# Chapitre 7

# Graphes

## 7.1 Notions et jargons

Un *graphe orienté* $G$ est une paire $(V, E)$, où $V$ est un ensemble fini et $E$ est une relation binaire définie sur $V$ (i.e. un sous-ensemble de $V \times V$). Les éléments de $V$ sont appelés les *sommets* et les éléments de $E$ sont appelés les *arcs* du graphe $G$.

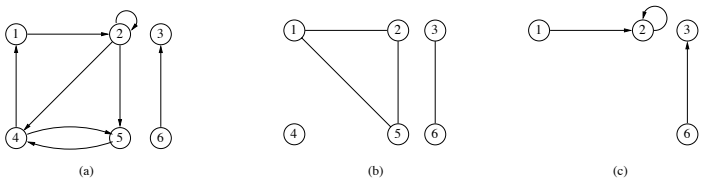



FIGURE 7.1 – Exemples de graphe orienté (a), non-orienté (b) et de sous-graphe induit, (c) est le sous-graphe de (a) induit par le sous-ensemble de sommets $\{1, 2, 3, 6\}$

Un *graphe non-orienté* $G$ est une paire $(V, E)$, où $V$ est un ensemble fini et $E$ est une relation binaire symétrique définie sur $V$, i.e. $(u, v) \in E \Rightarrow (v, u) \in E$. Les éléments de $E$ sont appelés les *arêtes* de $G$.

Un grand nombre de systèmes ou de structures peuvent être considérés de façon abstraite comme un ensemble d'éléments, certains d'entre eux étant en relation, d'autres pas. Il est alors souvent profitable de les représenter à l'aide d'un graphe.

Si $(u, v)$ est un arc du graphe orienté $G = (V, E)$, on dit que l'arc $(u, v)$ est *incident* aux sommets $u$ et $v$ ou encore que l'arc $(u, v)$ quitte le sommet $u$ et arrive dans le sommet $v$. Dans le cas non-orienté, on dit simplement que l'arête $\{u, v\}$ est incidente aux sommets $u$ et $v$.

Dans les deux cas, le sommet $v$ est *adjacent* au sommet $u$. Si le graphe est non-orienté la relation adjacence est symétrique.

Le *degré* d'un sommet $u$ d'un graphe non-orienté est le nombre d'arêtes incidentes à ce sommet. Dans un graphe orienté, on désigne par *demi-degré extérieur* de $u \in V$ le nombre d'arcs qui quittent le sommet $u$, par *demi-degré intérieur* le nombre d'arcs qui arrivent dans le sommet $u$. Le *degré* d'un sommet d'un graphe orienté est la somme des demi-degrés intérieur et extérieur.

Un *chemin* de longueur $k$ entre un sommet $u$ et un sommet $u'$ d'un graphe $G = (V, E)$ est une séquence de sommets $v_0, v_1, \ldots, v_k$ tels que $u = v_0$, $u' = v_k$, et $(v_{i-1}, v_i) \in E$ pour $i = 1, \ldots, k$. On dit qu'un sommet $v$ est *atteignable* à partir d'un sommet $u$ si il existe un chemin entre $u$ et $v$. Un *cycle* est un chemin $v_0, v_1, \ldots, v_k$ tel que $k > 0$ et $v_0 = v_k$. Un chemin est dit *élémentaire* si tous ses sommets sont distincts (sauf éventuellement les deux extrémités, dans le cas d'un cycle élémentaire). Un graphe qui ne contient aucun cycle est dit *acyclique*.

Un graphe non-orienté est *connexe* si il existe un chemin entre chaque paire de sommets. Les *composantes connexes* sont les classes d'équivalence de la relation "être atteignable".

Un graphe orienté est *fortement connexe* si pour chaque paire de sommets $u$ et $v$ il existe un chemin entre $u$ et $v$ et un chemin entre $v$ et $u$. Les *composantes fortement connexes* sont les classes d'équivalence de la relation d'équivalence "être mutuellement atteignable".

On dit qu'un graphe $G' = (V', E')$ est un *sous-graphe* de $G = (V, E)$ si $V' \subseteq V$ et $E' \subseteq E$. Etant donné $V' \subseteq V$, le *sous-graphe induit* par $V'$ est le graphe $G' = (V', E')$ avec $E' = \{(u, v) \in E \ : \ u, v \in V'\}$.

On a donné des noms à certains graphes particuliers. Un *graphe complet* est un graphe non-orienté dans lequel chaque sommet est adjacent à tous les

autres. Un *graphe biparti* est un graphe non-orienté $G = (V, E)$ pour lequel il existe une partition de l'ensemble $V$ des sommets en deux sous-ensembles $V_1$ et $V_2$ telle que toutes les arêtes $\{u, v\} \in E$ relient un sommet de $V_1$ à un sommet de $V_2$. Un graphe non-orienté, connexe et acyclique est appelé un *arbre*. Un graphe non-orienté, acyclique est appelé une *forêt*.

## 7.2 Représentation

Il existe de nombreuses façons de représenter des graphes. On introduit ici la représentation par listes d'adjacence et la représentation par matrice d'adjacence.

### 7.2.1 Listes d'adjacence

Pour chaque sommet, on stocke la liste des sommets adjacents à celui-ci. En C, on utilise un tableau de pointeurs. Le pointeur `Adj[i]` contient l'adresse du premier maillon d'une liste chaînée. Les informations portées par les maillons de cette liste chaînée sont les numéros des sommets adjacents au sommet numéro `i`. L'encombrement mémoire de cette représentation est en $O(|V| + |E|)$.

```
typedef struct maillon {
  int sommet;
  struct maillon *suiv;
} MAILLON,*POINTEUR;

POINTEUR Adj[NB_SOMMETS];
```

### 7.2.2 Matrice d'adjacence

On stocke une matrice `MatAdj[][]` dont la taille est $|V| \times |V|$ et qui est définie de la façon suivante : si $(u, v) \in E$ alors `MatAdj[u][v]` $= 1$ et 0 sinon. L'encombrement mémoire de cette représentation est en $O(|V|^2)$.

```
short MatAdj[NB_SOMMETS][NB_SOMMETS];
```

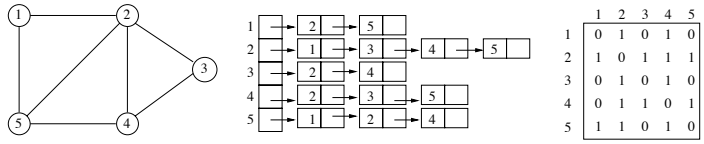



FIGURE 7.2 – Représentation d'un graphe par listes et matrice d'adjacence

### 7.2.3 Commentaires

La représentation par listes d'adjacence est souvent privilégiée parce qu'elle permet de représenter les graphes *creux* d'une façon plus compacte. Les graphes creux (par opposition aux graphes denses) sont ceux pour lesquels $|E|$ est beaucoup plus petit que $|V|^2$. La représentation par matrice d'adjacence est néanmoins intéressante si le graphe est dense ou si on doit souvent savoir si deux sommets sont reliés par une arête. Dans ce dernier cas, la représentation par listes d'adjacence est particulièrement mal adaptée.

**Exercice** 10 Un sommet d'un graphe orienté $G$ est un *puits* si son demi-degré intérieur est égal à $|V| - 1$ et si son demi-degré extérieur est égal à 0. Écrivez un algorithme en $O(|V|)$ qui, étant donnée la matrice d'adjacence de $G$, trouve si $G$ admet un sommet puits ou non, et le calcule, le cas échéant.

## 7.3 Calcul de plus courts chemins

Soit $G = (V, E)$ un graphe et $l : E \to \mathbb{R}$, une pondération des arcs. La longueur d'un chemin $p = (v_0, v_1, ..., v_k)$ est définie comme étant la somme des longueurs de ses arcs :

$$l(p) = \sum_{i=1}^{k} l(v_{i-1}, v_i) \text{ et } l(p) = 0 \text{ si } k = 0.$$

On note $\delta(u, v)$ la longueur d'un plus court chemin de $u$ à $v$. On appelle *plus court chemin* entre $u$ et $v$, un chemin dont la longueur est $\delta(u, v)$.

**Formulation du problème :** étant donné un graphe $G = (V, E)$, trouver un plus court chemin d'un sommet *source* $s \in V$ à tous les autres sommets du graphe.

**Remarque 1 :** on ne connait pas de façon de calculer un plus court chemin entre deux sommets $u$ et $v$ sans calculer des plus courts chemins de $u$ (ou de $v$) à tous les autres sommets du graphe.
**Remarque 2 :** il existe une variante du problème des plus courts chemins dans laquelle il faut calculer des plus courts chemins entre chaque paire de sommets du graphe.

### 7.3.1 Algorithme de Dijkstra

L'algorithme de Dijkstra permet de trouver des plus courts chemins d'un sommet $s$ donné à tous les autres sommets quand les pondérations des arcs sont *positives*. Dans cette partie, on supposera $l(u, v) \geq 0$ quelque soit $(u, v) \in E$.

Il consiste à partitionner l'ensemble des sommets en deux sous-ensembles $S$ et $V - S$. On maintient en permanence pour chaque sommet $u$ du graphe la longueur `dist[u]` du plus court chemin entre $s$ et $u$ dont tous les sommets intermédiaires sont dans $S$. A chaque étape, on sélectionne un sommet $u$ de $V - S$ pour lequel `dist[u]` est minimum. On ajoute $u$ à $S$ et on met à jour les longueurs estimées des sommets de $V - S$ adjacents à $u$.

**Algorithme 25:** Algorithme de Dijkstra

```
pour v ∈ V faire
    dist[v]=INFINI;
    pred[v]=AUCUN;
dist[s]=0;
S ← ∅;
F ← V;
tant que F ≠ ∅ faire
    u ← ExtraireMinimum(F)(=ArgMin_{v∈F} d[v]);
    S ← S ∪ {u};
    pour chaque v ∈ Adj[u] faire
        si dist[v] > dist[u] + l(u,v) alors
            dist[v] ← dist[u] + l(u,v);
            pred[v] ← u
```

**Théorème 1** *A la fin de l'exécution de l'algorithme de Dijkstra* `dist[u]` $= \delta(s, u)$ *pour tout* $u \in V$.

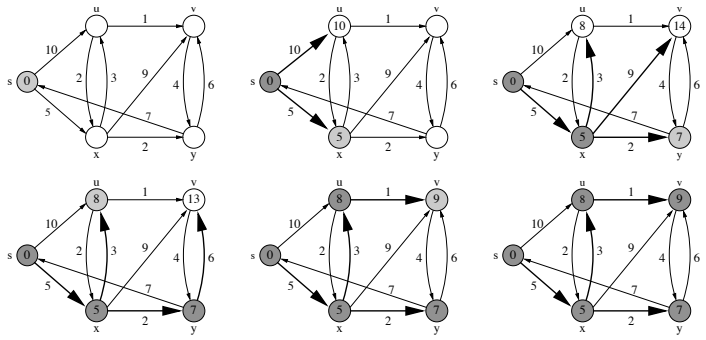



FIGURE 7.3 – Illustration de l'algorithme de Dijkstra sur un exemple.

**Preuve** On supposera pour simplifier la preuve que tous les sommets sont atteignables à partir de $s$.

Montrons la correction de l'algorithme de Dijkstra en considérant l'invariant : $\forall u \in S, dist[u] = \delta(s,u)$.

La propriété est vraie si $S = \emptyset$ ou lorsque $S = \{s\}$ et $dist[s] = 0$ (puisque les pondérations ne sont pas négatives). Supposons qu'elle soit vraie pour $|S| = k$, où $1 \leq k < |V|$.

Soit $u$ le nouveau sommet rajouté à $S$. Montrons que $dist[u] = \delta(s,u)$.

Soit $s \cdots x - y \cdots u$ un plus court chemin reliant $s$ à $u$, où $x$ et $y$ sont consécutifs et $x$ est le dernier sommet dans $S$. On a :

$$\begin{aligned}
\delta(s,u) &= l(s \cdots x - y \cdots u) \\
&\geq l(s \cdots x - y) && \text{car les poids ne sont pas négatifs} \\
&= \delta(s,x) + l(x,y) && \text{par définition de } \delta \\
&= dist[x] + l(x,y) && \text{par hypothèse de récurrence} \\
&\geq dist[y] \\
&\geq dist[u] && \text{puisque c'est } u \text{ qui a été choisi.}
\end{aligned}$$

Par ailleurs, comme $dist[u] < \infty$, il existe un sommet $z \in S$ tel que $dist[u] = dist[z] + l(z,u) = \delta(s,z) + l(z,u) \leq \delta(s,u)$. Donc, $dist[u] = \delta(s,u)$.

□

**Exemple :** Pour l'exemple 7.3, on pourra représenter les calculs de la manière suivante :

| pred | sommet \ étape | | | | | |
|---|---|---|---|---|---|---|
| | s | 0 | | | | |
| ~~$x$~~ | u | $\infty$ | 10 | 8 | 8 | |
| s | x | $\infty$ | 5 | | | |
| ~~$x$~~ ~~$y$~~ u | v | $\infty$ | $\infty$ | 14 | 13 | 9 |
| x | y | $\infty$ | $\infty$ | 7 | | |
| | $S$ | s | x | y | u | v |

Le plus court chemin de s à u est : s - x - u - v. Sa longueur est 9.

**Complexité :** Si $F$ est simplement implanté par un tableau, l'opération d'extraction du minimum est en $O(|V|)$. De plus, comme $|V|$ opérations de ce type sont effectuées, cette partie de l'algorithme est donc en $O(|V|^2)$. L'autre partie consiste à mettre à jour les distances estimées en parcourant les listes d'adjacences. Chaque liste d'adjacence est parcourue une seule fois et on sait que la somme des longueurs de ces listes est en $O(|E|)$. L'algorithme est donc en $O(|V|^2 + |E|)$.

Si le graphe est creux, on obtient un gain de complexité en implantant $F$ par une *file de priorité*[1]. Chaque opération d'extraction du minimum est alors en $O(\log |V|)$ et comme précédemment, $O(|V|)$ telles opérations sont effectuées. La création initiale du tas est en $O(|V|)$. Dans un tas binaire, la mise à jour du tas après l'affectation `dist[v]` $=$ `dist[u]` $+ l(u, v)$ se fait également en $O(\log |V|)$ et de nouveau, il y a au plus $|E|$ telles opérations. Dans le cas des graphes creux et en utilisant un tas binaire, l'algorithme de Dijkstra est en $O((V + E) \log V)$.

**Exercice** 11 Écrivez l'algorithme `DiminueClé(T[1,n], i, e)` qui prend en entrée un *tasmin* $T$, un indice $1 \leq i \leq n$ et un réel positif $e$ et qui retourne le tasmin $T$ dans lequel $T[i]$ a été modifié en $T[i] - e$ (la structure de tasmin est préservée). L'algorithme doit fonctionner en temps $O(\log n)$.

**Exercice** 12 Montrez comment on peut adapter l'algorithme de Dijsktra de manière qu'il calcule les distances minimales de tous les sommets $v$ d'un graphe orienté $G = (V, E)$ à un sommet cible $t \in V$. Appliquez cette variante à l'exemple ci-dessus, en prenant $t = y$.

1. Une file de priorité est une structure permettant de maintenir un ensemble $F$ de nombres avec une implémentation efficace des opérations suivantes : extraction du plus petit élément, insertion ou suppression d'un élément et modification d'un élément. On les représente usuellement pas des *tasmin*, c'est-à-dire des tas vérifiant la propriété *minHeap*.

## 7.4 Algorithme de Bellman-Ford

Nous allons voir comment résoudre, grâce à la programmation dynamique, le problème de la recherche d'un plus court chemin entre deux sommets $s$ et $t$ d'un graphe orienté $G = (V, E)$ dans le cas où il peut y avoir des arcs de longueurs négatives mais pas de cycle de longueur négative. Dans cette section, on notera $n := |V|$ le nombre de sommets et $m := |E|$ le nombre d'arcs du graphe $G$.

L'approche que nous allons décrire est basée sur l'observation suivante : *Si le graphe $G$ ne contient aucun cycle de longueur négative, alors il existe un plus court chemin élémentaire (sans répétition de noeuds) entre $s$ et $t$ qui possède au plus $n-1$ arcs.* En effet, si un chemin $\pi$ reliant $s$ à $t$ possède au moins $n$ arcs, il passe par au moins $n+1$ sommets. Par le principe des tiroirs [2], on en déduit qu'un même sommet apparaît au moins deux fois. Le chemin $\pi$ contient donc un cycle. Si ce cycle a une longueur totale positive, $\pi$ n'est pas le chemin le plus court entre $s$ et $t$. Si ce cycle a une longueur totale égale à 0, il existe un chemin qui relie $s$ à $t$, de longueur égale à celle de $\pi$ et possédant moins d'arcs.

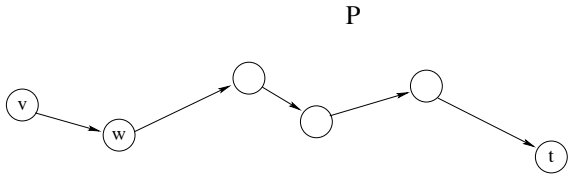



FIGURE 7.4 – Un plus court chemin allant de $v$ à $s$ en utilisant au plus $i$ arcs

Notons $OPT(i, v)$ la longueur minimale d'un chemin de $v$ à $t$ contenant au maximum $i$ arcs. A cause de l'observation précédente, notre problème est de calculer $OPT(n-1, s)$. Notre but est de calculer $OPT(i, v)$ en utilisant des sous-problèmes. Considérons un chemin $P$ optimal pour le sous-problème $OPT(i, v)$ (voir Figure 7.4) :

- si le chemin $P$ utilise au plus $i-1$ arcs, alors $OPT(i, v) = OPT(i-1, v)$ ;
- si le chemin $P$ utilise exactement $i$ arcs, soit $(v, w)$ le premier arc de $P$, alors $OPT(i, v) = l(v, w) + OPT(i-1, w)$.

Il n'existe aucune autre alternative, on obtient donc la formule récursive

2. si $n$ tiroirs contiennent $n+1$ objets, au moins 1 tiroir contient au moins 2 objets.

suivante (pour $i > 0$)

$$OPT(i, v) = \min(OPT(i-1, v), \min_{w \in V}(l(v, w) + OPT(i-1, w))). \quad (7.1)$$

En utilisant, la formule de récurrence 7.1, on obtient l'algorithme de programmation dynamique suivant pour calculer $OPT(n-1, s)$.

**Algorithme 26:** Algorithme de Bellman-Ford

**entrée** : Un graphe orienté $G$ et deux sommets $s$ (source) et $t$ (cible).
**pour** $v \in V \setminus \{t\}$ **faire**
  $M[0, v] = \infty$
$M[0, t] = 0$
**pour** $i = 1$ **à** $n-1$ **faire**
  **pour** $v \in V$ **faire**
    $M[i, v] = \min(M[i-1, v], \min_{w \in V}(l(v, w) + M[i-1, w]))$
**retourner** $M[n-1, s]$

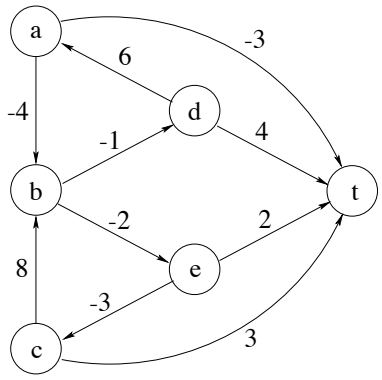
La correction de cet algorithme se déduit par induction de la formule 7.1. Chacune des $n^2$ entrées de la table $M$ est calculé en temps $O(n)$. L'algorithme de Bellman-Ford calcule donc correctement la longueur d'un plus court chemin entre $s$ et $t$ dans tout graphe $G$ qui ne contient aucun cycle de longueur négative et s'exécute en temps $O(n^3)$.

Par exemple, considérons le graphe de la Figure 7.6(a), le but est de calculer pour chaque sommet $v \neq t$ un plus court chemin allant $v$ à $t$. La table 7.6(b) présente les valeurs $M[i, v]$ obtenus par l'algorithme de Bellman-Ford. Par conséquent, une ligne de la matrice contient les longueurs des plus courts chemins allant d'un sommet donné à $t$, alors qu'on fait croître progressivement le nombre d'arcs autorisés dans un chemin. Par exemple, le plus court chemin allant de $d$ à $t$ est mis à jour 4 fois puisqu'il passe de $d-t$ à $d-a-t$, puis $d-a-e-t$, et finalement $d-a-b-e-c-t$.

### 7.4.1 Amélioration du temps de calcul

Il est possible d'améliorer les temps de calcul dans le cas où le nombre d'arcs $m$ est beaucoup plus petit que $n(n-1)$ (le nombre maximum d'arcs du graphe). En fait, l'algorithme de Bellman-Ford peut-être implémenté de façon à s'exécuter en temps $O(mn)$. En effet, lors de l'évaluation de l'expression

$$\min_{w \in V}(l(v, w) + M[i-1, w])$$




(a)

| | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| $t$ | 0 | 0 | 0 | 0 | 0 | 0 |
| $a$ | $\infty$ | $-3$ | $-3$ | $-4$ | $-6$ | $-6$ |
| $b$ | $\infty$ | $\infty$ | 0 | $-2$ | $-2$ | $-2$ |
| $c$ | $\infty$ | 3 | 3 | 3 | 3 | 3 |
| $d$ | $\infty$ | 4 | 3 | 3 | 2 | 0 |
| $e$ | $\infty$ | 2 | 0 | 0 | 0 | 0 |

(b)

FIGURE 7.5 – Pour le graphe de la figure (a), l'algorithme de Bellman-Ford construit la table de la figure (b).

dans le calcul de la formule 7.1, il suffit de considérer les sommets $w$ qui sont des voisins de $v$ (i.e. $(v, w)$ est arc du graphe). Le coût d'évaluation de chacune des entrées $M[i, v]$ de la table est donc en $O(n_v)$. Il y a une entrée à évaluer pour chaque sommet $v \in V$ et pour chaque indice $0 \leq i \leq n-1$. Par conséquent, le coût global de l'algorithme est

$$O(n \sum_{v \in V} n_v).$$

Rappelons que, pour un graphe orienté, la somme des longueurs des listes d'adjacences $\sum_{v \in V} n_v$ est égale à $m$ (chaque arc est compté une fois). L'algorithme de Bellman-Ford s'exécute donc en temps $O(mn)$.

### 7.4.2 Amélioration de l'espace mémoire

Pour diminuer l'espace mémoire utilisé par l'algorithme de Bellman-Ford, on peut garder une seule valeur $M[v]$ pour chaque sommet $v$ au lieu de mémoriser $M[i, v]$ pour chaque valeur de $i$. Dans ce cas, $i$ devient simplement un compteur et pour $i = 1, \ldots, n-1$, on effectue la mise à jour suivante

$$M[v] = \min(M[v], \min_{w \in V}(l(v, w) + M[w])).$$

La correction de cette nouvelle version repose sur l'observation suivante. *Tout au long de l'exécution de l'algorithme, $M[v]$ représente la longueur*

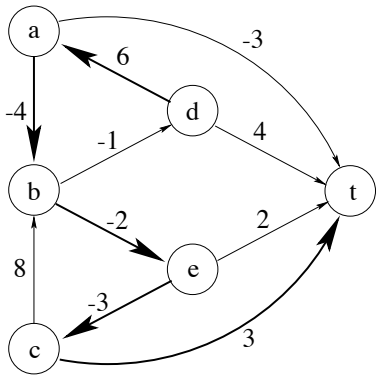
*d'un chemin allant de $v$ à $t$, et après $i$ mises à jour la valeur $M[v]$ est au plus la longueur d'un plus court chemin entre $v$ et $t$ contenant au maximum $i$ arcs.* Les plus courts chemins étant élémentaires $n-1$ mises à jour suffisent pour que $M[s]$ soit la longueur d'un court chemin entre $s$ et $t$.

### 7.4.3 Construction du chemin

Pour aider à la reconstruction du chemin, nous allons maintenir une information supplémentaire `premier`$[v]$ qui représente le premier sommet après $v$ dans le chemin de longueur $M[v]$ entre $v$ et $t$. Pour maintenir `premier`$[v]$, il suffit de modifier sa valeur à chaque fois que $M[v]$ est modifié. En d'autres termes, lorsque la valeur de $M[v]$ est remplacé par le minimum $\min_{w \in V}(l(v,w) + M[w])$, `premier`$[v]$ est mis à jour avec le sommet $w$ qui permet d'atteindre ce minimum. Considérons le graphe $P$ avec $V$ comme ensemble de sommets et tous les arcs de la forme $\{(v, \texttt{premier}(v)\}$ comme ensemble d'arcs. Tout d'abord remarquons que, si le graphe $P$ contient un cycle $C$, alors ce cycle est de longueur négative. Nous avons supposé que le graphe $G$ ne contient aucun cycle de longueur négative donc le graphe $P$ est acyclique. Considérons maintenant le chemin que nous obtenons en partant $v$ et en suivant les arcs de $P$, de $v$ à $\texttt{premier}[v] = v_1$, puis de $v_1$ à $\texttt{premier}[v_1] = v_2$ ainsi de suite. Le graphe $P$ ne contenant aucun cycle et $t$ étant le seul sommet de $P$ n'ayant aucun arc sortant, ce chemin conduit nécessairement à $t$. On peut montrer que, quand l'algorithme se termine, ce chemin est un plus court chemin entre $v$ et $t$.

## 7.5 Protocoles de routage

Une application importante des algorithmes de plus courts chemins concerne le routage des informations dans les réseaux de communications. Il s'agit de déterminer le chemin le plus efficace pour router les messages jusqu'à leurs destinations. Le réseau de communication est représenté par un graphe dans lequel les sommets correspondent aux routeurs ; il y a un arc $(v, w)$ si les routeurs correspondant aux sommets $v$ et $w$ sont directement connectés par une ligne de communications orientée de $v$ vers $w$. La longueur $l(v, w)$ de l'arc $(v, w)$ représente le délai d'acheminement d'un message sur la ligne $(v, w)$. Les délais sont naturellement non négatifs, par conséquent, on pourrait utiliser l'algorithme de Dijkstra pour calculer les plus courts chemins. Cependant, cet algorithme requiert une représentation globale du réseau : il maintient un sous-ensemble $S$ de noeuds du réseau pour lesquels les plus courts chemins sont connus et, à chaque étape, il prend une décision globale




(a)

| | $a$ | $b$ | $c$ | $d$ | $e$ |
|---|---|---|---|---|---|
| $M$ | $-6$ | $-2$ | 3 | 0 | 0 |
| `premier` | $b$ | $e$ | $t$ | $a$ | $c$ |

(b)

FIGURE 7.6 – Pour le graphe de la figure (a), l'algorithme de Bellman-Ford construit les tables de la figure (b). Les arcs de $P$ sont en gras.

pour choisir un sommet qui rejoint $S$. Plutôt que de devoir collecter toutes les informations nécessaires à travers le réseau, il est préférable d'utiliser un algorithme capable de travailler avec des informations locales concernant seulement ses voisins dans le réseau.

Il se trouve que l'algorithme de Bellman-Ford a justement cette propriété de "localité". Supposons que chaque noeud $v$ maintient sa propre variable $M[v]$. Pour mettre à jour cette variable, ce noeud a besoin seulement de connaître les valeurs $M[w]$ de chacun de ses voisins pour calculer

$$\min_{w \in W} (l(v, w) + M[w]).$$

Nous allons maintenant présenter une amélioration de l'algorithme de Bellman-Ford qui le rend plus efficace et en même temps mieux adapté aux problèmes de routage.

Dans l'algorithme précédent, à chaque itération, un sommet $v$ a besoin, pour recalculer sa marque $M[v]$, de demander à chacun de ses voisins $w$ leur propre marque. Il se peut très bien qu'aucun des voisins n'ait changé de marque depuis la dernière mise à jour de $M[v]$ et dans ce cas, il est inutile de recalculer $M[v]$. L'amélioration que nous allons présenter permet d'éviter ces calculs superflus. Pour cela, chaque noeud dont la marque est modifiée en informe ses voisins, qui devront donc à leur tour mettre à jour leur propre marque. Cette façon de procéder permet aussi de pouvoir arrêter l'algorithme si aucune marque ne change pendant une itération.

**Algorithme 27:** Algorithme de Bellman-Ford-Version-2

```
entrée : Un graphe orienté G et deux sommets s (source) et t (cible).
pour v ∈ V \ {t} faire
    M[v] = ∞
M[t] = 0
pour i = 1 à n − 1 faire
    pour w ∈ V faire
        si M[w] a été modifié à l'itération précédente alors
            pour (v, w) ∈ E faire
                M[v] = min(M[v], l(v, w) + M[w])
                si la valeur de M[v] a été modifiée alors
                    premier[v] = w
    si Aucune valeur n'a été modifiée pendant cette itération alors
        Arrêter l'algorithme.
retourner M[s]
```

Dans cette version, à chaque itération, les noeuds dont la marque a changé envoient un message de notification à chacun de leurs voisins. Cependant, si les noeuds correspondent à des routeurs dans un réseau, nous ne pouvons pas espérer que les choses se déroulent de façon aussi synchronisée. Il se peut que certains noeuds envoient les notifications de changement beaucoup plus vite que d'autres. Par conséquent, les noeuds du réseau exécutent en réalité une version asynchrone de l'algorithme : chaque noeud dont la marque change devient *actif* et doit alors notifier ce changement à ses voisins.

On peut montrer que cette version de l'algorithme, dans laquelle les mises à jour se font dans un ordre quelconque, se termine avec les valeurs exactes.

Il est souvent utile de connaitre les plus courts chemins entre chaque paire de routeurs. Dans ce cas, on exécute l'algorithme pour chacune des destinations en conduisant en parallèle les $n$ calculs.

**Algorithme 28:** Algorithme de Bellman-Ford-Asynchrone

**entrée** : Un graphe orienté $G$ et deux sommets $s$ (source) et $t$ (cible).
**pour** $v \in V \setminus \{t\}$ **faire**
  $M[v] = \infty$
$M[t] = 0$
Déclarer $t$ actif et tous les autres noeuds inactifs.
**tant que** *il existe un noeud actif* **faire**
  Choisir un noeud actif $w$
  **pour** $(v, w) \in E$ **faire**
    $M[v] = \min(M[v], l(v, w) + M[w])$
    **si** *la valeur de* $M[v]$ *a été modifiée* **alors**
      `premier`$[v] = w$
      $v$ devient actif
  $w$ devient inactif