité, la priorité de l'élément i est $p = T[i]$ et son temps d'attente est de $n - t$.*

Exemple 6.2 *Le tableau suivant n'est pas un tas car $T[3] < T[7]$.*

12	*9*	*6*	*6*	*8*	*5*	*7*	*2*	*3*
1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*

Pour qu'il devienne un tas, il suffit de changer $T[3]$ en lui donnant, par exemple, la valeur 7.

6.2 Représentation d'un tas

Nous pouvons représenter un tas $T[1 \dots n]$ par une arborescence étiquettée $H_T = (S, A, r)$ où $S = \{1, \dots, n\}$ est l'ensemble de sommets, $r = 1$ est la racine, l'étiquette du sommet i vaut $T[i], \forall 1 \leq i \leq n$ et l'ensemble d'arcs est $A = \{(i, 2i), (i, 2i+1) \mid 1 \leq i \leq [n/2]\}$.
La notion d'arborescence ainsi que celle d'arbre et le vocabulaire utilisé sont définies dans la section "Graphes sans cycles : arbres et arborescences".

Remarque 6.3 *L'arborescence associée à un tas est appelée aussi "arbre binaire" car c'est une arborescence où chaque sommet possède au plus deux successeurs. Un arbre binaire peut être défini aussi par récurrence de la manière suivante :*

- *L'arbre binaire vide est noté Nil. Il ne possède aucun sommet.*

- (r, A_g, A_d) *est un arbre binaire où* r *est la racine et* A_g *et* A_d *sont deux arbres binaires, appellés sous-arbre gauche et sous-arbre droit. Les racines des arborescences* A_g, *et* A_d *sont les successeurs de* r.

Nous utiliserons les notions de hauteur d'une arborescence et d'arborescence complète pour évaluer la complexité du tri par tas.

Definition 6.4 *Une arborescence binaire est complète si tout sommet qui n'est pas une feuille possède exactement deux successeurs.*

Definition 6.5 *La hauteur d'une arborescence est la longueur du plus long chemin qui relie la racine à une feuille.*

Les notions précédentes sont reliées par la proposition suivante :

Proposition 6.6 *Le nombre de sommets d'une arborescence binaire de hauteur* h *est* $h = \lfloor log_2 n \rfloor$.

Preuve : Le nombre de sommets n d'une arborescence binaire de hauteur h peut être encadré de la manière suivante :

$$\sum_{i=0}^{h-1} 2^i + 1 \leq n \leq \sum_{i=0}^{h} 2^i = 2^{h+1} - 1 < 2^{h+1}.$$

Nous obtenons, $h \leq log_2 n < h + 1$, d'où $h = \lfloor log_2 n \rfloor$.

■

Exercice 4 *Dessinez l'arborescence qui représente le tas suivant :*

12	*9*	*7*	*6*	*8*	*5*	*7*	*2*	*3*
1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*

Dans de nombreuses applications, il est nécessaire de faire la différence entre les éléments qui font partie du tas (par exemple, les villes d'un réseau routier) et les priorités qu'ils leur sont attribuées (par exemple, les plus courtes distances de ces villes depuis un point fixé du réseau). La structure de données tas s'enrichie en devenant :

$$\begin{array}{ll} Type\ Tas = & Enregistrement \\ & e : Tableau[1..n]\ d'Elément; \\ & p : Tableau[1..n]\ de\ Réel; \\ & ne : Entier \\ & Fin \end{array}$$

Le champ *ne* de l'enregistrement du type *tas* représente le nombre d'élements qui font partie du tas. Ce nombre est inférieure ou égale au nombre maximal d'éléments n que le tas peut contenir.

6.3 Opérations sur le tas

Les deux principales opérations que nous pouvons efféctuer sur un tas T sont celles d'insertion d'un élément et de suppression de la racine. Dans le deux cas, le résultat des opérations doit être un tas. Cela signifie que le tableau des priorités $T.p$ doit vérifier la propriété $(*)$.

Definition 6.7 *Un élément e d'un tas T est à sa bonne place si c'est une feuille ou si sa priorité est supérieure ou égale à celles des ses fils.*

- **Insertion** Pour insérer un élément e de priorité p dans un tas T, on l'ajoute d'abord à la fin du tas et on le fait remonter, ensuite, vers la racine du tas, jusqu'à ce qu'il se trouve à sa bonne place.

  Exercice 5 *Insérer l'élément de priorité* 11 *dans le tas de l'exercice précédent.*

  Nous écrivons, en premier lieu, la fonction *BienplacéR*. Cette fonction teste si l'élément d'indice i est bien placé dans la remontée de T. Nous dirons qu'un élément d'indice i d'un tas T est bien placé dans la rémontée si et seulement si c'est la racine de T $(i = 1)$ ou si sa priorité est inférieure ou égale à celle de son père $(T.p[i] \leq T.p[[i/2]])$.

  Fonction $BienplacéR(i : Entier; T : Tas)$: Booléen;
  Var *Rép* : Booléen;
  Début
  $Rép \leftarrow faux$;
  Si $(i = 1)$ **alors** $Rép \leftarrow vrai$
  Sinon
  Si $(T.p[i] \leq T.p[[i/2]])$ **alors** $Rép \leftarrow vrai$;
  $BienplacéR \leftarrow Rép$
  Finfonction

  Nous utilisons la fonction *BienplacéR* pour écrire la procédure $InsérerT$, qui insère un élément e de priorité p à sa bonne place dans un tas T.

  Procédure $InsérerT(e : Elément; p : Réel;$ **Var** $T : Tas)$;
  Var i : Entier;
  Début
  {On insère l'élément e de priorité p à la fin du tableau}
  $i := T.ne + 1$;
  $T.ne \leftarrow i$;
  $T.e[i] \leftarrow e$;
  $T.p[i] \leftarrow p$;
  Tantque $(NonBienplacéR(i, T))$ **faire**
  Début

{Si e n'est pas à sa bonne place, on l'echange avec son père}
Echanger$(T.p[i], T.p[[i/2]])$;
Echanger$(T.e[i], T.e[[i/2]])$;
$i \leftarrow \lfloor i/2 \rfloor$;
Fintantque
Finprocédure

- **Suppression de la racine** Pour supprimer la racine d'un tas, nous procédons de la manière suivante :
  - On remplace la racine avec le dernier élément.
  - On supprimer le dernier élément du tas.
  - On fait descendre la racine à sa bonne place dans les tas, en l'échangeant, à chaque fois, avec le fils dont la priorité est la plus élévée.

Exercice 6 *Supprimer la racine du tas de l'exercice 4.*

Comme pour l'insertion, nous pouvons écrire la fonction *BienplacéD* qui vérifie si l'élément d'indice i est bien placé dans la descente de T. Nous dirons qu'un élément e d'indice i est bien placé dans la descente si et seulement si :

- e n'a pas de fils ($2i > n$).
- e a un seul fils ($2i = 1$) et sa priorité est supérieure ou égale à celle de son fils ($T.p[i] \geq T.p[2i]$).
- e a deux fils et sa priorité est supérieure ou égale à la plus élévée des priorités des ses fils ($T.p[i] \geq T.p[2i]$ et $T.p[i] \geq T.p[2i+1]$).

Exercice 7 *Ecrire la fonction BienplacéD.*

Nous pouvons ensuite, en utilisant la fonction *BienplacéD*, écrire la procédure *DescendreT*. Cette procédure prend en entrée deux entiers $i \leq j$ et un tas T et modifie T en faisant descendre l'élément d'indice i à sa bonne place jusqu' à l'indice j du tas T. Nous écrivons maintenant la procédure qui permet de supprimer la racine d'un tas non vide.

Procédure *SupprimerR*(**Var** $T : Tas$)**;**
Var k **: Entier;**
Début
$k \leftarrow T.ne$**;**
$T.e[1] \leftarrow T.e[k]$**;**
$T.p[1] \leftarrow T.p[k]$**;**
$T.ne := k - 1$**;**
DescendreT$(1, T.n_e, T)$**;**
Finprocédure

Il est clair que le nombre d'opérations effécutées par les procédures *InsérerT*, *DescendreT* et *SupprimerR* est directement proportionnel à la hauteur de l'arborescence binaire qui représente le tas. Nous obtenons, comme corollaire de la Proposition 6.6, que leur complexité en temps est de l'ordre $\mathcal{O}(log_2 n)$.

6.4 Le tri par tas

En utilisant la structure de tas, on peut obtenir un tri d'un tableau contenant n éléments en une complexité de l'ordre $\mathcal{O}(nlog_2 n)$. Le tris par tas fonctionne comme un tri par selection du maximum, que nous rappellons ici.
Considérons un tableau $A = A[1 \cdots n]$ et un partage de rang i, $1 \leq i \leq n$, de A. C'est à dire une partition de A vérifiant :

1. $B = A[1 \cdots i]$;
2. $C = A[i+1 \cdots n]$;
3. C est trié en ordre croissant;
4. Les éléments de C sont tous supérieurs ou égaux aux éléments de B.

Pour etendre la partie triée de A d'un cran vers la gauche, $C = A[1+1 \cdots n]$, à la partie $A[i \cdots n]$, on cherche l'indice du maximum, i_{max}, des éléments de B et on échange $A[i_{max}]$ et $A[i]$. On obtient ainsi un partage de rang $i-1$ de A. En répétant cette modification du tableau A pour tout i allant de n à 1, on obtient le tri de A par selection du maximum.

Algorithme *Triparmaximum*
{**Entrée** : Un tableau $A[1 \cdots n]$ de taille n}
{**Sortie** : Le tableau A trié dans l'ordre croissant}
Var i, i_{max} : Entier;
Début
$\{B = A, C = \emptyset\}$
$i \leftarrow n$;
Tantque $(i > 1)$ **faire**
Début
$i_{max} \leftarrow IndiceMax(A[1 \cdots i])$;
{$IndiceMax$ est une fonction qui retourne l'indice du maximum de $A[1 \cdots i]$}
Echanger$(A[i_{max}], A[i])$;
$i \leftarrow i - 1$;
Fintantque
Finalgorithme

Supposons maintenant que, outre les quatre propriétés qui doivent être vérifiées par un partage de rang i d'un tableau A, on impose au tableau $B = A[1 \cdots i]$ la structure de tas. Il est alors inutile de chercher l'indice du maximum de B car il

se trouve toujours à l'indice $i = 1$. Après avoir échanger $A[1]$ et $A[i]$, le tableau B n'est plus un tas à cause de $= A[1]$. Pour que B devienne un tas il faut faire descendre l'élément $A[1]$ à sa bonne place dans B. Nous obtenons ainsi l'algorithme de tri suivant :

Algorithme *Tripartas*
{**Entrée** : Un tableau $A[1 \cdots n]$ de taille n}
{**Sortie** : Le tas T dont le champ p contient les éléments de A triés dans l'ordre croissant}
Var i : Entier;
Début
Entasser(A, n, T);
{Cette procédure, décrite dans le paragraphe suivant, transforme le tableau A de taille n en tas}
$i \leftarrow n$;
Tantque $(i > 1)$ **faire**
Début
Echanger$(T.p[1], T.p[i])$;
Echanger$(T.e[1], T.e[i])$;
DescendreT$(1, i, T)$;
$i \leftarrow i - 1$;
Fintantque
Finalgorithme

Dans l'algorithme *Tripartas*, on exécute une fois la procédure *Entasser* et n fois la procédure *DescendreT*. Sa complexité est donc le maximum entre $\mathcal{O}(nlog_2n)$ et la complexité de la procédure *Entasser*, que nous analysons dans le paragraphe suivant.

6.5 La procédure Entasser

Pour transformer un tableau A de taille n en tas T, nous utilisons la procédure *InsérerT* pour tous les éléments de A.

Procédure *Entasser*$(A : Tableau[1 \ldots n]; n : Entier; \textbf{Var } T : tas)$;
Var i : Entier;
Début
Tantque $(i \leq n)$ **faire**
Début
InsérerT$(A[i], T)$;
$i \leftarrow i + 1$;
Fintantque
Finprocédure

Exercice 8 *Exécuter le tri par tas sur le tableau*

3	*7*	*5*	*8*	*2*	*1*	*12*	*4*	*5*
1	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*

7 Graphes sans cycle : arbres et arborescences

Nous considérons dans ce paragraphe certains graphes non orientés : les arbres et les arborescences.

7.1 Composantes connexes

Soit $G = \langle S, A\rangle$ un graphe non orienté. Nous définissons, sur l'ensemble des ses sommets S, une relation d'équivalence C par :

$x\ C\ y$ si et seulement si $x = y$ ou s'il existe une chaîne reliant les sommets x et y.

La relation C détermine une partition de l'ensemble de sommets S. Cela signifie qu'il existe $k \geq 1$ sous-ensembles de S, $C_1, \ldots, C_k$ qui vérifient :

- $S = \bigcup_{i=1}^{k} C_i$
- $C_i \bigcap C_j = \emptyset, \forall i \neq j$

Les sous-ensembles C_i de S s'appellent les *composantes connexes* de G. Un graphe est dit *connexe* s'il possède une seule composante connexe. Le problème de déterminer les composantes connexes de G sera traité dans la section suivante.

7.2 Arbres

Un *sous-graphe* d'un graphe $G = \langle S, A\rangle$ est un graphe $G' = \langle S', A'\rangle$ où $S' \subset S$ et $A' \subset A$.
Un graphe non orienté $G = \langle S, A\rangle$ est *sans cycle* si aucun de ses sous-graphes n'est un cycle. Un graphe non orienté connexe et sans cycle est appelé *arbre*. La connexité d'un graphe G est équivalente à l'existence d'une chaîne reliant x à y pour toute paire de sommets $\{x, y\}$ de G. L'absence de cycle se traduit par l'unicité de cette chaîne. Nous avons la proposition suivante.

Proposition 7.1 *Un graphe non orienté $G = \langle S, A\rangle$ est un arbre si et seulement si pour toute paire de sommets $\{x, y\}$ il existe une unique chaîne reliant x et y.*

Nous allons à présent étudier l'effet produit sur le nombre de composantes connexes d'un graphe non orienté $G = \langle S, A\rangle$, par la suppression ou par l'adjonction d'une arête.

7.3 Suppression d'arêtes et composantes connexes

Soit $G = \langle S, A\rangle$ un graphe non orienté. Pour tout arête $u = \{x, y\}$ dont les deux extrémités appartiennent à S, nous noterons $G - u$ le graphe $G - u = \langle S, A/\{u\}\rangle$. Une arête $u \in A$ est appelée *isthme* si le nombre des composantes connexes du graphe $G - u$ est strictement supérieur au nombre des composantes connexes de G.

Proposition 7.2 *Une arête $u \in A$ de G est un isthme si et seulement si elle n'appartient à aucun cycle de G.*

Preuve. Soit $u = \{x_i, x_{i+1}\}$ un isthme. Supposons qu'il fasse partie d'un cycle, $c = (x_0, \ldots, x_i, x_{i+1}, \ldots, x_0)$. Dans le graphe $G - u$, les sommets x_i et x_{i+1} restent liés par la chaîne $c' = (x_{i+1}, \ldots, x_0, \ldots, x_i)$ extraite de c. $G - u$ et G ont donc le même nombre de composantes connexes. Nous obtenons une contradiction avec le fait que u soit un isthme.

Supposons maintenant que l'arête $u = \{x_i, x_{i+1}\}$ ne fasse pas partie d'aucun cycle de G, alors toute chaîne reliant x_i et x_{i+1} passe par u. Cela implique que dans le graphe $G - u$, les sommets x_i et x_{i+1} ne sont reliés par aucune chaîne. Le nombre de composantes connexes de $G - u$ est donc strictement supérieur à celui de G. L'arc u est un isthme.

■

Nous obtenons la caractérisation suivante d'un arbre.

Corollaire 7.3 *Un graphe non orienté et connexe G est un arbre si et seulement si chaqu'une de ses arêtes est un isthme.*

7.4 Adjonction d'arêtes et composantes connexes

Soit $G = \langle S, A \rangle$ un graphe non orienté. Soient $x, y \in S$ tel que $\{x, y\} \notin A$. Notons $u = \{x, y\}$ et $G + u$ le graphe $G + u = \langle S, A \cup \{u\} \rangle$. Deux cas mutuellement distincts se présentent :

- Les sommets x et y appartiennent à deux composantes connexes différentes de G. L'adjonction de u a comme effet de fusionner ces deux composantes. Le graphe $G + u$ a un nombre de composantes connexes strictement inférieur à celui de G et donc u est un isthme dans $G + u$.

- Les sommets x et y appartiennent à la même composante connexe de G. L'adjonction de u n'a aucun effet sur le nombre de composantes connexes, G et $G + u$ ont le même nombre de composantes connexes. Puisque x et y font partie de la même composante connexe de G, il existe une chaîne $c = (x, \ldots, y)$ de G les reliant. La chaîne c' obtenue par concatenation de c avec u est un cycle de $G + u$. Ainsi, le nombre de cycles de $G + u$ est strictement supérieur à celui de G.

Proposition 7.4 *Soit $G = \langle S, A \rangle$ un graphe non orienté ayant n sommets et m arêtes. Les propriétés suivantes sont vérifiées*

1. *Si G est connexe alors $m \geq n - 1$.*
2. *Si G est sans cycles alors $m \leq n - 1$.*

Preuve. Nous pouvons construire le graphe G en partant du graphe sans arêtes, appelé graphe discret et noté $G_0 = \langle S, \emptyset \rangle$ en procédant par adjonctions successives des arêtes de G, $A = \{u_1, \ldots, u_m\}$. Cela donne une suite de graphes : $G_0, \ldots, G_m$ telle que $G_i = G_{i-1} + u_i$ pour tout $1 \leq i \leq m$. Les deux propriétés de la proposition suivent du fait que l'adjonction d'une arête a comme effet, soit de réunir deux composantes connexes distinctes soit de créer un cycle.

■

Corollaire 7.5 *Si $G = \langle S, A \rangle$ est un arbre alors $m = n - 1$.*

Exercice 9 *Prouver qu'un graphe non orienté possédant $n - 1$ arêtes, la connéxité implique l'absence de cycles et l'absence de cycles la connexité.*

7.5 Arborescences

Une arborescence est un couple (H, r) où H est un arbre et r est un sommet de cet arbre, appelé *racine*. Une arborescence peut-être vue comme un graphe orienté, en remplaçant chaque arête $\{u, v\}$ par l'arc (u, v) si la chaîne qui relie r à v passe d'abord par u et par l'arc (v, u) sinon. Une arborescence de racine r est alors un graphe orienté dans lequel il existe un chemin de r vers n'importe quel autre sommet. Une des propriétés les plus importantes des arborescences est leur structure récursive.

Proposition 7.6 *Soit (H) un'arborescence de racine r et soient $r_1, \ldots, r_k$ les successeurs de r. Si l'on supprime de H les arcs $(r, r_i), \forall 1 \leq i \leq k$, les composantes connexes du graphe ainsi obtenu, sont les k sou-arborescences de H de racines $r_1, \ldots, r_k$.*

Dans une arborescence (H, r), pour tout sommet s, on appelle *prédécesseur* de s l'unique sommet qui précéde s sur le chemin qui relie la racine r à s. La racine r de G est le seul sommet sans prédécesseur. Un sommet sans succeseurs est une *feuille* et un sommet qui possède au moins un successeurs est un *noeud*. La hauteur d'une arborescence G, noté *hauteur*(G), est le nombre d'arêtes contenues sur un plus long chemin, en nombre d'arêtes, reliant la racine à une feuille.

8 Gestion des partitions d'un ensemble : recherche des composantes connexes par la méthode de l'union et de la recherche (union-find)

Dans cette section, nous considérons le problème de la gestion efficace d'une partition d'un ensemble. Ce problème est appelé le problème de l'union et de la recherche ("union-find" en anglais). Il apparaît dans de nombreuse situations, notamment dans la recherche des composantes connexes et d'un arbre couvrant minimun d'un graphe non orienté et valué.

8.1 Cas statique

Nous avons déja rencontré un algorithme, celui de Roy-Warshall, qui permet de vérifier si la version non orienté d'un graphe orienté est connexe. Pour cela il suffit de vérifier que la matrice d'accessibilité ne contiennent que des 1. Nous pouvons aussi, dans le cas où le graphe n'est pas connexe, utiliser l'algorithme de Roy-Warshall, pour calculer les composantes connexes (faire par exercice).
L'algorithme de Roy-Warshall résoud le problème de la recherche des composantes connexes d'un graphe à n sommets avec une compléxite de $\mathcal{O} = (n^3)$ dans le cas statique où le nombre des arêtes du graphe n'évolue pas.

8.2 Cas évolutif

Une *forêt d'arborescences* est un graphe T dont les composantes connexes sont des arborescences. Si un graphe $G = \langle S, A\rangle$ a le mêmes composantes connexes de T, nous dirons que la forêt d'arborescences T représente la *partition en composantes connexes* de G. Chaque composante connexe de G peut être représentée par la racine de l'arborescence de T appartenant à cette composante. Le problème de décider si deux sommets $x, y \in S$ appartiennent à la même composante connexe, revient à tester l'égalité entre les racines des arborescence auxquelles ils appartiennent.
Considérons le cas évolutif où le graphe $G = \langle S, A\rangle$ est construit à partir du graphe discret $G_0 = \langle S, \emptyset\rangle$ en procédant par adjonctions successives des arêtes $A = \{u_1, \ldots, u_m\}$. Cela donne une suite de graphes : $G_0, \ldots, G_i, \ldots, G_m$ telle que $G_m = G$ et $G_i = G_{i-1} + u_i$ pour tout $1 \leq i \leq m$.
Nous pouvons associer à cette suite de graphes, une suite de forêts d'arborescences : $T_0, \ldots, T_i, \ldots, T_m$, où chaque T_i est une forêt d'arborescences qui représente la partition en composantes connexes du graphe G_i. Pour detéterminer T_i à partir de T_{i-1}, nous observons que les composantes connexes de G_i peuvent être obtenues par regroupement des composantes connexes des graphes G_{i-1}. Comme nous avons $G_i = G_{i-1} + u_i$, $u_i = \{x_i, y_i\}$, les problèmes qui se posent sont :

- Décider à quelles composantes connexes, C_{x_i}, C_{y_i}, de G_{i-1} appartiennent les sommets x_i et y_i.
- Réunir C_{x_i}, C_{y_i} si $C_{x_i} \bigcap C_{y_i} = \emptyset$.

Pour représenter une forêt d'arborescences, nous utilisons un tableau d'entiers, le tableau des prédécesseurs π, défini par

$$\pi[x] = \begin{cases} -1 & \text{si le sommet } x \text{ est une racine} \\ \text{le prédécesseur de } x \text{ dans la forêt} & \text{sinon} \end{cases}$$

8.2.1 L'algorithme Composantes_connexes

Représentation de T_0 : dans la forêt discrète, chaque arborescence est composée d'un seul sommet, sa racine. Donc le tableau π qui représente T_0 est défini par : $\pi[x] = -1$ pour tout $x \in S$.

Construction de T_i : pour construire T_i à partir de T_{i-1}, nous définissons la fonction *Racine* et la procédure *Réunir*.

La fonction *Racine* permet de trouver la composante connexe de G_{i-1} à laquelle un sommet x appartient.

Fonction *Racine*(x : Entier; π : Tableau$[1, \ldots, n]$ d'Entier) : Entier;
{Renvoie la racine de l'arborescence à laquelle x appartient dans la forêt représentée par le tableau π}
Var i : Entier;
Début
$i \leftarrow x$;
Tant que $\pi[i] > -1$ **faire** $i \leftarrow \pi[i]$;
Retourner(i);
Finfonction

La complexité de la fonction *Racine* depend de la hauteur maximale des arborescences contenues dans la forêt représentée par le tableau π.

La procédure *Réunir* réunit deux composantes connexes en une seule si leurs racines r_1 et r_2 sont différentes.

Procédure *Réunir*(r_1, r_2 : Entier; **Var** π :Tableau$[1, \ldots, n]$ d'Entier);
{Raccroche l'arborescence de racine r_2 à celle de racine r_1, si ces deux arborescences sont distinctes, dans la forêt représentée par le tableau π}
Début
Si $r_1 <> r_2$ **alors** $\pi[r_2] \leftarrow r_1$
Finprocédure

La complexité de la procédure procédure *Réunir* est constante.

La procédure suivante, initialise le tableau π.

Procédure *Forêt_discrète*(G : *GrapheLArête*; **Var** π :Tableau$[1, \ldots, n]$ d' Entier);
Var i : Entier;
Début
{π représente la forêt discrète}
Pour i **de** 1 **à** $G.n$ **faire** $\pi[i] \leftarrow -1$;
Finprocédure

On rappelle que le type *GrapheLArête* est défini par :

$$
\begin{array}{llll}
Type & Arête = & & Enregistrement \\
 & & & s1 : Entier; \\
 & & & s2 : Entier; \\
 & & & suiv : \uparrow Arête \\
' & & & Fin; \\
 & GrapheLArête = & & Enregistrement \\
 & & & n : Entier; \\
 & & & LA : \uparrow LArête \\
 & & & Fin;
\end{array}
$$

Nous pouvons à présent écrire l'algorithme de recherche des composantes connexes.

Algorithme *Composantes_connexes*
{**Entrée** : Un graphe non orienté G de type *GrapheLArête*}
{**Sortie** : Le tableau π représentant les composantes connexes de G}
Var r_1, r_2 : Entier; $p : \uparrow Arête$; π : Tableau$[1, \ldots, n]$ d'Entier;
Début
Forêt_discrète(G, π);
$p \leftarrow G.LA$;
Tantque $p \neq Nil$ **faire**
Début
$r_1 \leftarrow Racine(p \uparrow .s1, \pi)$;
$r_2 \leftarrow Racine(p \uparrow .s2, \pi)$;
Réunir(r_1, r_2, π)
$p \leftarrow p \uparrow .suiv$
Fintanque
Finalgo

Exercice 10 *En utilisant l'algorithme Composantes_connexes, construire la forêt d'arborescences du graphe non orienté dont les arêtes sont connues progressivement dans l'ordre suivant :*
$\{3,4\}, \{6,8\}, \{1,2\}, \{5,6\}, \{9,10\}, \{7,8\}, \{4,5\}, \{11,10\}, \{4,6\}, \{2,8\}, \{12,10\},$.
Déterminez la hauteur maximale des arborescences.

8.3 Complexité de l'algorithme *Composantes_connexes*

Soit G d'un graphe à n sommets dont les arêtes (en nombre de $n-1$) sont données dans l'ordre suivant : $\{2,1\}, \{3,1\}, \{4,1\}, \ldots, \{n,1\}$. La forêt obtenue contient une seule arborescence de hauteur $n-1$. Comme la complexité, dans le pire des cas, de la fonction *Racine* est $\mathcal{O}(n)$, et comme celle de la procédure *Réunir* est $\mathcal{O}(1)$, l'algorithme *Composantes_connexes* a une complexité de $\mathcal{O}(mn)$.
Pour améliorer la complexité de cet algorithme, nous proposons deux heuristiques qui, combinées entre elles, conduisent à un algorithme très efficace.

8.3.1 L'heuristique de l'union pondérée

Pour améliorer la complexité de l'algorithme *Composantes_connexes*, on modifie la procédure *Réunir* de manière telle à, lors de l'adjonction de l'arête $\{x, y\}$, ne pas raccrocher systématiquement la racine de l'arborescence contenant y à celle de l'arborescence contenant x, mais à raccrocher la racine de l'arborescence qui contient le moins d'éléments à celle de l'arborescence qui en contient le plus.
Pour cela, nous modifions la définition du tableau π de la manière suivante :

- si r est une racine alors $\pi[r]$ contient une valeur strictement négative dont la valeur absolue est le nombre d'éléments de l'arborescence dont r est la racine
- si r n'est pas une racine, $\pi[r]$ contient le prédécesseur de r dans l'arborescence qui contient r

Nous remarquons qu'au début $\pi[r] = -1$ pour tout $r \in S$. En effet, dans la forêt discrète, chaque arborescence contient un seul élément, sa racine.

Procédure *Réunir_pondéré*(r_1, r_2 : Entier; **Var** π : Tableau$[1, \ldots, n]$ d'Entier);
{Raccroche la racine de l'arborescence contenant le moins d'éléments à celle qui en contient les plus, dans le cas où les deux arborescences sont distinctes et r_1 et r_2 en sont les racines}
Début
Si $r_1 \neq r_2$ **alors**
Si $\pi[r_1] > \pi[r_2]$ **alors**
{l'arborescence de racine r_1 a strictement moins d'éléments que l'arborescence de racine r_2}
$\pi[r_2] \leftarrow \pi[r_2] + \pi[r_1]$;
$\pi[r_1] \leftarrow r_2$
{On raccroche r_2 à r_1}
Sinon
$\pi[r_1] \leftarrow \pi[r_1] + \pi[r_2]$;
$\pi[r_2] \leftarrow r_1$
{On raccroche r_1 à r_2}
Finprocédure

On remarque que si $\pi[r_1] = \pi[r_2]$ alors on raccroche r_2 à r_1.

Nous obtenons ainsi un algorithme amélioré de recherche des composantes connexes.

Algorithme *Composantes_connexes_pondérés*
{**Entrée** : Un graphe non orienté G de type *GrapheLArête*}
{**Sortie** : Le tableau π représentant la partition en composantes connexes de G}
Var r_1, r_2 : Entier; $p : \uparrow$*Arête*; π : Tableau$[1, \ldots, n]$ d'Entier;
Début
Forêt_discrète(G, π);

$p \leftarrow G.LA$;
Tantque $p \neq Nil$ **faire**
Début
$r_1 \leftarrow Racine(p\uparrow.s1, \pi)$;
$r_2 \leftarrow Racine(p\uparrow.s2, \pi)$;
Réunir_pondéré(r_1, r_2, π);
$p \leftarrow p\uparrow.suiv$
Fintantque
Finalgo

Exercice 11 *Exécuter l'algorithme Composantes_connexes_pondérés sur les graphe de l'exercice 10.*

8.3.2 Complexité de l'algorithme Composantes_connexes_pondérés

La complexité de l'algorithme *Composantes_connexes_pondérés* suit de la proposition suivante.

Proposition 8.1 *Après une suite d'unions pondérées, toute arborescence à n sommets de la forêt d'arborescence T, représentée par le tableau des prédécesseurs π, a une hauteur inférieure ou égale à $log_2 n$*

Preuve. Par récurrence sur le nombre k des sommets contenus dans une arborescence de T.
Base : Si une arborescence A de T ne contient qu'un seul sommet alors $h(A) = 0 = log_2 1$.
Hypothèse de récurrence : Supposons la proposition vraie pour toute arborescence de T contenant un nombre de sommets strictement inférieure à k .
Soit A une arborescence de hauteur h contenant $k+1$ sommets. L'arborescence A résulte d'une union pondérée de deux arborescences A_1 et A_2 de hauteur respectivement h_1 et h_2 et ayant respectivement k_1 et k_2 sommets. Supposons $k_1 < k_2$, l'arborescence A a été alors obtenue par raccrochage de A_1 à A_2. La hauteur de l'arborescence A vérifie,

$$h = max(h_1 + 1, h_2).$$

Puisque, par hypothèse de récurrence, $h_1 < log_2 k_1$ et $h_2 < log_2 k_2$, nous obtenons :

$$h < max(log_2 k_1 + 1, log_2 k_2).$$

Des propriétés des logarithmes suit que $log_2 k_1 + 1 = log_2 2k_1 < log_2(k_1 + k_2)$ car $k_1 < k_2$. D'où $h < log_2(k_1 + k_2) = log_2 k$. Les cas $n_1 > n_2$ et $n_1 = n_2$ se prouvent de manière similaire.

■

Nous pouvons déduire de cette proposition que, si la forêt d'arborescences, qui représente la partition en composantes connexes d'un graphe à n sommets, est construite par unions pondérées successives, trouver à quelle composante connexe

appartient un sommet et tester si deux sommets sont dans la même composante requièrent, dans le pire des cas, une complexité de $\mathcal{O}(log_2 n)$. La complexité de l'algorithme *Composantes_connexes_pondérés* est donc de l'ordre de $\mathcal{O}(mlog_2 n)$.

8.3.3 L'heuristique de la compression des chemins

Pour rendre encore plus compacte la forme des arborescences obtenues, nous modifions la fonction *Racine* pour un sommet x, en raccrochant directement à la racine r de l'arborescence contenant x, tous les sommets situés sur le chemin c qui relie x à la racine r. Nous dirons, dans ce cas, que nous avons effectué une compression du chemin c. Puisque nous voulons à la fois trouver la racine r de l'arborescence contenant x et effectuer la compression du chemin reliant x à r en modifiant le tableau *père*, nous utiliserons une procédure plutôt qu'une fonction, c'est la procédure *Trouver_rapide* :

Procédure *Trouver_rapide*(x : Entier; **Var** r : Entier; **Var** π : Tableau$[1, \ldots, n]$ d'Entier);
{Dans r on trouve la racine de l'arborescence à laquelle x appartient et la compression du chemin reliant r à x est réalisée en modifiant le tableau des prédécesseurs}
Var i, j: entier;
Début
$i \leftarrow x$;
Tantque $\pi[i] > -1$ **faire** $i \leftarrow \pi[i]$;
$r \leftarrow i$; {r est la racine}
$i \leftarrow x$;
Tant que $\pi[i] > -1$ **faire**
Début
{compression du chemin}
$j \leftarrow i$;
$i \leftarrow \pi[i]$;
{on raccroche j à r}
$\pi[j] \leftarrow r$;
Fintantque
Finprocédure

Exercice 12 *Soit* $\pi = [-7, 1, 1, 3, 3, 5, 5, -1, 8, -1]$ *une forêt d'arborescence. Appliquez la procédure* $Trouver\_rapide(7, r, \pi)$.

Nous obtenons un algorithme amélioré de recherche des composantes connexes.

Algorithme *Composantes_connexes_rapide*
{**Entrée** : Un graphe non orienté G de type $GrapheLArête$}
{**Sortie** : Le tableau π représentant la partition en composantes connexes de G}
Var r_1, r_2 : Entier; $p : \uparrow Arête$: Tableau$[1, \ldots, n]$ d'Entier;
Début

```
p :← G.LA;
Forêt_discrète(G, π);
Tantque p ≠ Nil faire
  Début
    Trouver_rapide(p ↑ .s1, r1, π);
    Trouver_rapide(p ↑ .s2, r2, π);
    Réunir_pondéré(r1, r2, π);
    p ← p ↑ .suiv;
  Fintantque
Finalgo
```

Exercice 13 *Exécuter l'algorithme Composantes_connexes_rapide sur les graphe de l'exercice 10.*

8.3.4 Complexité de l'algorithme Composantes_connexes_rapide

On peut montrer que *Trouver_rapide* a une complexité, dans le pire des cas, de $\mathcal{O}(n + m\alpha(n + m, n))$, où $\alpha(n + m, n)$ est une fonction qui croît très lentement, bien plus lentement que la fonction logaritmique. Dans la pratique, $\alpha(u, v)$ est une constante inférieure ou égale à 4.

9 Arbres couvrants minimum : les algorithmes de Kruskal et Prim

Dans un circuit électrique, on a souvent besoin de rélier les broches des composantes électriquement équivalentes. Pour interconnecter n broches d'un circuit électrique, on utilise un arrangement de $n - 1$ cables, chacun reliant deux broches. Parmi tous les arrangements possibles, celui qui utilise une longueur de cable minimale est souvent le plus souhaitable. On peut modéliser un circuit électrique en utilisant un graphe $G = \langle S, A, p\rangle$ non orienté et valué dont les sommets sont les broches. Deux sommets de ce graphe seront réliés par une arête si les broches correspondantes sont interconnectés. L'évaluation $p(u)$ d'une arête $u \in A$ représente la longueur du cable. L'arrangement qui utilise une longueur minimale du cable sera un sous-graphe de G, $H = \langle S, A'\rangle$, qui soit connexe, sans cycle, dont d'évaluation $p(H) = \sum_{u\in A'} p(x, y)$ soit minimale. On appelle H un *arbre couvrant (de poid) minimum.* L'adjectif couvrant signifie que H contient tous les sommets de G et minimum que $p(H)$ est le minimum de l'ensemble $\{p(T) \mid T$ arbre couvrant pour G$\}$.
Cette section est dédié aux deux algorithmes qui permettent de construire un arbre couvrant minimum d'un graphe non orienté et valué.

Remarque 9.1 *Si $G = \langle S, A, p\rangle$ est un graphe connexe, on peut toujours trouver un arbre couvrant pour G, en supprimant de G les arêtes qui forment des cycles. Le graphe obtenu est alors encore connexe. C'est un arbre. L'existence suit du fait que l'on a seulement un nombre fini d'arbres couvrant pour G.*

9.1 L'algorithme de Kruskal

Soit $G = \langle S, A, p\rangle$ un graphe non orienté, connexe et valué. L'algorithme de Kruskal est un algorithme glouton qui utilise la même méthode de l'algorithme de recherche de composantes connexes. Il s'agit de construire l'unique composante connexe de G en faisant attention à ne pas avoir de cycle. Pour cela, on ordonne d'abord l'ensemble des arêtes $A = \{u_1, \ldots, u_n\}$ en ordre croissant de leur poids. On construit ensuite, l'unique composante connexe de G, à partir du graphe discret $G_0 = \langle S, \emptyset\rangle$ et en procédant par adjonctions successives des arêtes de A. Si l'arête u_i rélie deux composantes connexes différentes elle est ajoutée à l'arbre couvrant minimum sinon elle est rejetée car son ajout provoquerait la formation d'un cycle dans l'arborescence relative à cette composante.

Algorithme *Kruskal*;
{**Entrée** : Un graphe G non orienté et connexe donné par la liste des ses arêtes}
{**Sortie** : L'ensemble ACM d'arêtes de G qui forment un arbre couvrant minimum pour G}
Var ACM, AO : Ensemble d'arête;
Début
$ACM \leftarrow \emptyset$; $AO \leftarrow \emptyset$;
Tri(A, AO);
{Tri est une procédure qui permet de trier l'ensemble A et range les valeurs triées de A dans AO}
Pour toute arête $\{x, y\} \in AO$ **faire**
    Début
    Si x et y n'appartiennent pas à la même composante connexe **alors**
        Début
        Ajouter$(\{x, y\}, ACM)$;
        Réunir les deux composantes connexes en une seule;
        Finsi
    Fintanque
Finalgo

Exemple 9.2 *Exécuter l'algorithme de Kruskal sur le graphe dont l'ensemble des arêtes est le suivant :*
$A = \{(\{1, 2\}, 8), (\{1, 3\}, 2), (\{1, 4\}, 1), (\{2, 4\}, 7), (\{2, 5\}, 5), (\{3, 4\}, 3), (\{3, 5\}, 7), (\{4, 5\}, 10)\}$.

9.2 Implantation de l'algorithme de Kruskal

Pou implanter l'algorithme de Kruskal, nous utilisons, comme structure des données, le tableau π pour gérer la partition en composantes connexes de la suite de graphes, $G_0, \ldots, G_i, \ldots, G_m$ où $G_0 = \langle S, \emptyset\rangle$ et $G_m = G$, $G_i = G_{i-1} + u_i$ pour tout $1 \leq i \leq m$ et $AO = \{u_1, \ldots, u_m\}$. Nous utilisons les procédures *Réunir_pondéré* et *Trouver_rapide* pour obtenir une implantation efficace de l'algorithme de Kruskal.

Algorithme*Kruskal_rapide*;
{**Entrée** : Un graphe non orienté et connexe G de type $GrapheLArête$}
{**Sortie** : L'ensemble ACM d'arêtes de G qui forment un arbre couvrant minimum pour G}
Var ACM: Ensemble d'Arête; AO : $\uparrow Arête$; π : Tableau$[1 \cdots n]$ d'Entier; r_1, r_2 : Entier; $p \leftarrow \uparrow Arête$;
Début
$ACM \leftarrow \emptyset$; $AO \leftarrow []$;
Pour i **de** 1 **à** n **faire** $\pi[i] := -1$;
{π représente la forêt discrète}
Tri$(G.LA, AO)$;
{La procédure Tri range les valeurs triées de la liste $G.LA$ dans la liste AO}
$p \leftarrow AO$;
Tantque $p \neq Nil$ **faire**
Début
Trouver_rapide $(p \uparrow .s1, r_1, \pi)$;
Trouver_rapide $(p \uparrow .s2, r_2, \pi)$;
Si $r_1 \neq r_2$ **alors**
Début
Ajouter$(\{p \uparrow .s1, p \uparrow .s2\}, ACM)$;
Réunir_pondéré$(r_1,\ r_2,\ \pi)$
Finsi
$p \leftarrow p \uparrow .suiv$
Fintantque
Finalgo

Exemple 9.3 *Exécuter l'algorithme de Kruskal_rapide sur le graphe de l'exemple 9.2.*

9.2.1 Complexité de l'algorithme de *Kruskal_rapide*

La complexité de cet algorithme depend de la complexité de la fonction Tri, elle est donnée par $Max\{\mathcal{O}(Tri),\ \mathcal{O}(n + m\alpha(n + m, n))\}$ où $\alpha(n + m, n)$ est une fonction qui croît très lentement, que l'on peut considéder comme une constante.

9.3 L'algorithme de Prim

Soit $G = \langle S, A\rangle$ un graphe non orienté. Pour tout $T \subset S$, nous notons :

$$\Omega(T) = \{\{x, y\} \mid x \in T, y \notin T\}, \text{ le } \textit{cocycle} \text{ de } T,$$

L'algorithme de Prim utilise la propriété fondamentale suivante des arbres couvrants minimum :

Proposition 9.4 *Soit $G = \langle S, A, p\rangle$ un graphe non orienté, connexe et valué et soit $T \subset S$. Si $\{t, s\}$ est une arête du graphe de poids minimum parmi les arêtes du cocycle $\Omega(T)$, alors il existe un arbre couvrant minimum contenant l'arête $\{t, s\}$.*

Preuve Supposons par l'absurde qu'il n'existe pas d'arbre couvrant minimum de G contenant l'arête $\{t, s\}$. Soit R un arbre couvrant minimum pour G. L'ajout de $\{t, s\}$ crée donc un cycle c dans R. Le cycle c est de forme $c = (t, s, \ldots, s', t', \ldots t)$, où $t, t' \in T$ et $s, s' \in S \setminus T$. Par hypothèse sur le choix de $\{t, s\}$, nous avons que $p(t', s') \geq p(t, s)$. Si $p(t', s') > p(t, s)$, en enlevant de R l'arête $\{t', s'\}$ et en la remplaçant par l'arête $\{t, s\}$, nous obtenons un arbre couvrant dont l'évaluation est strictement inférieure à celle de R. Cela contredit l'hypothèse de minimalité de R. Si $p(t', s') = p(t, s)$ alors il existe un arbre couvrant minimum de G contenant l'arête $\{t, s\}$ et cela contredit l'hypothèse qu'il n'existe pas d'arbre couvrant minimum de G contenant l'arête $\{t, s\}$.

■

Algorithme*Prim*;
{**Entrée** : Un graphe non orienté et connexe G donné par l'ensemble des ses sommets S et la liste des arêtes A}
{**Sortie** : L' ensemble ACM d'arêtes de G qui forment un arbre couvrant minimum pour G}
Var T : Ensemble d'Entier; ACM : Ensemble d'Arête;
Début
$ACM \leftarrow \emptyset$; $T \leftarrow \emptyset$;
Choisir(r, S);
Ajouter(r, T);
{On fait pousser un ACM à partir du sommet r}
Tant que $T \neq S$ **faire**
  Début
  Calculer $\Omega(T)$;
  Choisir une arête $\{x, y\} \notin ACM$ d'évaluation minimale dans $\Omega(T)$;
  {On suppose $x \in T$ et $y \notin T$}
  Ajouter(y, T);
  Ajouter$(\{x, y\}, ACM)$;
  Fintanque
Retourner(ACM);
Finalgo

Exemple 9.5 *Exécuter l'algorithme de Prim sur le graphe de l'exemple 9.2 à partir du sommet* 1.

L'opération fondamentale de l'algorithme de Prim est la recherche d'une arête de poids minimal d'un cocycle. Cette opération est appliquée à une suite de cocycles d'ensembles de sommets dans laquelle chaque ensemble contient un sommet de plus que le précédent. Pour implanter de manière efficace l'algorithme de Prim,

l'important est de faciliter la selection de la nouvelle arête à ajouter à l'arbre couvrant de poids minimal. Une implantation efficace de ces opérations utilise une file de priorité F pour gérer les sommets du graphe qui ne sont pas dans l'arbre couvrant de poids minimum. À F est associé un champ *clé* défini par clé[v] est le poids d'une arête de poids minimum qui rélie le sommet v à l'arbre couvrant de poids minimum et clé[v]$= \infty$ si cette arête n'existe pas. Avec cette implantation l'algorithme de Prim a une complexité de l'ordre de $\mathcal{O}(nlog_2 n)$.

10 Parcours de graphes orientés

Pour la résolution de certains problèmes sur les graphes, il est souvent nécessaire d'explorer le graphe, c'est à dire de procéder à un examen exhaustif des sommets et des arcs. Tous les algorithmes d'exploration de graphes suivent le même schéma

10.1 L'algorithme d'exploration de graphes

L'exploration d'un graphe construit un ensemble de sommets appelé *frontière*. Au début la frontière contient le sommet à partir duquel on commence l'exploration du graphe. Ensuite, on rétire de la frontière un sommet et on le développe en générant tous ses successeurs. On obtient ainsi un ensemble de sommets que l'on ajoute à la frontière. Le dévéloppement des sommets de la frontière continue jusqu'à ce que il ne reste plus de sommets à dévélopper. La présence de circuits dans le graphe conduit à générer un sommet un nombre infini de fois, faisant échouer l'algorithme. Pour éviter cela, on ajoute à l'algorithme un ensemble dans lequel on garde tous les sommets qui ont été générés. A chaque fois que l'on génére un successeur du sommet de la frontière que l'on est en train de dévélopper, on vérifie qu'il n'a pas encore été généré avant de l'ajouter à la frontière. L'ensemble des sommets générés possède une structure arborescente appelée *arborescence d'exploration*, qui constitue la sortie de l'algorithme. Elle est implantée par le tableau des prédécesseurs π.

Algorithme *ExplorationDeGraphe*;
{**Entrée** : Un graphe G à n sommets et un sommet i}
{**Sortie** : L'arborescence d'exploration π constituée des tous les sommets accessibles à partir de i}
Var $Frontière, Généré$: Ensemble d'Entier; s, t : Entier;
Début
$Frontière \leftarrow \emptyset$; Ajouter($i, Frontière$);
$Généré \leftarrow \emptyset$; Ajouter($i, Généré$);
Pour tout k de 1 à n **faire** $\pi[i] \leftarrow -1$;
Tantque $Frontière \neq \emptyset$ **faire**
Début
Choisir un sommet $s \in Frontière$;
{Stratégie d'exploration}
Rétirer($s, Frontière$);

Pour tout successeur t de s **faire**
Début
Si $t \notin$ *Généré* **alors**
Début
Ajouter$(t, Frontière)$;
Ajouter$(t, Généré)$;
$\pi[t] \leftarrow s$
Finsi
Fintantque;
Retourner(π)
Finalgo.

Cet algorithme explore tous les sommets accessibles à partir du sommet i. Dans le cas d'un graphe non connexe, il faudra appliquer cet algorithme sur chaque composante connexe. Nous obtiendront donc une forêt d'exploration.

10.2 Les différentes stratégies d'exploration

Les algorithmes d'exploration de graphes différent dans leur stratégie d'exploration, c'est à dire dans la manière qu'ils ont de choisir le prochain sommet à dévélopper. Cela se traduit par la manière avec laquelle l'ensemble *Frontière* est implanté : liste FIFO, pile ou file de priorité.

Pour illustrer ces différentes stratégies d'exploration, nous utilisons le graphe dont l'ensemble des sommets est $S = \{1, 2, 3, 4, 5, 6\}$ et dont les listes des successeurs sont $\mathbf{1} : 2, 3;\ \mathbf{2} : 3, 4, 5;\ \mathbf{3} : 6; \mathbf{4} : 5;\ \mathbf{5} : Nil; \mathbf{6} : 5$. L'exploration commence par le sommet 1.

- **Parcours en profondeur** : La frontière est gérée avec une pile. On choisit de dévélopper d'abord le sommet de la frontière dont la distance en nombre d'arcs (dite profondeur), du sommet i est la plus élévé. Lorsque un sommet s est rétiré de la frontière, ses successeurs, dont la profondeur strictement supérieure à celle de s, sont insérés au début de la pile. Ainsi, on dévéloppe, en premier, le sommet le plus profond.

  Exemple 10.1 *Montrer l'évolution de la frontière et de l'arborescence d'exploration du parcours en profondeur du graphe exemple, à partir du sommet* 1.

- **Parcours en profondeur avec retour en arrière** : C'est une variante du parcours en profondeur dans laquelle on ne génère que un successeur à la fois au lieu de le générer tous. La condition d'arrêt est remplacée par $(F \neq \emptyset)$ et (Généré $\neq S$). Ce type de parcours utilise moins de mémoire que le parcours en profondeur. Cette stratégie d'exploration est connue aussi sous le nom d'exploration par rebroussement ou back-tracking.

Exemple 10.2 *Montrer l'évolution de la frontière et de l'arborescence d'exploration du parcours en profondeur avec retour en arrière du graphe exemple, 'a partir du sommet* 1*.*

- **Parcours en largeur** : La frontière est gérée avec une file FIFO. On choisit de dévélopper d'abord le sommet le moins profond. Lorsque un sommet s est rétiré de la frontière, ses successeurs, dont la profondeur est strictement supérieure à celle de s, sont insérés à la fin de la file. Ainsi, on dévéloppe, en premier, le sommet le moins profond.

  Exemple 10.3 *Montrer l'évolution de la frontière et de l'arborescence d'exploration du parcours en largeur du graphe exemple, à partir du sommet* 1*.*

- **L'algorithme de Dijkstra** : La frontière est une file de priorité. On choisit de dévélopper d'abord le sommet dont la valeur de l'attribut d est minimale.

  Exemple 10.4 *Montrer l'évolution de la frontière et de l'arborescence d'exploration de l'algorithme de Dijkstra sur le graphe orienté et valué G à six sommets dont les listes des successeurs sont :*
  $\mathbf{1} : (\mathbf{2}, 2), (\mathbf{3}, 5);\ \mathbf{2} : (\mathbf{3}, 1), (\mathbf{4}, 4), (\mathbf{5}, 1);\ \mathbf{3} : (\mathbf{6}, 1); \mathbf{4} : (\mathbf{5}, 2);\ \mathbf{5} : Nil; \mathbf{6} : (\mathbf{5}, 1).$
  La notation $\mathbf{x} : (\mathbf{y}, p)$ *signifie l'arc* (x, y) *de poids* p*.*

10.3 Implantation du parcours en profondeur dans la version par rebroussements (backtracking)

Le parcours en profondeur d'un graphe orienté $G = \langle S, A \rangle$ est un procédé récursif de marquage des sommets de G, dans le but d'obtenir deux ordres totaux sur l'ensemble des sommets, l'ordre préfixe P et celui suffixe S, et la forêt d'exploration π, dite aussi forêt en profondeur. Ces trois objets donnent de nombreuses informations sur la structure du graphe, comme nous le verrons dans la suite. Nous allons implanter la version par rebroussements du parcours en profondeur. On dit qu'un sommet est exploré si tous ses successeurs ont été explorés. Lors du parcours en profondeur, les sommets sont explorés en empruntant les arcs qui sortent du sommet v, découvert le plus récemment et dont on a pas encore exploré tous les successeurs. Lorsque tous les successeurs de v on été explorés, on revient en arrière pour explorer les successeurs non encore explorés du sommet u à partir duquel v a été découvert.
Nous associons à chaque sommet une couleur parmi blanc, gris et noir :

- *blanc*, qui signifie *non encore découvert*,
- *gris*, qui signifie *en cours d'exploration*,
- *noir*, qui signifie *exploré*.

Les sommets sont explorés en suivant l'ordre croissant. Au début, tous les sommets de G ont la couleur blanche. Ensuite, un sommet passe de blanc à gris quand on le

visite pour la première fois et de gris à noir quand on a terminé d'explorer tous ses successeurs. L'ordre préfixe des sommets est celui du passage du blanc au gris et celui suffixe du gris au noir. Un arc $(v, u) \in A$ fait partie de la forêt d'exploration, on l'appelle *arc de liaison*, si le sommet u est découvert pour la première fois pendant le balayage de la liste des successeurs de v. Pour écrire l'algorithme du parcours en profondeur, nous utiliserons la structure des données suivante :

- Le graphe G est de type GrapheTL.
- Le type *Etat = (blanc, gris, noir).*
- *Couleur* est un tableau, dont les éléments appartiennent au type *Etat*, défini par *couleur*$[u]$ est l'état du sommet u.
- P est un tableau d'entiers défini par $P[u]$ donne la date de la découverte du sommet u.
- P^* est un tableau de sommets défini par $P^*[i]$ contient le somment que l'on découvre à l'instant i.
- S est un tableau d'entiers défini par $S[u]$ donne la date de la fin d'exploration du sommet u.
- S^* est un tableau de sommets défini par $S[i]$ contient le sommet que l'on termine d'explorer à l'instant i.
- π est un tableau, dont les indices sont les sommets de G, défini par $\pi[u] = -1$, si u est une racine de la forêt d'exploration et par $\pi[u]$ est le prédécesseur du sommet u sinon.

Algorithme *Parcours en Profondeur*;
{**Entrée** : Le graphe G de type GrapheTL}
{**Sortie** : Les tableaux P, S, P^*, S^*, π}
Var u, i_p, i_s : Entier;
Début
Pour u **de** 1 **à** $.Gn$ **faire**
    Début
    couleur$[u] \leftarrow$ *blanc*;
    $\pi[u] \leftarrow -1$;
    $P[u] \leftarrow 0$; $S[u] \leftarrow 0$;
    $P^*[u] \leftarrow 0$; $S^*[u] \leftarrow 0$;
    Finpour
$i_p \leftarrow 0$; $i_s \leftarrow 0$,
Pour u **de** 1 **à** $.Gn$ **faire**
    Début
    Si (*couleur*$[u]$ = *blanc*) **alors** *Visiter_en_profondeur*$(u, \ldots)$;
    Finpour

Retourner (P, S, P^*, S^*, π)
Finalgo

La procédure récursive *Visiter_en_prof* est définie par le code :

Procédure *Visiter_en_profondeur*(u : *Entier*; ...);
Var p : ↑Cellule, ...;
Début
$couleur[u] \leftarrow gris$;
$i_p \leftarrow i_p + 1$;
$P[u] \leftarrow i_p$;
$P^*[i_p] \leftarrow u$;
$p \leftarrow G.L[u]$;
Tantque $p \neq$ Nil **faire**
  Début
  $v \leftarrow p\uparrow .sommet$;
  Si ($couleur[v] = blanc$) **alors**
    Début
    $\pi[v] \leftarrow u$;
    Visiter_en_profondeur($v, \ldots$);
    Finsi;
  $p \leftarrow p\uparrow .suiv$;
  Fintantque;
  $couleur[u] \leftarrow noir$;
  $i_s \leftarrow i_s + 1$;
  $S[u] \leftarrow i_s$;
  $S^*[i_s] \leftarrow u$;
Retourner(...)
Finprocédure

Nous laissons au lecteur le soin de préciser les paramètres de la procédure *Visiter_en_profondeur*.

Exemple 10.5 *On considère le graphe* $G = \langle S, A \rangle$ *donné par ses listes de successeurs :*
1 : 3, 4
2 : 1
3 : 2, 4
4 : 2
5 : 7
6 : 3, 5
7 : 4, 6
8 : 4, 7

Exécuter le parcours en profondeur de G pour obtenir les ordres P et S et la forêt d'exploration π.

10.3.1 Complexité du parcours en profondeur

La complexité de l'algorithme du parcours en profondeur depend du nombre d'appels et de la complexité de la procédure *Visiter_en_prof*. La procédure *Visiter_en_profondeur* est exécutée exactement une fois pour chaque sommet du graphe, puisque elle est invoquée seulement pour les sommets blancs et qu'elle commence pour le peindre en gris. Pendant l'exécution de *Visiter_en_profondeur* sur un sommet u, la boucle, qu'elle contient, est exécutée nombre de successeurs de u, $|Succ(u)|$, fois. Comme $\sum_{u \in S} |Succ(u)| \mid = \mathcal{O}(\mid A \mid)$, le temps d'exécution de la totalité des appels de *Visiter_en_profondeur*, et donc du parcours en profondeurs, est $\mathcal{O}(\mid S \mid + \mid A \mid) =$ Max$\{\mathcal{O}(\mid S \mid, \mathcal{O}(\mid A \mid)\}$.

10.4 Propriétés du parcours en profondeur

Dans cette section, on considère une variante de l'algorithme du parcours en profondeur. On remplace les deux compteurs i_p, i_s par un seul compteur i. Cela permet de faire en sorte que la date de découverte de chaque sommet soit toujours inférieure à la date de fin d'exploration.

Nous enonçons, sans les prouver, deux propriétés fondamentales du parcours en profondeur. Nous notons π la forêt d'arborescences du parcours en profondeur de G.

Théorème 10.6 (Théorème des parenthèses) *Dans un parcours en profondeur d'un graphe orienté G, les propriétés suivantes sont vérifiées :*

1. *Si u et v appartiennent à deux arborescences différentes de π alors les intervalles $[P[u], S[u]]$ et $[P[v], S[v]]$ sont disjoints.*

2. *Si u et v appartiennent à la même arborescence de π et si u est un descendant de v alors l'intervalle $[P[u], S[u]]$ est inclus dans l'intervalle $[P[v]S[v]]$.*

Exemple 10.7 *Dans le parcours en profondeur du graphe de l'exemple 10.5, calculer les intervalles pour les sommets :*

1. $u = 4, v = 1$,

2. $u = 3, v = 8$,,

3. $u = 2, v = 4$.

Remarquer que si u et v appartiennent à la même arborescence et que si u n'est ni descendent ni ascendant de v alors on ne peut rien dire sur les intervalles $[P[u], S[u]]$ et $[P[v], S[v]]$.

Nous utiliserons le résultat contenu dans le théorème suivant, pour tester la présence de circuits dans un graphe.

Théorème 10.8 (Théorème du chemin blanc) *Dans la forêt en profondeur π d'un graphe orienté G, un sommet u est un descendant d'un sommet v si et seulement si à la date $P[v]$, où le parcours découvre pour la première fois le sommet v, il existe dans G un chemin de v vers u dont tous les sommets, sauf v, sont blancs.*

10.4.1 Classification des arcs

Dans la forêt en profondeur π d'un graphe G, il est possible de définir quatre types d'arcs :

- *Arcs de liason*, sont les arcs de G qui font partie de π.
- *Arcs de retour*, sont les arcs (u, v) de G reliant un sommet u à un ascendant v dans π.
- *Arcs en avant*, sont les arcs (u, v) de G qui ne sont pas d'arc de liason et qui rélient un sommet u à un descendant v dans π.
- *Arcs traversiers*, sont tous les autres arcs de G.

Exemple 10.9 *Ajouter à la forêt d'exploration du parcours en profondeur du graphe G de l'exemple 10.5, les arcs de G qui n'en font pas partie et identifier chaque arc.*

10.5 Les applications du parcours en profondeur

Le parcours en profondeur possède de nombreuses applications. Il est utilisé dans la recherche de circuits, des composantes fortement connexes et des tris topologiques d'une graphe orienté.

10.5.1 Présence de circuits

Les arcs de retour permettent de tester la présence de circuits dans un graphe orienté, comme le montre la proposition suivante :

Proposition 10.10 *Un graphe orienté G possède un circuit si et seulement si le parcours en profondeur de G génère un arc de retour.*

Preuve.
$\leftarrow$: Supposons qu'un arc de retour (u, v) soit généré par le parcours en profondeur de G. Alors v est un ascendant de u dans l'arborescence de la forêt d'exploration à laquelle ils appartiennent. Comme les arcs de liaison sont des arcs de G, il existe un chemin de v vers u dans G. L'arc de retour (u, v), qui appartient aussi à G, clôt ce chemin et le graphe possède un circuit.
$\rightarrow$: Supposons que G contienne un circuit c et montrons que le parcours en profondeur de G génére un arc de retour. Soit v le premier sommet de c découvert par

le parcours en profondeur et soit u le prédécesseur de v dans c. A la date $P[v]$ il existe dans G un chemin composé de sommets blancs de v vers u, Par le théorème du chemin blanc, u est un descendant de v dans π. L'arc (u, v) est donc un arc de retour.

10.5.2 Tri topologique

Un *tri* ou une *liste topologique* d'un graphe orienté G est une liste qui contient tous les sommets de G et dans laquelle chaque sommet vient avant tous ses successeurs.

Exemple 10.11 *Trouver au moins deux tris topologiques pour le graphe suivant, donné par ses listes de successeurs :*
$1 : 4, 5$
$2 : 3$
$3 : Nil$
$4 : 2, 5$
$5 : 3$

Proposition 10.12 *Un graphe orienté $G = \langle S, A\rangle$ n'a pas de circuits si et seulement s'il existe une liste topologique des ses sommets.*

Preuve. S'il existe une liste topologique pour le graphe G aucun sommet $x \in S$ ne peut être au même temps ascendant et descendant d'un même sommet $y \neq x$. Cela implique que G est sans circuits.
Nous prouvons la condition nécessaire par récurrence sur le nombre n de sommets du graphe. Si $n = 1$ alors la propriété est vraie. Supposons qu'elle soit vraie pour tout graphe ayant moins de $n \geq 2$, sommets. Considérons un graphe G à $n+1$ sommets. Puisque G ne possède pas de circuits, il existe au moins un sommet s n'ayant pas de successeur. Le graphe $G - s$ obtenu de G en supprimant le sommet s et tous les arcs dont s est l'extrémité finale, est un graphe sans circuits à n sommets. Il existe, par hypothèse de récurrence, une liste topologique L_s de G_s. La liste L obtenu par concatenation de (s) et de L_s est alors une liste topologique pour G.

La proposition précédente sugère un algorithme permettant de tester si un graphe orienté est sans circuit. Le principe est de rechercher dans G un sommet s sans successeurs. Si ce sommet n'existe pas alors G possède un circuit sinon la réponse pour G est la même que pour $G - s$. Nous obtenons ainsi un algorithme dont la complexité est de l'ordre $\mathcal{O}(n^3)$ où n est le nombre de sommets de G, si le graphe est donné par sa matrice d'adjacence et de l'ordre de $\mathcal{O}(nm)$ où m est le nombre d'arcs, si G est donné par sa liste de successeurs. Nous avons vu que le parcours en profondeurs permet de tester la présence de circuits dans un graphe avec une complexié de l'ordre de $\mathcal{O}(n + m)$. Nous allons voir qu'il permet aussi de repérer une liste topologique d'un graphe orienté et sans circuits avec la même complexité. Soit S^* le tableau qui donne l'ordre suffixe associé au parcours en profondeur de

G. Nous notons S^{*-1} le tableau obtenu à partir de S^* en inversant l'ordre de ses éléments.

Proposition 10.13 *L'ordre suffixe inverse donne un tri topologique pour un graphe sans circuits G.*

Preuve. Il suffit de montrer que pour tout arc (u, v) de G, $S^*[v] < S^*[u]$. Lorsque l'arc (u, v) est emprunté par le parcours en profondeur, le sommet v ne peut pas être gris. En fait, cela signifierait que v est un ascendant de u dans π (v est découvert avant v) et (u, v) serait un arc de retour, ce qui contredit l'hyphotèse faite sur G. Le sommet v peut donc être blanc ou noir. Dans les deux cas, nous pouvons conclûre que $S^*[v] < S^*[u]$. En effet, si v est blanc, v devient un descendant de u dans π et si v est noir, l'inégalité est évidente.

10.5.3 Composantes fortement connexes

On définit sur l'ensemble des sommets d'un graphe orienté $G = \langle S, A\rangle$, une relation d'équivalence F par :
$x\ F\ y$ si et seulement si $x = y$ ou s'il existe un chemin de x vers y et un chemin de y vers x.
La relation F détermine une partition $F_1, \ldots, F_k$ de l'ensemble de sommets de S. Les sous-ensembles F_i de S s'appellent les *composantes fortement connexes* de G. Un graphe est *fortement connexe* s'il possède une seule composante fortement connexe.

Exemple 10.14 *Déterminer les composantes fortement connexes du graphe :*
$1 : 2$
$2 : 3, 5, 6$
$3 : 4, 7$
$4 : 3, 8$
$5 : 1, 6$
$6 : 7$
$7 : 6$
$8 : Nil$

Pour déterminer les composantes fortement connexes d'un graphe orienté G, nous introduisons la notion de *graphe inverse* de G, noté G^{-1}. Il est défini par $G^{-1} = \langle S, A^{-1}\rangle$ où $A^{-1} = \{(u, v) \mid (v, u) \in A\}$.

Remarque 10.15 *Si le graphe G est donné par ses listes de successeurs, la construction de G^{-1} demande une complexité de l'ordre $\mathcal{O}(\mid S \mid + \mid A \mid)$*

Exemple 10.16 *Construire le graphe inverse du graphe de l'exemple 10.14.*

10.6 Calcul des composantes fortement connexes selon la méthode de Kosaraju

Les composantes fortement connexes de G se déteminent en effectuent deux parcours en profondeur, l'un sur G et l'autre sur G^{-1}, selon la méthode de Kosaraju, dont on ne donnera pas la preuve.

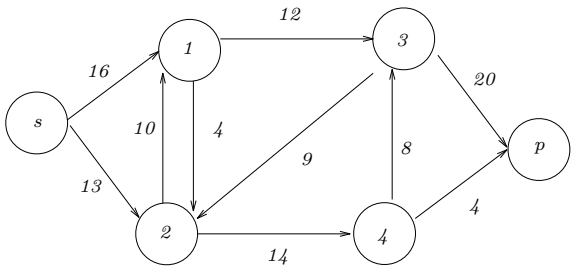
1. Exécuter le parcours en profondeur de G pour obtenir l'ordre suffixe S^*.
2. Construire G^{-1}.
3. Exécuter le parcours en profondeur de G^{-1} en utilisant à la place de l'ordre croissant des sommets, l'ordre suffixe inverse.
4. Les arborescences de la forêt d'exploration du parcours en profondeur de G^{-1} sont les composantes fortement connexes de G.

Exemple 10.17 *Exécuter la méthode de Kosaraju sur le graphe de l'exemple 10.14*

11 Flux et réseaux de transport

On utilise un graphe orienté particulier, que l'on appelle *réseau de transport*, pour modéliser le phénomème suivant : un matériau est produit par une source s avec un débit d, il se déplace à l'intérieur d'un système de tuyaux et il arrive à un puit p où il est consommé avec le même débit d. Les différents tuyaux sont réliés entre eux par des points de jonction où le matériau s'écoule sans gain ni perte. En outre, chaque tuyau a une capacité fixe, qui représente le débit maximal qui peut atteindre le matériau à travers le tuyau. Le *flot* du matériau représente la vitesse avec laquelle le matériau se déplace.
Un *réseau de transport* est un graphe orienté $G = \langle S, A\rangle$ muni d'une fonction de capacité $c : S \times S \longrightarrow \mathbb{R}$ telle que $c(x, y) \geq 0$ et si $c(x, y) = 0$ alors $(x, y) \notin A$. Dans G deux sommets ont un status particulier : la *source* s et le *puit* p. On suppose que tout sommet se trouve sur un chemin qui rélie s à p. Le graphe G est donc connexe et $|A| \geq |S| - 1$. On note $R = (\langle S, A\rangle, c, s, p)$ un réseau de transport.

Exemple 11.1 *Réseau de transport*



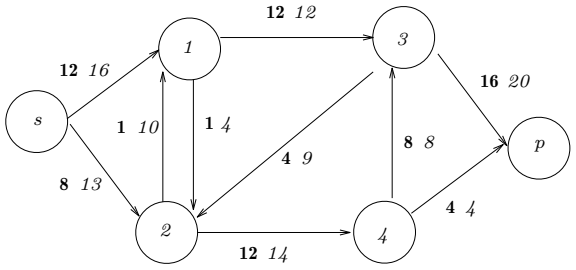
Un *flot* dans un réseau de transport $R = (\langle S, A\rangle, c, s, p)$ est une fonction $f : S\times S \longrightarrow \mathbb{R}$ qui satisfait les propriétés suivantes :

1. **Contrainte de capacité** : $f(u, v) \leq c(u, v),\ \forall u, v \in S$.
2. **Symétrie** : $f(u, v) = -f(v, u),\ \forall u, v \in S$.
3. **Conservation du flot** : $\sum_{(u,v)\in A} f(u, v) + \sum_{(u,v)\in A} f(v, u) = 0,\ \forall u \in S, u \neq s, p$.

La *valeur* d'un flot est définie par $|f| = \sum_{v \in S} f(s, v)$.

Remarque 11.2 *Par définition, si* $(u, v) \notin A$ *et* $(v, u) \notin A$ *alors* $c(u, v) = c(v, u) = 0$. *Par la propriété de contrainte de capacité* $f(u, v) \leq 0$ *et* $f(v, u) \leq 0$. *Puisque , par la propriété de symétrie,* $f(u, v) = -f(v, u)$, *on obtient* $f(u, v) = f(v, u) = 0$. *On en conclût que un flot non nul entre* u *et* v *implique* $(u, v) \in A$ *ou* $(v, u) \in A$. *En outre, par la propriété de symétrie, le flot entre un sommet et lui même vaut zéro. En effet,* $f(u, u) = -f(u, u)$.

Exemple 11.3 *Flot dans un réseau de transport*
Un flot f *dans le réseau de transport de l'Exemple 11.1. Les valeurs du flot sont en gras. La valeur du flot* f *est* $|f| = 20$.



Problème du flot maximal Etant donné un réseau de transport $R = (\langle S, A \rangle, c, s, p)$, trouver un flot dans R de valeur maximale.

11.1 La méthode de Ford-Fulkerson

La méthode de Ford-Fulkerson calcule un flot de valeur maximale dans un réseau de transport $R = (\langle S, A \rangle, c, s, p)$. C'est une méthode itérative. Au début, on définit dans R le flot nul, $f(u, v) = 0$, $\forall u, v \in S$. Le flot nul a une valeur nulle. À chaque itération, on augmente la valeur du flot en trouvant une chaîne améliorante, que l'on peut voir comme une chaîne de s vers p le long de laquelle on peut augmenter la valeur du flot. Nous montrerons que ce processus finit par engendrer un flot de valeur maximal dans R.

Méthode de Ford-Fulkerson(R)
1. Initialiser f à zéro.
2. **Tantque** il existe une chaîne améliorante e **faire**
augmenter la valeur de f sur e.
3. Retourner f.

11.1.1 Réseaux résiduels

Un *réseau résiduel* d'un réseau de transport $R = (\langle S, A \rangle, c, s, p)$ est un sous-graphe de R qui contient tout les sommets de R et les arcs de R qui peuvent supporter un flot plus important. Un réseau résiduel est donc un réseau de transport.

Soit f un flot dans R est soient $u, v \in S$. La quantité de flot supplémentaire qu'il est possible d'ajouter entre u et v sans dépasser la capacité $c(u, v)$ est appelée *capacité résiduelle* de (u, v). Elle est donnée par

$$c_f(u, v) = c(u, v) - f(u, v).$$

Exemple 11.4 *Considérons un arc (u, v) tel que $c(u, v) = 16$ et $f(u, v) = 11$ Alors $c_f(u, v) = 16 - 11 = 5$.*

Si le flot sur un arc (u, v) est strictement négatif, $f(u, v) < 0$, alors $c_f(u, v) > f(u, v)$. Soit f un flot dans R, le *réseau résiduel* R_f induit par f est $R_f = (\langle S, A_f\rangle, c_f, s, p)$ où $A_f = \{(u, v) \in S \times S \mid c_f(u, v) > 0\}$. La proposition suivante montre la relation entre un flot d'un réseau résiduel et le flot du réseau de transport initial.

Proposition 11.5 *Soit $R = (\langle S, A\rangle, c, s, p)$ un réseau de transport et soit f un flot dans R. Soit $R_f = (\langle S, A_f\rangle, c_f, s, p)$ le réseau résiduel induit par f et soit f' un flot dans R_f. Alors $(f + f')$ est un flot dans R dont la valeur est $|f + f'| = |f| + |f'|$.*

Preuve. La fonction $(f + f')$ est définie par $(f + f')(u, v) = f(u, v) + f'(u, v)$. Il faut montrer que $f + f'$ vérifie les propriétés de contrainte de capacité, de symétrie et de conservation du flot.

1. On remarque que, comme f' est un flot dans R_f, il vérifie la propriété de contrainte de capacité, $f'(u, v) \leq c_f(u, v)$, et que, par définition, $c_f(u, v) = c(u, v) - f(u, v)$. On obtient alors :

$$\begin{array}{rcl} (f + f')(u, v) & = & f(u, v) + f'(u, v) \\ & \leq & f(u, v) + c_f(u, v) \\ & = & f(u, v) + c(u, v) - f(u, v) \\ & = & c(u, v). \end{array}$$

2. Comme f et f' sont des flux, ils vérifient la propriété de symétrie, $f(u, v) = -f(v, u)$ et $f'(u, v) = -f'(v, u)$. On obtient alors les égalités suivantes :

$$\begin{array}{rcl} (f + f')(u, v) & = & f(u, v) + f'(u, v) \\ & = & -f(v, u) - f'(v, u) \\ & = & -(f + f')(v, u). \end{array}$$

3. Comme f et f' sont des flux, il vérifient la propriété de conservation du flot, $\sum_{v \in S} f(u, v) = -\sum_{w \in S} f(w, u)$ et $\sum_{v \in S} f'(u, v) = -\sum_{w \in S} f'(w, u)$. On obtient alors les égalités suivantes :

$$\begin{array}{rcl} \sum_{v \in S}(f + f')(u, v) & = & \sum_{v \in S}(f(u, v) + f'(u, v)) \\ & = & \sum_{v \in S} f(u, v) + \sum_{v \in S} f'(u, v) \\ & = & -\sum_{v \in S} f(w, v) - \sum_{w \in S} f'(w, v) \\ & = & -\sum_{v \in S}(f + f')(w, v). \end{array}$$

4. La valeur du flot $(f + f')$ est donnée par :

$$\begin{array}{rcl} |(f+f)'| & = & \sum_{v\in S}(f+f')(s,v) \\ & = & \sum_{v\in S} f(s,v) + \sum_{v\in S} f'(s,v) \\ & = & |f| + |f'| \end{array}$$

■

11.1.2 Chemins améliorants

Soit R_f le réseau résiduel induit par le flot f dans le réseau de transport R. Un chemin améliorant de R_f est un chemin élémentaire de R_f de la source s vers le puit p. On appelle *capacité résiduelle* d'un chemin améliorant e

$$c_f(e) = Minimum\{c_f(u,v) \mid (u,v) \in e\}.$$

Le lemme suivant, dont la preuve est évidente, montre la relation entre un chemin améliorant d'un réseau résiduel R_f et le flot f dans R.

Lemme 11.6 *Soit $R = (\langle S, A\rangle, c, s, p)$ un réseau de transport et soit f un flot dans R. Soit e un chemin améliorant de R_f. Soit $f_e : S \times S \longrightarrow \mathbb{R}$ la fonction définie par :*

$$f_e(u,v) = \begin{cases} c_f(e) & si\ (u,v) \in e \\ -c_f(e) & si\ (u,v) \notin e\ et\ (v,u) \in e \\ 0 & sinon \end{cases}$$

Alors f_e est un flot dans R_f de valeur $|f_e| = c_f(e) > 0$.

On obtient comme corollaire que $f + f_e$ est un flot dans R de valeur strictement supérieure à celle de f.

Corollaire 11.7 *Soit $R = (\langle S, A\rangle, c, s, p)$ un réseau de transport et soit f un flot dans R. Soit e un chemin améliorant de R_f et soit $f_e$la fonction définie dans le lemme précédente. Alors $f + f_e$ est un flot dans R et $|f + f_e| > |f|$.*

Le théorème fondamental sur le flot maximal dans un réseau de transport affirme que un flot f dans un réseau de transport R est maximal si et seulement si le réseau résiduel R_f induit par f ne possède pas de chemin améliorant. Pour prouver ce théorème, nous avons besoin de la notion de coupe qui est une notion duale de celle de flot.

11.2 Coupe dans un réseau de transport

Une coupe (E, T) dans un réseau de transport $R = (\langle S, A\rangle, c, s, p)$ est une partition de l'ensemble des sommets de R, $S = E \cup T,\ E \cap T = \emptyset$, telle que $s \in E$ et $p \in T$. La *capacité* d'une coupe (E, T) est

$$c(E,T) = \sum_{u\in E}\sum_{v\in T} c(u,v).$$

Une coupe d'un réseau R est *minimale* si sa capacité est le minumum de l'ensemble des capacités de toutes les coupes de R.
Soit f un flot dans R, le *flot net à travers* la coupe (E, T) est

$$f(E,T) = \sum_{u \in E} \sum_{v \in T} f(u,v).$$

Nous allons montrer que la valeur d'un flot f dans un réseau de transport R est égale au flot net à travers une coupe quelconque (E, T) de R, $|f| = f(E, T)$. Pour cela, on utile la notation suivante, dite de la *sommation implicite*,

$$f(X,Y) = \sum_{x \in X} \sum_{y \in Y} f(x,y).$$

Le lemme suivant, dont la preuve est laissée en exercice, contient des propriétés qui seront utiles dans la suite.

Lemme 11.8 *Soit $R = (\langle S, A\rangle, c, s, p)$ un un réseau de transport et soit f un flot dans R. Les propriétés suivantes sont vérifiées :*

1. $f(X,Y) = -f(Y,X),\ \forall X, Y \subset S.$
2. $f(X,X) = 0,\ \forall X \subset S.$
3. $f(X \cup Y, Z) = f(X,Z) + f(Y,Z),\ \forall X, Y, Z \subset S$ *avec* $X \cap Y = \emptyset.$
4. $f(Z, X \cup Y) = f(Z,X) + f(Z,Y),\ \forall X, Y, Z \subset S$ *avec* $X \cap Y = \emptyset.$
5. $|f| = f(S,p).$

Lemme 11.9 *Soit $R = (\langle S, A\rangle, c, s, p)$ un un réseau de transport, soit (E, T) un coupe de R et soit f un flot dans R. Alors, le flot net à travers (E, T) est $f(E, T) = |f|$.*

<u>*Preuve*</u> Comme $E \cup T = S$ et $E \cap T = \emptyset$, et comme $E = s \cup (E \setminus s)$ et $s \cap (E \setminus s)$, par le Lemme 11.8, on obtient :

$$\begin{array}{rcl} f(E,T) & = & f(E,S) - f(E,E) \\ & = & f(E,S) \\ & = & f(s,S) + f((E \setminus s), S) \end{array}$$

La loi de conservation du flot implique $f((E \setminus s), S) = 0$. On obtient que $|f| = f(s, S)$, ce qui prouve le lemme.

■

On a le corollaire suivant :

Corollaire 11.10 *La valeur d'un flot qualconque dans un réseau de transport est bornée supérieurement par la capacité d'une coupe quelconque.*

Preuve. Soit (E, T) une coupe pour un réseau de transport $R = (\langle S, A\rangle, c, s, p)$. D'aprs le Lemme 11.9, on a :

$$\begin{array}{rcl} |f| & = & f(E,T) \\ & = & \sum_{u\in E}\sum_{v\in T} f(u,v) \\ & \leq & \sum_{u\in E}\sum_{v\in T} c(u,v) \\ & = & c(E,T). \end{array}$$

■

On obtient, comme conséquence, que la valeur maximale d'un flot est bornée supérieurement par la capacité d'une coupe minimale. Nous pouvons prouver le th éorème fondamental sur le flot maximal, connu sous le nom de Théorème du flot maximal et de la coupe minimale.

Théorème 11.11 *Si f est un flot dans un réseau de transport $R = (\langle S, A\rangle, c, s, p)$, alors les conditions suivantes sont équivalentes :*

1. *f est un flot maximal.*
2. *Le réseau résiduel R_f ne contient aucun chemin améliorant.*
3. *$|f| = f(E, T)$ pour une certaine coupe (E, T).*

Preuve

$1 \longrightarrow 2$ Supposons, par l'absurde, que f soit un flot maximal pour R et que R_f contienne un chemin améliorant. D'après le Corollaire du Lemme 11.6, on peut définir dans R un flot dont la valeur est strictement supérieure à celle de f, ce qui est impossible car f est maximal.

$2 \longrightarrow 3$ Supposons que R_f ne contienne aucun chemin améliorant. On définit :

$$E = \{v \in S \mid \exists \text{ un chemin de } s \text{ ver } p \text{ dans } R_f\} \text{ et } T = S \setminus E.$$

La partition (E, T) de S est une coupe pour R. En effet, $s \in E$ de façon triviale et $p \in T$ car, par hypothèse, il n'existe aucun chemin de s vers p dans R_f. Pour chaque couple de sommets u, v tel que $u \in E$ et $v \in T$, on a $f(u, v) = c(u, v)$, puisque, dans le cas contraire ($c_f(u, v) \neq 0$) on aurait $(u, v) \in A_f$ et donc $v \in E$. D'après le Lemme 11.9, on a $|f| = f(E, T) = c(E, T)$.

$3 \longrightarrow 1$ D'après le corollaire du Lemme 11.9, $|f| \leq c(E, T)$ pour toute coupe (E, T). La condition $|f| = c(E, T)$ implique que f est un flot de valeur maximale.

■

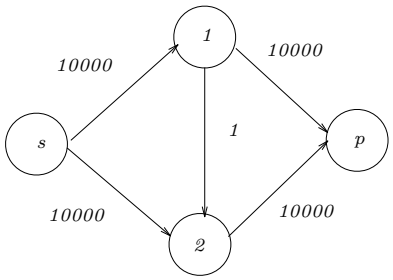
11.3 L'algorithme de Ford-Fulkerson

Par le Théorème 11.11, on peut tester la maximalité d'un flot f dans un réseau de transport R en vérifiant la présence dans R_f d'un chemin de s vers p. La méthode de Ford-Fulkerson est implantée par l'algorithme suivant :

Algorithme *Ford-Fulkerson*
{Entrée : Un réseau de transport $R = (\langle S, A\rangle, c, s, p)$}
{Sortie : Un flot maximal f dans R}
Début
Pour chaque arc $(u, v) \in A$ **faire**
Début
$f(u, v) \longleftarrow 0$;
$f(v, u) \longleftarrow 0$
Finpour
Construire R_f;
Tantque il existe un chemin e de s vers p dans R_f **faire**
Début
$c_f(e) \longleftarrow Minimum\{c_f(u, v) \mid (u, v) \in e\}$;
Pour chaque arc $(u, v) \in A$ **faire**
Début
Si $(u, v) \in e$ alors $f(u, v) \longleftarrow f(u, v) + c_f(e)$;
Si $(v, u) \in e$ alors $f(u, v) \longleftarrow f(u, v) - c_f(e)$
Finpour
Construire R_f;
Fintanque
Finalgo

La complexité de l'algorithme de Ford-Fulkerson depend de la manière dont on calcule un chemin améliorant.

Exemple 11.12 *Considérons le réseau de transport R suivant*



Un flot maximal sur ce réseau a une valer de 20.000 : 10.000 unités de flot traversent le chemin $s \longrightarrow 1 \longrightarrow p$ et 10.000 unités traversent le chemin $s \longrightarrow 2 \longrightarrow p$. Si

le chemin trouvé par l'algorithme de Ford-Fulkerson est $s \longrightarrow 1 \longrightarrow 2 \longrightarrow p$ le flot prend la valeur 1 après la première itération. Si le deuxième chemin améliorant trouvé par l'algorithme de Ford-fulkerson est $s \longrightarrow 2 \longrightarrow 1 \longrightarrow p$. Le flot prend la valeur 2. On peut continuer ainsi, en choisissant, comme chemin améliorant, le chemin $s \longrightarrow 1 \longrightarrow 2 \longrightarrow p$ aux itérations impaires et le chemin $s \longrightarrow 2 \longrightarrow 1 \longrightarrow p$ aux itérations paires. Les augmentations successives seraient alors au nombre de 20.000, augmentant la valeur du flot d'une seule unité à chaque fois. La complexité de l'algorithme de Ford-Fulkerson peut-être améliorée avec une recherche en largeur du chemin amĺiorant dans R_f où l'on suppose que chaque arc ait une pondération unitaire. Cette implantation de l'algorithme de Ford-Fulkerson a pour nom Algorithme de Edmonds-Karps. On peut montrer que l'algorithme de edmonds-Karps s'exécute avec une complexité de $\mathcal{O}(|S| \times |A|^2)$.