Graphes

■ **Note 1** Une grande partie est issue verbatim de mon ancien poly. À affiner.

■

I Graphes orientés

I.1 Définition

Définition I.1 Un graphe orienté est un couple $G = (S, A)$ où S est un ensemble fini et A est une relation **irréflexive** sur S, c'est-à-dire $A \subset \{(x, y) \mid x, y \in S, x \neq y\}$.

Les éléments de S sont appelés les **sommets** du graphe G et les éléments de A les **arêtes**. Si $(x, y) \in A$, on dit que x est la source de l'arête et y est son but. Quand le contexte n'est pas ambigu, on notera $x \to y$.

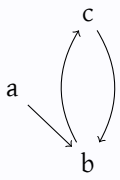
Exemple
- $G = (\{a, b, c\}, \{(a, b), (b, c), (c, b)\})$
- $D_n = (\llbracket 1, n \rrbracket, \{(a, b) \mid a \text{ divise } b, a \neq b\})$

Remarque Ici, on a exclu les graphes infinis ainsi que les graphes avec des boucles ou des arêtes parallèles. On est dans le contexte des **graphes finis simples**.

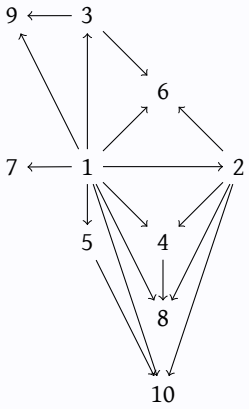
On représente graphiquement un graphe sous la forme d'un diagramme sagittaire où les sommets sont des points et où une arêtes (a, b) est une flèche allant du point a au point b.

Exemple

- G est représenté par



- D_{10} est représenté par



I.2 Voisins et degrés

Définition I.2 Soit $G = (S, A)$ un graphe orienté et $x \in S$.

On appelle **voisins sortants**, ou **successeurs**, de x les éléments de

$$v_+(x) = \{\ y \in S \mid (x, y) \in A\ \}$$

et on appelle **degré sortant** de x le cardinal de cet ensemble $d_+(x) = |v_+(x)|$.

De même, on parle de **voisins entrants**, ou **prédéceseurs** pour les éléments de

$$v_-(x) = \{\ z \in S \mid (z, x) \in A\ \}$$

et on parle de **degré entrant** pour $d_-(x) = |v_-(x)|$.

Théorème I.1 Soit $G = (S, A)$ un graphe, on a $\sum_{x\in S} d_+(x) = \sum_{x\in S} d_-(x) = |A|$

■ **Preuve**

On peut partitionner A en regroupant les arêtes de même source, on a ainsi :

$$A = \bigcup_{x\in S} \{\ (x, y) \mid y \in S, (x, y) \in A\ \} = \bigcup_{x\in S} \{\ (x, y) \mid y \in v_+(x)\ \}$$

Et en prenant la cardinal de cette égalité, on obtient directement $|A| = \sum_{x\in S} d_+(x)$. L'autre égalité est symétrique en considérant les arêtes de même but.

■

I.3 Chemin

Définition I.3 Soit $G = (S, A)$ un graphe orienté et $x, y \in S$.
Une suite finie $\varphi = (s_0, s_1, \dots, s_p)$ où $p \in \mathbb{N}, s_0 = x, s_p = y$ et

$$\forall i \in [\![1, p]\!], (s_{i-1}, s_i) \in A$$

est appelée un **chemin**, de longueur p, de x vers y. On a donc

$$s_0 \to s_1 \to \cdots \to s_p$$

On notera $\varphi : x \rightsquigarrow y$ pour signifier que φ est un tel chemin et $x \rightsquigarrow y$ pour signifier qu'il existe un chemin de x vers y. On dit alors que y est accessible depuis x.

Théorème I.2 $\rightsquigarrow$ est la plus petite relation sur S contenant $\to$ et qui soit réflexive et transitive.

■ **Preuve**
On considère $\to \subset R \subset S^2$ telle que R soit réflexive et transitive.
Soient $x, y \in S$ et $\varphi = (s_0, \dots, s_p) : x \rightsquigarrow y$.
Comme $s_{i-1} \to s_i$ on a $s_{i-1} R s_i$ et, par transitivité, $x = s_0 R s_p = y$. Donc $x \rightsquigarrow y \Rightarrow xRy$.
Ainsi $\rightsquigarrow \subset R$.

■

On dit que $\rightsquigarrow$ est la fermeture réflexive et transitive de $\to$.

Remarque Vocabulaire additionnel :
- Si tous les sommets sont distincts, sauf éventuellement le premier et le dernier, on dit que le chemin est **élémentaire**.
- Si toutes les arêtes sont distinctes, on dit que le chemin est **simple**.
- Si $x = y$, on dit que le chemin est **fermé**.
- Un chemin simple dont tous les sommets sont distincts est appelé une **chaîne**.
- Un chemin élémentaire fermé simple de longueur au moins 1 est appelé un **cycle**. *Comme le chemin vide issu de x est fermé, il est nécessaire de considérer des chemins non vides.*
- Un graphe contenant au moins un cycle est dit **cyclique**. Dans le cas contraire, on dit qu'il est **acyclique**.

Remarque Si $\varphi : x \rightsquigarrow y$ et $\psi : y \rightsquigarrow z$ on note $\varphi\psi : x \rightsquigarrow z$ la concaténée des deux chemins.

Définition I.4 On note $\leftrightarrow y$ la plus grande relation d'équivalence incluse dans $\rightsquigarrow$.
Plus précisément, on a

$$x \leftrightarrow y \iff (x \rightsquigarrow y \wedge y \rightsquigarrow x)$$

Les classes d'équivalences pour $\leftrightarrow$ sont appelées les composantes **fortement connexes** du graphe. S'il n'y a qu'une classe, on dit que le graphe est **fortement connexe**.

I.4 Sous-graphe

Définition I.5 Soit $G = (S, A)$ et $X \subset S$. On appelle **sous-graphe** induit par X le graphe (X, A_X) où

$$A_X = \{\ (a, b) \in A \mid a \in X \wedge b \in X\ \}$$

Un graphe G' est un **sous-graphe** de G quand c'est le sous-graphe de G induit par les sommets de G' (ils sont alors nécessairement des sommets de G).

Remarque Les composantes fortement connexes sont les sous-graphes fortement connexes maximaux pour l'inclusion.

I.5 Implémentation

I.5.i Énumération de sommets

Si on considère un graphe $G = (S, A)$, il est assez naturel de représenter ses sommets dans un tableau. Pour cela, on fixe naturellement un ordre sur ces sommets et on va associer à chaque sommet son indice dans le tableau.

Par exemple, si $S = \{a, b, c, d\}$ on va pouvoir considérer `[ a, b, c, d ]` et ainsi associer à `a` son indice `0` dans le tableau. L'ordre est arbitraire, on aurait pu considérer `[ b, c, a, d ]` et l'indice de `a` aurait alors été 2.

Ce qui compte, c'est de pouvoir travailler directement sur les indices et pas sur les éléments. Une manière de s'en convaincre est d'imaginer le graphe d'un réseau social où un sommet correspond au profil d'une personne, et contient donc beaucoup (*trop*) d'informations. Il est bien plus raisonnable de lui associer un identifiant unique et d'utiliser cet identifiant ensuite.

Quand on va implémenter des graphes, on peut donc supposer que les sommets sont les entiers de `0` à `n-1` où $|S| = n$. Il sera toujours possible de retrouver la correspondance avec les sommets eux-mêmes.

Remarque Si on est bloqué par le coût de l'opération permettant d'obtenir l'indice d'un sommet, qui est en $O(|S|)$ par une recherche linéaire, on peut très bien définir un dictionnaire associant à chaque sommet son indice en $O(1)$.

Il se trouve qu'on ne va pas forcément s'intéresser à ces questions mais plus aux raisonnements sur la structure du graphe.

I.5.ii Matrice d'adjacence

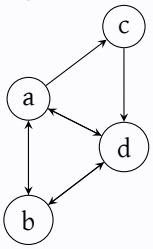
Définition I.6 Soit $G = (S, A)$ un graphe et $S = \{s_1, \dots, s_n\}$ une énumération sans répétition des sommets de G. La matrice $M_G = (m_{ij})_{1 \leq i,j \leq n} \in \mathcal{M}_n(\mathbb{R})$ définie par

$$\forall i, j \in [\![1, n]\!], m_{ij} = \begin{cases} 1 & \text{si } s_i \to s_j \\ 0 & \text{sinon} \end{cases}$$

est appelée une **matrice d'adjacence** de G.

Remarque Il s'agit bien d'**une** matrice d'adjacence et pas de **la** matrice car elle dépend de l'énumération des sommets.

Exemple Si on considère le graphe



On aura pour l'énumération a, b, c, d la matrice :

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

En fait, la ième ligne correspond au ième sommet de l'énumération. Ici, la dernière ligne 1100 correspond au

sommet d et donne dans l'ordre a, b, c, d la présence ou non d'une arête $d \to *$.

Si on considère l'énumération b, a, d, c on aura la matrice :

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

On en déduit ainsi une implémentation directe d'un graphe en représentant la matrice d'adjacence comme l'a fait pour des images.

```python
m = [ [ 0, 1, 1, 1 ],
    [ 1, 0, 0, 1 ],
    [ 0, 0, 0, 1 ],
    [ 1, 1, 0, 0 ] ]
```
Python

C

```ocaml
let m = [| [| 0; 1; 1; 1 |];
     [| 1; 0; 0; 1 |];
     [| 0; 0; 0; 1 |];
     [| 1; 1; 0; 0 |] |]
```
OCaml

Remarque Il y a le même lien entre les différentes matrices d'adjacence d'un graphe donnée et la matrice d'un endomorphisme dans les différentes permutations d'une même base, à chaque fois on obtient la nouvelle matrice en permutant de même ses lignes et ses colonnes.

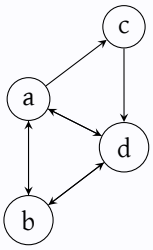
Cette représentation permet d'accéder, en lecture comme en écriture, à une arête en temps constant. Cependant, pour récupérer les voisins d'un sommet, il est nécessaire de parcourir toute la ligne correspondante, donc en $O(|S|)$.

Remarque Dans la majorité des cas, A est petit par rapport à $|S|^2$. Ainsi, la matrice contient très peu de 1 et beaucoup de 0, on dit qu'elle est *creuse*. Une telle matrice *creuse* peut-être efficacement représenté par l'ensemble fini des arêtes (i, j). Par exemple, avec une table de hachage. Cependant, dans le cas des graphes, il est possible d'avoir une structure creuse plus efficace.

A titre d'exemple, un graphe issu d'un réseau social est présenté plus bas, il contient 300000 arêtes pour 8000 sommets. Donc le rapport $|A|/|S|^2$ est de 4 pour 1000 !

I.5.iii Listes d'adjacences

La donnée de $v_+(x)$ pour chaque sommet $x \in S$ suffit à reconstruire A.



Exemple Si on reprend le graphe précédent, on a $S = \{a, b, c, d\}$ et $A = \{(a, b), (a, c), (a, d), (b, a), (b, d), (c$

Or $v_+(a) = \{b, c, d\}, v_+(b) = \{a, d\}, v_+(c) = \{d\}$ et $v_+(d) = \{a, b\}$.

Si on pose $V_+(x) = \{\ (x, y) \mid y \in v_+(x)\ \}$ on a directement $A = \bigcup_{x \in S} V_+(x)$.

On en déduit ainsi une représentation d'un graphe où on place dans un tableau chaque $v_+(x)$ représenté par une liste chaînée ou un tableau dynamique.

Remarque Pour représenter $v_+(x)$ on a plusieurs choix : on peut utiliser directement les valeurs des sommets ou utiliser des indices dans une énumération. C'est le second choix qu'on fera en général car il est plus simple. Cependant, en cas de suppression d'un sommet, les indices changent, et il faut renuméroter dans les listes.

Pour le graphe précédent, on pourra donc considérer une énumeration `[ a, b, c, d ]` et la représenta-

tion sous forme de liste d'adjacence pourra être :

```python
l = [ [ 1, 2, 3 ], [ 0, 3 ], [ 3 ], [
```
Python

```ocaml
let l = [|
  [ 1; 2; 3 ]; [ 0; 3 ]; [ 3 ]; [ 0; 1 ]
  |]
```
OCaml

L'accès en lecture ou en écriture à une arête est alors en $O(|A|)$ mais on peut parcourir les voisins sortant en $O(|A|)$ également. Pour un sommet donné, on peut même préciser $O(d_+(x))$. Accéder à la liste peut même se faire en $O(1)$.

Pour obtenir les voisins entrants, il est par contre nécessaire de tester la présence de x dans chacune des autres listes, on obtient donc un algorithme en $O(|S| + |A|)$: on parcourt chaque case du tableau des listes puis chaque maillon de listes d'adjacence.

Il est possible d'améliorer cela en utilisant un structure plus efficace pour stocker les ensembles. Cela peut-être un dictionnaire reposant sur une table de hachage. L'avantage de cela est que pour tester l'appartenance $y \in v_+(x)$ on sera en $O(1)$ avec un dictionnaire alors qu'on sera en $O(d_+(x))$ avec une liste.

I.5.iv Comparaison

opération	Matrice	Listes	Dictionnaire
complexité spatiale	$O(\|S\|^2)$	$O(\|S\| + \|A\|)$	$O(\|S\| + \|A\|)$
arête test	$O(1)$	$O(\|A\|)$	$O(1)$
arête ajout	$O(1)$	$O(1)$	$O(1)$
arête suppression	$O(1)$	$O(\|A\|)$	$O(1)$
sommet ajout	$O(\|S\|^2)$	$O(\|S\|)$	$O(\|S\|)$
sommet suppression	$O(\|S\|^2)$	$O(\|S\|)$	$O(\|S\|)$
voisins/degré +	$O(\|S\|)$	$O(1)$	$O(1)$
voisins/degré -	$O(\|S\|)$	$O(\|S\| + \|A\|)$	$O(\|S\|)$

Notons qu'il est possible d'améliorer certaines complexités en utilisant des tableaux dynamiques, notamment les ajouts et suppressions de sommets.

Remarque Il semble a priori clair qu'on devrait utiliser des dictionnaires pour manipuler des graphes en voyant ce tableau. Pourtant, on utilisera presque exclusivement des listes d'adjacence. Pourquoi ? En grande partie parce que les opérations rendues efficaces sont rares et ne justifient pas la difficulté accrue de manipulation. En effet, l'opération la plus importante est de pouvoir énumérer les voisins, et une liste chaînée le permet facilement et efficacement.

II Graphes non orientés

II.1 Définition et adaptation du vocabulaire

Définition II.1 Un graphe non orienté est un couple $G = (S, A)$ où S est un ensemble fini et A est un ensemble de paires d'éléments de S, c'est-à-dire d'ensembles à deux éléments $\{x, y\}$ où $x, y \in S$.

Les éléments de S sont appelés les **sommets** du graphe G et les éléments de A les **arêtes**. Si $\{x, y\} \in A$, on note $x \smile y$.

Remarque Si $G = (S, A)$ est un graphe orienté, on peut en déduire deux graphes non orientés :

- un graphe par défaut $G^- = (S, A^-)$ où

$$\forall x, y \in S, x \smile y \iff (x \to y \wedge y \to x)$$

- un graphe par excès $G^+ = (S, A^+)$ où

$$\forall x, y \in S, x \smile y \iff (x \to y \vee y \to x)$$

On a également A symétrique $\iff$ $G^+ = G^-$. Cela correspond à un graphe où chaque arête est dans les deux sens $x \leftrightarrow y$ et, donc, on peut oublier l'orientation.

De la même manière, on peut associer à un graphe non orienté G son graphe orienté symétrique obtenu en doublant chaque arête. C'est à dire en posant si $x \to y$ si $x \smile y$. On a donc $\forall x, y \in S, x \to y \iff y \to x$ et le graphe est bien symétrique.

On reprend directement l'essentiel du vocabulaire des graphes orientés symétriques avec des simplifications :

Définition II.2 Soit $G = (S, A)$ un graphe et $x \in S$.

- On appelle *voisins* de x les éléments de

$$v(x) = \{\ y \in S \mid x \smile y\ \}$$

- On appelle *degré* de x l'entier $d(x) = |v(x)|$.

Théorème II.1 $\sum_{x \in S} d(x) = 2|A|$

On étend directement la notion de **chemin** mais il faut faire attention au fait que **simple** n'a pas le même sens entre un graphe non orienté symétrique et un graphe non orienté. En effet, on $x \to y \to x$ est simple pour un graphe orienté alors que $x \smile y \smile x$ ne l'est pas vu qu'il s'agit de la même arête.

II.2 Connexité

Définition II.3 On définit $\smile^*$ comme étant la fermeture réflexive et transitive de $\smile$.

Théorème II.2

- $\smile^*$ est une relation d'équivalence dont les classes sont appelées les **composantes connexes** du graphe.
- $x \smile^* y \iff$ il existe un chemin de x à y.

Définition II.4 Un graphe n'ayant qu'une classe d'équivalence pour $\smile^*$ est dit **connexe**. Cela signifie qu'il existe un chemin entre toute paire de sommets.

II.3 Graphe acyclique connexe

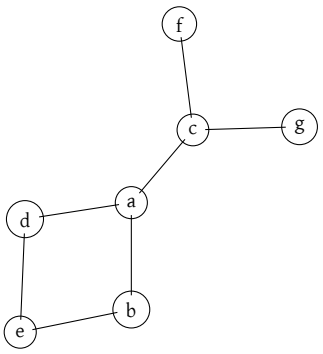
Définition II.5 On appelle **arbre** un graphe acyclique connexe. Quand on a distingué un sommet, on parle d'**arbre enraciné**.

II.4 Graphe biparti

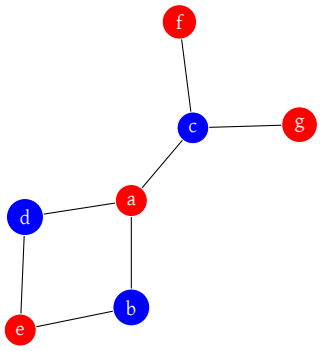
Définition II.6 Soit $G = (S, A)$ un graphe, on dit que le graphe est **biparti** lorsqu'il existe une partition $S = S_1 \cup S_2$ et que

$$\forall \{x, y\} \in A, x \in S_1 \iff y \in S_2$$

Ici, le graphe est biparti avec $S_1 = \{a, e, f, g\}$ et $S_2 = \{b, c, d\}$.



On peut s'en rendre compte en colorant les sommets.



En fait, être biparti est équivalent à pouvoir être coloré en deux couleurs en sorte que deux sommets reliés soient de couleur différente.

III Graphes classiques

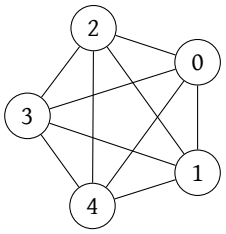
On présente ici brièvement des graphes ou des familles de graphes classiques qui serviront, notamment pour les exemples.

III.1 Graphes complets

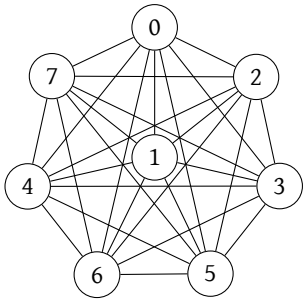
Définition III.1 Soit $n \in \mathbb{N}^*$, on appelle **graphe complet** à n sommets le graphe non orienté K_n de sommets $[\![0, n-1]\!]$ et tel que

$$\forall x, y \in [\![0, n-1]\!], x \neq y \iff x \smile y$$

K_5 :

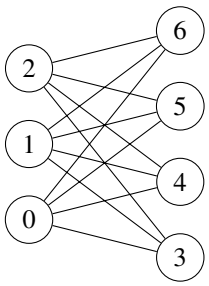


K_8 :



Définition III.2 Soit $n, p \in \mathbb{N}^*$, on appelle **graphe bipartite complet** à (n, p) sommets le graphe non orienté $K_{n,p}$ de sommets $[\![0, n+p-1]\!]$ et tel que

$$\forall x \in [\![0, n-1]\!], \forall y \in [\![n, n+p-1]\!], x \smile y$$



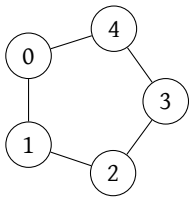
$K_{3,4}$:

III.2 Cycles

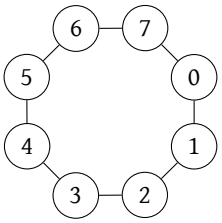
Définition III.3 Soit $n \in \mathbb{N}^*$, on appelle **cycle** de longueur n le graphe non orienté C_n de sommets $[\![0, n-1]\!]$ et tel que

$$\forall x, y \in [\![0, n-1]\!], |x-y| = 1 \iff x \smile y$$

C_5 :



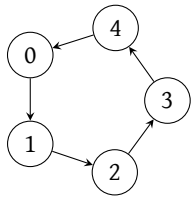
C_8 :



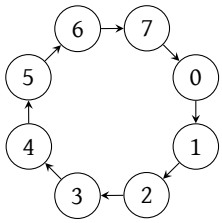
Définition III.4 Soit $n \in \mathbb{N}^*$, on appelle **cycle orienté** de longueur n le graphe orienté C_n^o de sommets $[\![0, n-1]\!]$ et tel que

$$\forall x, y \in [\![0, n-1]\!], y \equiv x+1\ [n] \iff x \to y$$

C_5^o :



C_8^o :



III.3 Grilles et hypercubes

On va manipuler ici des vecteurs de dimension d à coordonnées entières dans $[\![0, n-1]\!]$.
On sait que $[\![0, n-1]\!]^d$ est en bijection avec $[\![0, n^d-1]\!]$ et on considère la bijection explicite suivante :

$$\varphi(x) = \sum_{i=0}^{d-1} x_i \text{ où } x = (x_0, \dots, x_{d-1})$$

Soit $x = (x_0, \dots, x_{d-1})$ et $y = (y_0, \dots, y_{d-1}) \in [\![0, n-1]\!]^d$, on note $d_1(x, y) = \sum_{i=0}^{d-1} |x_i - y_i|$ qui compte la somme des différences de coordonnées. Ainsi $d_1((1,2,3),(3,2,2)) = |1-3| + |2-2| + |3-2| = 3$. Comme les coordonnées sont entières, on a le théorème suivant qui indique que les vecteurs à distance 1 sont ceux qui différent exactement de 1 dans une unique coordonnée :

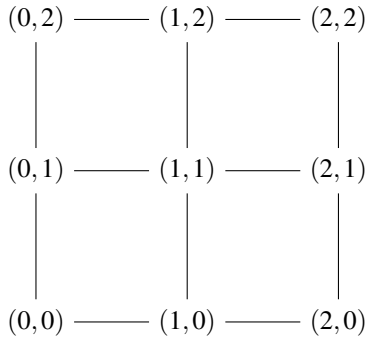
Théorème III.1 Soient $x, y \in [\![0, n-1]\!]^d$, on a

$$d_1(x,y) = 1 \iff \exists i \in [\![0,d-1]\!], \begin{cases} \forall j \in [\![0,d-1]\!], j \neq i \Rightarrow x_j = y_j \\ |x_j - y_j| = 1 \end{cases}$$

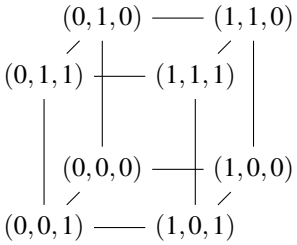
On en déduit un graphe de grille des vecteurs à distance 1 :

Définition III.5 Soit $n, d \in \mathbb{N}^*$, on appelle **grille de dimension d et de taille n** le graphe G_n^d dont les sommets sont dans $[\![0, n-1]\!]^d$ et où

$$\forall x, y \in [\![0,n-1]\!]^d, x \sim y \iff d_1(x,y) = 1$$



G_3^2 :



G_2^3 :

Définition III.6 Soit $n \in \mathbb{N}^*$, on appelle **hypercube de dimension n** le graphe $H_n = G_2^n$.

On vient de voir l'hypercube de dimension 3 dans l'exemple précédent.

Remarque L'hypercube de dimension n est intéressant car il permet de représenter chaque nombre de $[\![0, 2^n - 1]\!]$ comme un sommet du graphe. Par exemple, toute chaîne hamiltonienne, c'est-à-dire qui contient chaque sommet du graphe, correspond à un ordre d'énumération des nombres qui vérifie qu'un nombre et son successeur ne diffèrent que sur un bit.

IV Parcours

ITC- Sup | MP2I- S2

IV.1 Principe

La grande majorité des algorithmes sur les graphes consistent à parcourir les sommets de voisins en voisins pour effectuer des traitements. La manière dont on les parcourt pouvant changer selon les différentes applications.

Citons, par exemple, le fait de déterminer les composantes connexes d'un graphe ou de trouver le plus court chemin entre deux sommets.

On va se placer dans le cadre d'un graphe orienté représenté par listes d'adjacence et en supposant que les sommets sont identifiés par leur indice. Dans ce cadre, si on a, en fait, un graphe non orienté, il sera représenté par son graphe orienté symétrique comme on l'a vu plus haut.

On va également considérer qu'on veut effectuer un traitement, ou une visite, pour ces sommets ou les arêtes empruntées.

IV.2 Parcours en profondeur récursif

On présente ici une première manière élémentaire de les parcourir en tirant partie de la récursivité :

- on considère une fonction `parcours` et l'appel à `parcours` pour le sommet `x` va effectuer des appels récursifs à `parcours` pour chaque sommet y de $v_+(x)$.

Le problème est qu'on ne veut pas traiter deux fois un sommet et on veut que les appels terminent. Pour cela, on introduit une notion d'état associé à chaque sommet. Un sommet peut-être

- **Inconnu**, cela correspond au fait qu'il n'est pas encore apparu en tant que voisin.
- **Découvert**, il est apparu mais n'a pas encore été traité complètement.
- **Traité** (ou **Visité**), il a non seulement été traité, mais également tous les sommets parcouru grâce à lui.

Pour maintenir cet état dans le code, le plus simple est de considérer un tableau d'entiers `etat` où `etat[i]` donne l'état du sommet `i`.

Définition IV.1 Ce parcours est appelé un **parcours en profondeur** récursif. En anglais, on parle de **depth-first search** et on utilise couramment l'acronyme **DFS**.

IV.2.i Première version

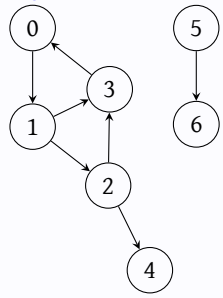
On adapte directement le principe précédent en un programme.

```python
INCONNU = 0
DECOUVERT = 1
TRAITE = 2

def parcours(ladj, x, etat):
  if etat[x] != TRAITE:
    print(x)
    for y in ladj[x]:
      if etat[y] == INCONNU:
        etat[y] = DECOUVERT
        parcours(ladj, y, etat)
    etat[x] = TRAITE

def lance_parcours(ladj, x):
  # État initial inconnu pour tous
  etat = [ INCONNU ] * len(ladj)
  etat[x] = DECOUVERT
  parcours(ladj, x, etat)
```

Python

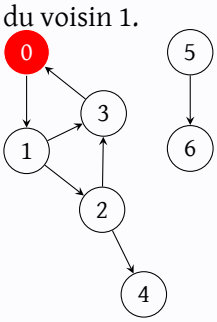
Ici, on se contente d'afficher les sommets rencontrés.

Exemple On va réaliser un parcours en profondeur du graphe suivant en partant du sommet 0 :

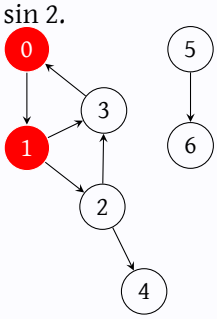


On indique les sommets inconnus en blanc, les sommets découvert en rouge et les sommets traités en vert.

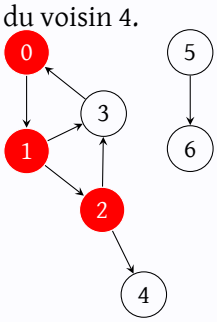
On affiche 0 et on appelle relance le parcours du voisin 1.



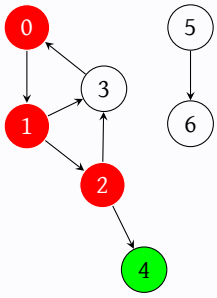
On affiche 1 et on relance le parcours du voisin 2.



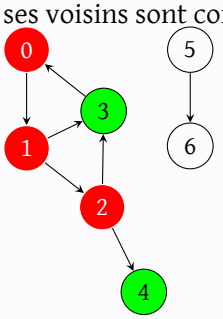
On affiche 3 et on appelle relance le parcours du voisin 4.



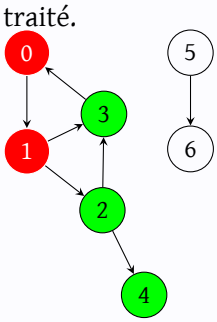
On affiche 4 et comme il n'a pas de voisins non traités, on traite 4 et on sort de l'appel.



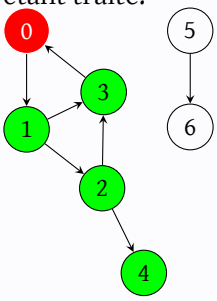
On revient à 2 et on relance le parcours du voisin 3. On affiche 3, et on le traite car tous ses voisins sont connus.



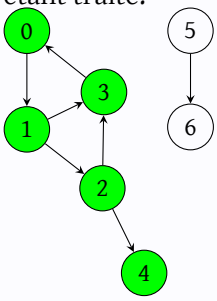
On revient à 2 et cette fois-ci il n'y a plus de voisins, on peut marquer 2 comme étant traité.



On revient à 1 et on peut marquer 1 comme étant traité.



On revient à 0 et on peut marquer 0 comme étant traité.



On a donc une vague de descente dans le graphe qui marque les sommets comme découverts et ensuite on remonte la pile d'appels récursifs en marquant les sommets comme étant traités.

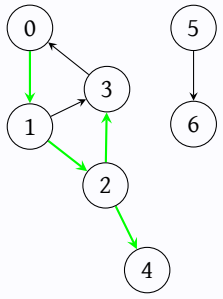
Les sommets sont ici affichés dans l'ordre 0 1 2 4 3. On remarque que les sommets 5 et 6 sont inaccessibles, ils restent ainsi inconnus tout le long du parcours.

IV.2.ii Arbre de parcours

On peut noter les arêtes empruntées lors du parcours précédents pour découvrir un nouveau sommet, et donc relancer un appel récursif. On obtient alors une structure arborescente qui est appelé l'arbre du parcours en pro-

fondeur.

Exemple Dans le parcours précédent, on obtient l'arbre suivant :



Pour obtenir cet arbre, on va construire un tableau `parent` pendant le parcours. Lorsqu'on emprunte une arête $x \to y$ pour découvrir y, on note $parent[y] = x$. Par défaut, $parent[y]$ est indéfini (`None` ou une valeur d'indice invalide comme -1).

Théorème IV.1 Si $parent[y]$ est défini à l'issue du parcours depuis x, alors il existe un chemin $\varphi : x \rightsquigarrow y$ où les arêtes $s \to s'$ empruntées lors du chemin vérifient toutes $parent[s'] = s$.

Comme $parent[y]$ est défini pour tous les sommets découverts et que les sommets découverts finissent tous par être traités, on peut en déduire directement le théorème suivant :

Théorème IV.2 A l'issue du parcours depuis x, les sommets traités sont les sommets accessibles depuis x, c'est-à-dire les éléments de

$$\{\, y \in S \mid x \rightsquigarrow y \,\}$$

On présente le calcul de ces chemins dans le programme suivant :

```python
INCONNU = 0
DECOUVERT = 1
TRAITE = 2

def parcours(ladj, x, etat, parent):
  if etat[x] != TRAITE:
    print(x)
    for y in ladj[x]:
      if etat[y] == INCONNU:
        parent[y] = x
        etat[y] = DECOUVERT
        parcours(ladj, y, etat, parent)
    etat[x] = TRAITE

def arbre_parcours(ladj, x):
  etat = [ INCONNU ] * len(ladj)
  parent = [ None ] * len(ladj)
  etat[x] = DECOUVERT
  parcours(ladj, x, etat, parent)
  return parent

def chemin(parent, y):
  # Renvoie le chemin x -> ... -> y
  # en sens inverse
  p = [ y ]
  while parent[y] != None:
    y = parent[y]
    p.append(y)
  return p
```

Python

IV.2.iii Composantes connexes

On peut déduire directement du théorème précédent, le corollaire suivant dans le cas non orienté :

Théorème IV.3 Dans un graphe non orienté, les sommets traités depuis un parcours en profondeur issu d'un sommet sont exactement les sommets de sa composante connexe.

On en déduit alors un algorithme pour obtenir les composantes connexes d'un graphe non orienté :

Algorithme - COMPOSANTES CONNEXES
- Entrées :
  Un graphe non orienté $G = (S, A)$
- ⋆ Tant qu'il y a des sommets non traités
  - On choisit un sommet non traité x
  - On effectue un parcours récursif depuis x
  - Tous les sommets traités par ce parcours forment une composante connexe

On va en déduire le programme suivant :

```python
INCONNU = 0
DECOUVERT = 1
TRAITE = 2

def parcours(ladj, x, etat, composante):
  if etat[x] != TRAITE:
    composante.append(x)
    for y in ladj[x]:
      if etat[y] == INCONNU:
        etat[y] = DECOUVERT
        parcours(ladj, y, etat, composante)
    etat[x] = TRAITE

def composantes_connexes(ladj):
  composantes = []
  etat = [ INCONNU ] * len(ladj)
  for i in range(len(ladj)):
    if etat[i] == INCONNU:
      composante = [ ]
      parcours(ladj, x, etat, composante)
      composantes.append( composante )
  return composantes
```
Python

IV.2.iv Détection de cycles

Si on est en train de traiter le sommet x et qu'on rencontre une arête $x \to y$ où y est découvert mais non traité, c'est qu'on est dans l'appel récursif de y et donc que x est un de ses descendants : $y \rightsquigarrow x$ en rajoutant la nouvelle arête $y \rightsquigarrow x \to x$ on en déduit un cycle dans le cas d'un graphe orienté.

Pour un graphe non orienté représenté par un graphe orienté symétrique, il faut faire attention à ne pas prendre un aller-retour $x \to y \to x$ pour un cyle. On demande donc à avoir la condition :

```python
etat[y] == DECOUVERT and parent[x] != y
```
Python

On en déduit le programme suivant :

```python
INCONNU = 0
DECOUVERT = 1
TRAITE = 2

def parcours(ladj, x, etat, parent, est_oriente):
  if etat[x] != TRAITE:
```

```python
    for y in ladj[x]:
      if etat[y] == INCONNU:
        parent[y] = x
        etat[y] = DECOUVERT
        cycle = parcours(ladj, y, etat, parent, est_oriente)
        if cycle != None:
          return cycle
      elif etat[y] == TRAITE and (est_oriente or parent[x] != y):
        return chemin(parent, y)

    etat[x] = TRAITE
  return None

def detecte_cycle(ladj, x, est_oriente):
  etat = [ INCONNU ] * len(ladj)
  parent = [ None ] * len(ladj)
  etat[x] = DECOUVERT
  cycle = parcours(ladj, x, etat, parent, est_oriente)
  if cycle != None:
    print('Cyclique : ', cycle)
  else:
    print('Acyclique')
```

Python

IV.2.v Classification des arêtes

MP2I- S2 | OI- Spé

IV.2.vi Temps d'entrée et de sortie

MP2I- S2 | OI- Spé

IV.3 Parcours quelconque

On va considérer ici que l'on dispose d'une structure de donnée `sac` dans laquelle on peut placer des éléments et en sortir. On considère aussi qu'on a une moyen de marquer les sommets. Par défaut, ils sont non marqués.

On considère alors l'algorithme suivant decrit en pseudo-code et auquel on fera référence comme étant le parcours **quelconque** dans la suite :

Algorithme - PARCOURS QUELCONQUE
- Entrées :
  - ⋆ Un graphe $G = (S, A)$
  - ⋆ Un somme $s \in S$ source du parcours
- ⋆ On crée un sac ne contenant que s
- ⋆ Tant que le sac est non vide
  - ○ On tire un sommet x du sac
  - ○ Si x n'est pas marqué
    - ☐ On marque x
    - ☐ pour chaque $x \to y \in A$
      - On ajoute y dans le sac

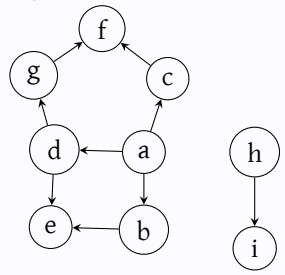
On remarque que les somets marqués sont exactement les sommets accessibles depuis s, on retrouve ainsi le théorème vu pour le parcours récursif.

En fait, le parcours quelconque ne permet pas de déduire plus d'information que cela, mais dans un graphe non orienté c'est déjà suffisant pour en déduire les composantes connexes.

La question qui se pose alors est celle de la stratégie d'ajout/tirage dans le sac. Selon ce que l'on considère, l'ordre de visite des sommets va changer. On va considérer trois stratégies :
- LIFO : Last In First Out, le prochain sommet tiré est le dernier ajouté
- FIFO : First In First Out, le prochain sommet tiré est le sommet le plus anciennement ajouté
- aléatoire : on tire aléatoirement et uniformément un sommet du sac

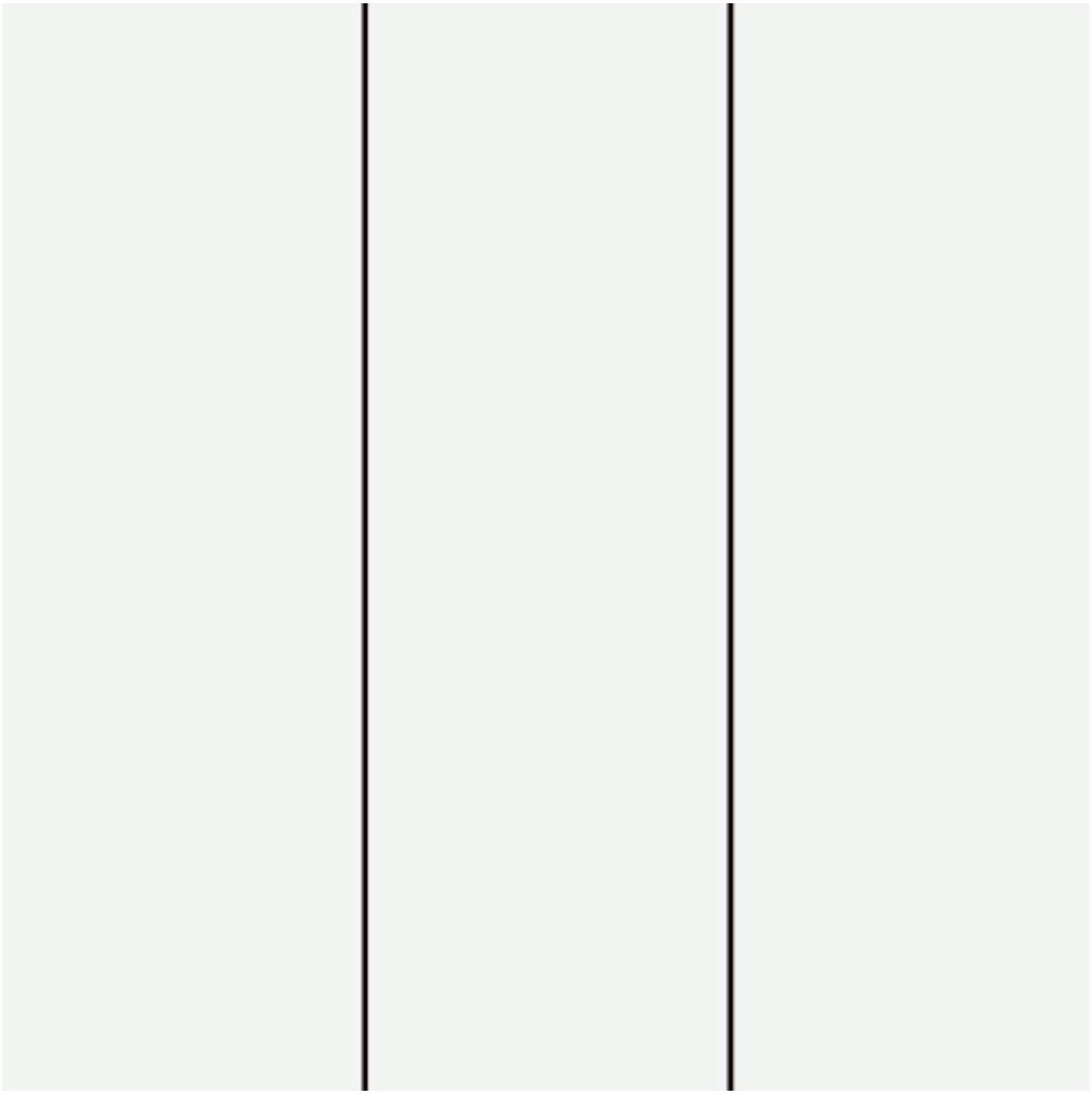
Définition IV.2 Un sac avec une stratégie LIFO est appelé une **pile**.
Un sac avec une stratégie FIFO est appelé une **file**.

Exemple Si on considère le graphe suivant :



En effectuant un parcours quelconque issu de a :
- en stratégie LIFO, on va voir les sommets dans l'ordre a, b, e, c, f, d, g.
- en stratégie FIFO, on va voir les sommets dans l'ordre a, b, c, d, e, f, g
- en stratégie aléatoire, on pourrait avoir a, b, c, e, d, f, g

Une manière de *voir* ces trois différentes stratégies est de considérer une grille et de colorer les sommets marqués avec une couleur changeant à chaque marquage. On peut voir dans l'ordre LIFO, FIFO, aléatoire l'effet de ces trois stratégies en partant du même point sur l'animation suivante :



Remarque La stratégie LIFO semble *bornée* : elle choisit un cap et elle s'y tient tant que c'est possible. La stratégie FIFO va elle s'étendre comme une onde partant du point de départ. De manière assez étonnante, la stratégie aléatoire semble être une bonne approximation de la stratégie FIFO, ce qui montre le caractère exceptionnel du LIFO.

Le parcours quelconque en stratégie FIFO s'appelle un *parcours en largeur*. Comme on va le voir, le parcours avec stratégie LIFO est exactement le *parcours en profondeur* précédent.

IV.3.i Implémentation en Python

ITC- Sup

On va supposer qu'on dispose de quatre fonctions manipulant des sacs :

- `sac_vide()` qui renvoie un nouveau sac vide
- `ajoute(sac, x)` qui ajoute `x` dans le sac `sac`
- `retire(sac)` qui retire un élément du sac `sac` et le renvoie
- `est_vide(sac)` qui renvoie un booléen indiquant si le `sac` est vide

Le programme du parcours quelconque s'écrit alors

```python
def parcours_sac(ladj, source):
  a_traiter = sac_vide()
  ajoute(a_traiter, source)
  marque = [ False ] * len(ladj)

  while not est_vide(a_traiter):
    x = retire(a_traiter)
    if not marque[x]:
      marque[x] = True
      for y in ladj[x]:
        ajoute(a_traiter, y)
```
Python

Remarque Pour utiliser ce parcours, on peut rajouter des instructions au moment où on marque un sommet ou au moment où on traite une arête.

Pour implémenter un sac, on va d'abord utiliser le type `list` de `Python` :

- Pour créer un sac vide, on utilise la valeur `[]` :

```python
def sac_vide():
  return []
```
Python

- Pour tester si le sac est vide, on utilise donc le test à `[]` :

```python
def est_vide(sac):
  return sac == []
```
Python

- Pour ajouter un élément `x` dans le sac `s` dans tous les cas :

```python
def ajoute(sac, x):
  sac.append(x)
```
Python

- (FIFO) Pour retirer le dernier élément ajouté, on peut utiliser directement `pop` :

```python
def retire(sac):
  return sac.pop()
```
Python

* (LIFO) Pour retirer l'élément le plus anciennement ajouté, on peut utiliser la fonction de suppression `del l[0]` qui retire le premier élément

```python
def retire(sac):
  x = sac[0]
  del sac[0]
  return x
```
Python

- C'est anecdotique, mais pour retirer un élément au hasard, il faut choisir un indice `i`, renvoyer l'élément qui s'y trouve en le supprimant du sac avec `del s[i]` :

```python
from random import randint
def retire(sac):
  i = randint(len(sac))
  x = sac[i]
  del sac[i]
  return x
```
Python

Si on peut supposer que les quatre premières opérations sont en temps constant, les deux dernières sont linéaire en le nombre de sommets dans le sac. Ce qui est assez coûteux.

En effet, le parcours quelconque va accéder en pire cas à chaque sommet et à chaque liste d'adjacence et ajouter chaque sommet dans le sac. Donc, une complexité en $O(|S| + |A| + |S|f(|S|))$ où $f(n)$ est la complexité du retrait dans un sac de taille n.

On obtient donc $O(|S| + |A|)$ en LIFO mais $O(|S|^2)$ en FIFO car $|A| = O(|S|^2)$ vu que $A \subset S^2$.

En Python, il est possible d'avoir une file (queue en anglais) permettant de réaliser le retrait FIFO en $O(1)$:
- On importe le type deque : `from collections import deque`
- On crée une file vide avec `s = deque()`
- On ajoute un élément x avec `s.append(x)`
- Mais on dispose d'un fonction efficace `s.popleft()` pour retirer le premier élément.

En fait, on dispose également d'un `appendleft` et du `pop`. Toutes ces opérations étant en $O(1)$.
Dans l'implémentation précédente, il suffit donc de remplacer les deux fonctions suivantes :

```python
def sac_vide():
  return deque()

def est_vide(sac):
  return len(sac) == 0

def ajoute(sac, x):
  sac.append(x)

def retire(sac):
  return sac.popleft()
```
Python

Remarque OI- Sup

La structure de donnée deque est une liste doublement chaînée. Elle permet de remonter dans la chaîne d'un maillon vers le maillon dont il est le suivant.

En pratique, on pourra définir des fonctions de parcours utilisant directement les bonnes structures :

```python
def parcours_profondeur(ladj, source):
  a_traiter = [ source ]
  marque = [ False ] * len(ladj)

  while a_traiter != []
    x = a_traiter.pop()
    if not marque[x]:
      marque[x] = True
      for y in ladj[x]:
        a_traiter.append(y)

# à placer en début du programme :
from collections import deque

def parcours_largeur(ladj, source):
  a_traiter = deque([ source ])
  marque = [ False ] * len(ladj)

  while len(a_traiter) != 0:
    x = a_traiter.popleft()
    if not marque[x]:
      marque[x] = True
      for y in ladj[x]:
        a_traiter.append(y)
```
Python

IV.4 Parcours en largeur

On va vu que le parcours quelconque en FIFO s'appelait le parcours en largeur. Il permet de parcourir les sommets d'un graphe en rayonnant à partir du sommet source. En effet on traite d'abord :
- le sommet source src
- les voisins de src
- les voisins des voisins de src
- ...

Pour pouvoir parler de chemins, il est nécessaire d'introduire une notion de parenté. Pour cela, on a deux pos-

sibilités.

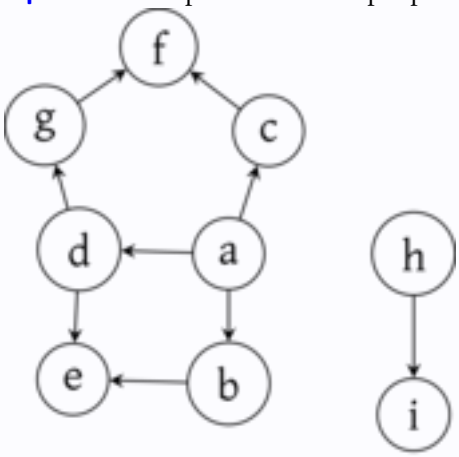
- Soit on n'ajoute plus des sommets dans le sac mais des couples (sommet, parent) qui correspondent a une arête de découverte :

Algorithme - PARCOURS QUELCONQUE AVEC COUPLES
- Entrées :
  - ⋆ Un graphe $G = (S, A)$
  - ⋆ Un somme $s \in S$ source du parcours
- ⋆ On crée un sac ne contenant que (s, s)
- ⋆ Tant que le sac est non vide
  - ○ On tire un couple (x, p) du sac
    - □ Si x est non marqué
      - On indique que $parent[x] = p$
      - pour chaque $x \to y \in A$
        - ⋆ On ajoute (y, x) dans le sac

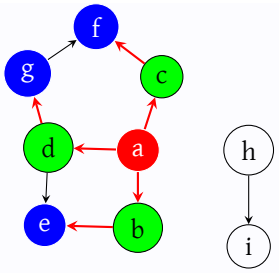
- Soit on modifie le parcours *quelconque* pour en déduire un parcours quelconque qui a l'air a priori d'être *optimisé*.

Algorithme - PARCOURS QUELCONQUE *OPTIMISÉ*
- Entrées :
  - ⋆ Un graphe $G = (S, A)$
  - ⋆ Un somme $s \in S$ source du parcours
- ⋆ On crée un sac ne contenant que s
- ⋆ Tant que le sac est non vide
  - ○ On tire un sommet x du sac
    - □ pour chaque $x \to y \in A$
      - Si y est non marqué
        - ⋆ On indique que $parent[y] = x$
        - ⋆ On ajoute y dans le sac
        - ⋆ On marque y

Exemple On va reprendre l'exemple précédent



On obtient avec un parcours en largeur le schéma suivant :



où on a noté les arêtes de parenté ainsi que d'une même couleur les éléments à égale distance de a dans le parcours en largeur.

On remarque qu'on a obtenu le chemin $a \to c \to f$ là où un parcours en profondeur aurait pu donner le chemin plus long $a \to d \to g \to f$.

Comme semble le suggérer l'exemple précédent, on peut démontrer le théorème suivant :

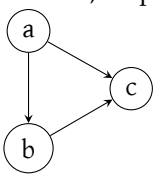
Théorème IV.4 Les chemins donnés par un parcours en largeur depuis x sont de plus petite longueur :
Si $\varphi : x \rightsquigarrow y$ est donné par le parcours, alors $\forall \psi : x \rightsquigarrow y, |\psi| \geq |\varphi|$.

IV.4.i Preuve

OI- Spé | MP2I- S2

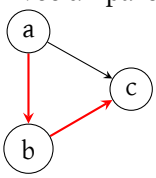
IV.5 Pseudo-parcours en profondeur

On peut se poser la question de la nature du parcours quelconque optimisé effectué avec une stratégie LIFO. Naïvement, on peut penser qu'il s'agit d'un parcours en profondeur. Cependant, si on considère le graphe :

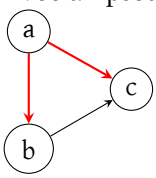


On va avoir les deux arbres différents suivants :

Avec un parcours en profondeur :



Avec un pseudo-parcours en profondeur :



On perd une propriété fondamentale du parcours en profondeur qui est de qu'un parent ne peut empecher un de ses enfants de découvrir un sommet. Ici, dans le pseudo-parours, a bloque la découverte de c par b.

Exercice 1 Montrer qu'il n'existe aucune stratégie permettant au parcours quelconque *optimisé* d'être un parcours en profondeur.

V Tri topologique

Un cas très important de graphes orientés est celui de graphes de dépendances :

- les sommets sont des tâches à réaliser
- Une arête $x \to y$ indique la tâche x doit être réalisée avant la tâche y.

Citons, par exemple, les dépendances entre chapitres dans un cours, entre unités de programme lors d'une compilation, entre produits dans un procédé industriel...

Ces graphes ont tous la particularité d'être orienté et acycliques. En effet, si on a un cycle, cela signifie qu'il y a une dépendance inextricable.

Remarque On parle en anglais de **DAG** pour *directed acyclic graph*.

Définition V.1 Soit $G = (S, A)$ un graphe orienté acyclique. Une énumération $(x_1, \dots, x_n)$ des sommets de S est appelée un **tri topologique** lorsque pour toute arête $x_i \to x_j$ on a $i < j$.

Autrement dit, un tri topologique est un ordre linéaire de traitement des tâches respectant les dépendances.

Théorème V.1 Soit G un graphe orienté acyclique et $x_1, \dots, x_p$ les sommets de degré entrant nuls de G.
Si on effectue plusieurs parcours en profondeur récursifs depuis chacun des x_i avec la même horloge, alors les sommets triés dans l'ordre décroissant des temps de sortie constituent un un tri topologique de G.

■ **Preuve**
Il suffit de vérifier que si $x \to y$ n'est pas une arête arrière, alors $t_s(x) \geq t_s(y)$ où $t_s(x)$ est le temps de sortie de x.

■

VI Plus courts chemins

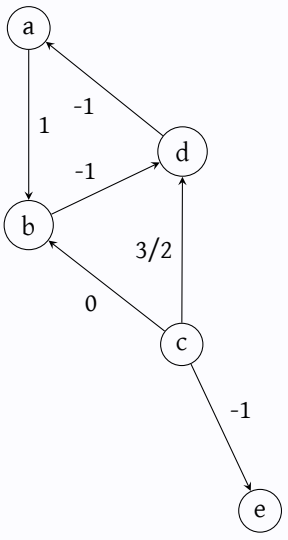
VI.1 Graphes pondérés et définition du problème

Définition VI.1 Un graphe pondéré est la donnée d'un graphe, orienté ou non, $G = (S, A)$ et d'une fonction de pondération $\pi : A \to \mathbb{R}$ qui associe à chaque arête un nombre réel, son poids.

Remarque En informatique pratique, $\mathbb{R}$ n'existe pas vraiment. Les poids seront donc le plus souvent des entiers ou des flottants. Cependant, les problèmes possiblement rencontrés par des poids réels sont pertinents et le cadre unifié rend les définitions plus simples.

Définition VI.2 On définit le poids d'un chemin φ comme la somme des poids de ses arêtes.

Exemple Dans le graphe suivant :



le chemin $\varphi = c \to b \to d \to a$ est de poids $\pi(\varphi) = 0 - 1 - 1 = -2$.
On remarque que $b \to d \to a \to b$ est un cycle de poids -1, on peut donc l'emprunter autant de fois qu'on le souhaite sur un chemin passant par a, b ou d et faire diminuer son poids autant qu'on le souhaite. On parle de cycle de poids négatif et leur présence est problématique.

Théorème VI.1 Si G est un graphe pondéré **sans cycles de poids négatif**, φ est un chemin et ψ est un sous-chemin de φ obtenu en enlevant un cycle, alors $\pi(\varphi) \geq \pi(\psi)$.

Du théorème précédent, on en déduit que pour obtenir un chemin de poids minimum, on peut supposer que le chemin est élémentaire. Or, comme il existe un nombre fini de chemins élémentaires entre deux sommets, on peut en déduire la définition suivante :

Définition VI.3 Soit $G = (S, A, \pi)$ un graphe pondéré **sans cycles de poids négatifs** et $x, y \in S$ tels que $x \rightsquigarrow y$, on appelle plus court chemin de x à y, un chemin $\varphi : x \rightsquigarrow y$ tel que

$$\pi(\varphi) = \min \{ \pi(\psi) \mid \psi : x \rightsquigarrow y \}$$

Il est possible de travailler avec la complétion d'un graphe pondéré en considérant toutes les arêtes possibles mais en spécifiant que si $x \to y$ n'est pas une arête du graphe initial, alors $\pi(x \to y) = \infty$. On peut alors poser pour tout couple $(x, y) \in S$, $dist(x, y) = \min \{ \pi(\psi) \mid \psi : x \rightsquigarrow y \}$ sachant que cette distance vaut ∞ lorsque y n'est pas accessible depuis x.

On va s'intéresser à deux problèmes :

Problème - PLUSCOURTCHEMINSOURCE
- Entrées :
  - ⋆ Un graphe pondéré $G = (S, A, \pi)$
  - ⋆ Un sommet source $s \in S$
- Sortie :
  - ⋆ La donnée pour chaque $x \in S$, de la valeur $dist(s, x)$.
  - ⋆ Éventuellement, un chemin $\varphi : s \rightsquigarrow x$ réalisant cette distance.

Problème - PLUSCOURTCHEMINTOUTCOUPLE
- Entrées :
  - ⋆ Un graphe pondéré $G = (S, A, \pi)$
- Sortie :
  - ⋆ La donnée pour chaque couple $x, y \in S$, de la valeur $dist(x, y)$.
  - ⋆ Éventuellement, un chemin $\varphi : x \rightsquigarrow y$ réalisant cette distance.

Le premier problème est naturellement inclus dans le second.

VI.2 Cas des poids rationnels

MP2I- S2

Il se trouve qu'on a déjà trouvé un algorithme permettant de résoudre le problème quand $\forall e \in A, \pi(e) = 1$: le **parcours en largeur**.

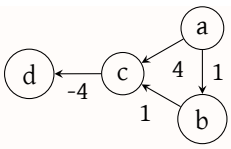
Si les poids sont dans $\mathbb{Q}_+^*$, on peut se ramener à des poids dans $\mathbb{N}^*$ en multipliant chaque poids par le ppcm des dénominateurs des poids. Cela ne change pas la relation d'ordre entre les poids de chemins, et on peut donc les plus courts chemins sont les mains.

Si $\pi(x \to y) = k > 1$, on peut rajouter $k - 1$ sommets artificiels $z_1, \ldots, z_{k-1}$ et des arêtes de poids unitaire $x \to z_1 \to z_2 \to \cdots \to z_k$ à la place de l'arête $x \to y$. En procédant ainsi, on se ramène donc à un graphe dont les poids sont unitaires et on peut résoudre le problème avec un parcours en largeur.

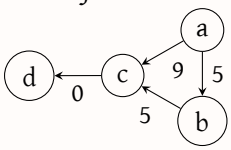
Si on considère que la longueur des poids en mémoire fait partie de l'entrée, alors cette réduction a l'air très coûteuse, mais si on considère qu'ils sont constants en espace, alors toute cette réduction ne fait que grossir la constante du $O(|S| + |A|)$!

Remarque On pourrait être tenté de supprimer les poids négatifs en ajoutant une constante à chaque poids, cependant, en faisant cela, on pénalise les chemins en fonction de leur longueur et on ne préserve pas la notion de plus court chemin.

Considérons par exemple le graphe :



Le chemin le plus court entre a et c est de poids 2 et est le chemin $a \to b \to c$.
Si on ajoute 5 à chaque poids pour qu'ils soient > 0, on obtient le graphe :

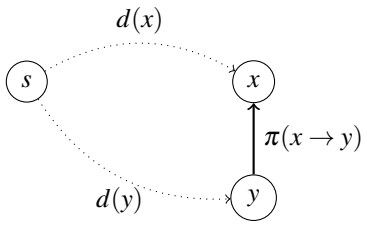


Et ici, le chemin le plus court devient $a \to c$ de poids 9 et non plus $a \to b \to c$ qui devient de poids 10.
Plus généralement, si on ajoute η à chaque poids, alors le poids de φ devient $\pi(\varphi) + \eta|\varphi|$ qui n'est pas croissante sur les chemins ordonnés par poids.

VI.3 Algorithme de Dijkstra

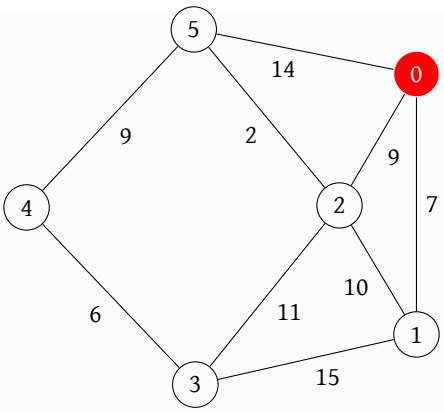
ITC- Sup | MP2I- S2

On se concentre ici sur le problème `PlusCourtCheminSource` dans le cas de poids positifs.
Si $G = (S, A, \pi)$ est un graphe pondéré et $s \in S$, on considère un étiquettage $d : S \to \mathbb{R} \cup \{\infty\}$ qu'on va faire évoluer dans un algorithme et tel que $d(x)$ représente le plus petit poids trouvé jusqu'ici d'un chemin entre s et x. En l'absence de cycle négatif, on sait qu'un tel chemin peut être supposé élémentaire.
Au départ, on pose $d(s) = 0$ et $\forall x \in S \backslash \{s\}, d(x) = \infty$.
L'idée est d'emprunter les raccourcis, c'est-à-dire les arêtes $y \to x$ telles que $d(y) + \pi(y \to x) < d(x)$:

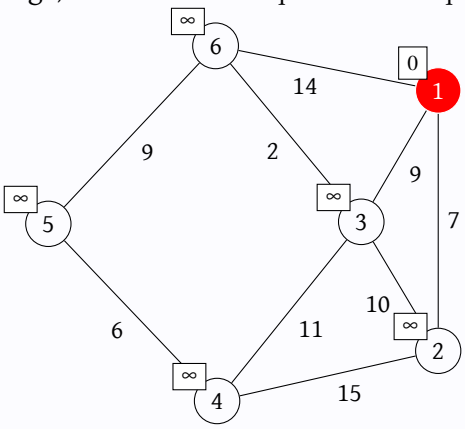


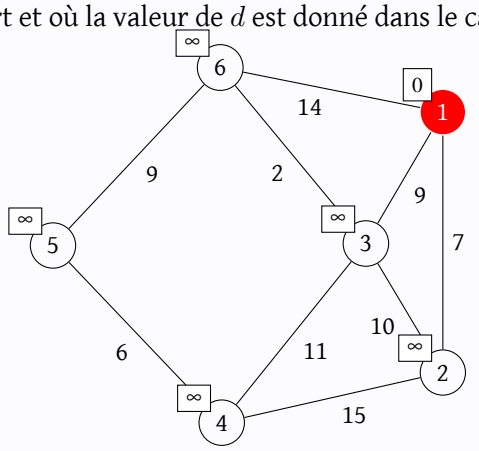
Algorithme - DIJKSTRA
- Entrées :
  - ⋆ Un graphe pondéré $G = (S, A, \pi)$ où $\pi : A \to \mathbb{R}_+$
  - ⋆ Un sommet $s \in S$
- ⋆ On crée un tableau `d` avec $d(s) = 0$ et $d(x) = \infty$ pour $x \neq s$.
  - ⋆ On crée un tableau `a_traiter` et on ajoute `s`
  - ⋆ Les sommets sont initialement non marqués
  - ⋆ Tant que `a_traiter` est non vide
    - ○ On retire le sommet `x` de `a_traiter` de valeur $d(x)$ **minimale**
    - ○ Si `x` est non marqué
      - ☐ On marque `x`
      - ☐ Pour chaque arête $x \to y$ où $d(x) + \pi(x \to y) < d(y)$
        - On pose $d(y) = d(x) + \pi(x \to y)$
        - On ajoute y dans `a_traiter`

Exemple On considère le graphe suivant

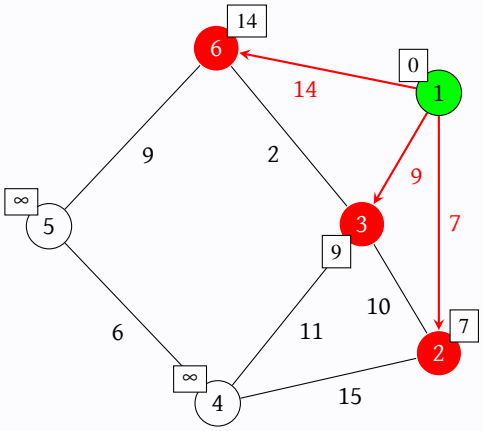


pour $s = 1$ où les sommets dans `a_traiter` sont indiqués en rouge, les sommets marqués sont indiqués en vert et où la valeur de d est donné dans le cadre à côté du sommet :

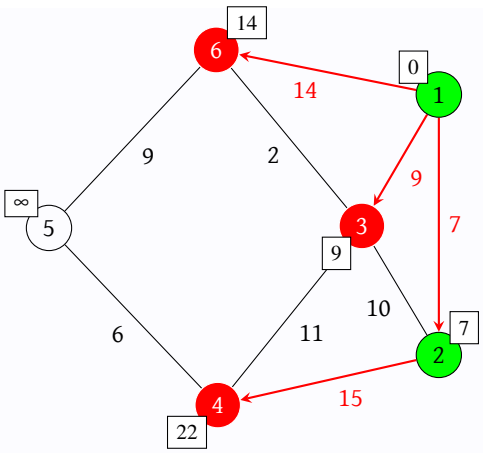




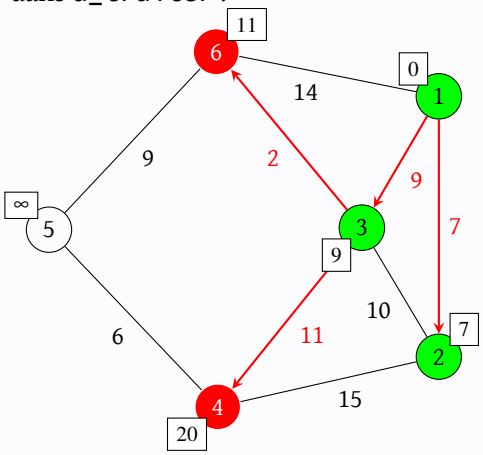
On commence par retirer 1 qui est seul dans `a_traiter` avec $d(1) = 0$. On peut alors ajouter les sommets $6, 3$ et 9 :



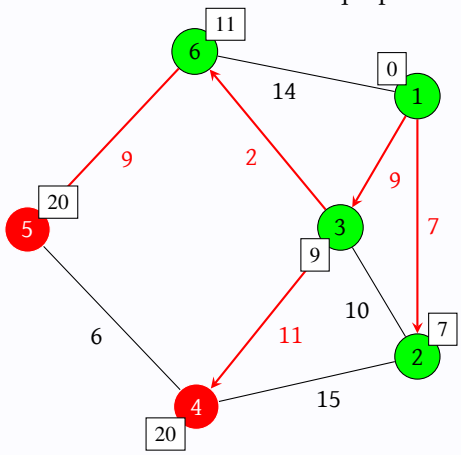
On retire alors le sommet 2 qui a la plus petite valeur de d. L'arête $2 \to 15$ vérifie la condition mais pas l'arête $2 \to 3$. On ajoute donc 4 dans les sommets à traiter :



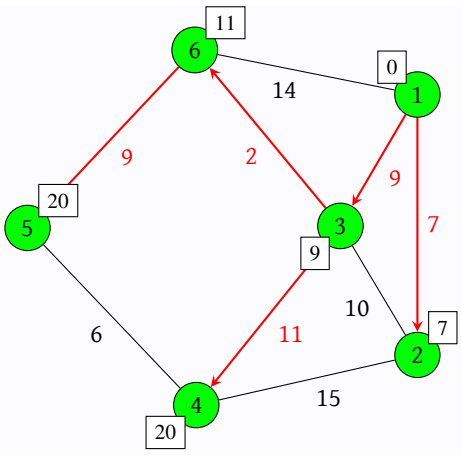
On retire alors 3 et on voit que les deux arêtes $3 \to 6$ et $3 \to 4$ permettent d'améliorer d. On rajoute donc 4 et 6 dans `a_traiter` :



On retire alors le sommet 6 qui permet d'emprunter $6 \to 5$ et qui rajoute ainsi 5 à `a_traiter` :



On traite les deux derniers sommets en les retirant.



Cet algorithme est un parcours quelconque où la stratégie de sortie du sac est de retirer le sommet de plus petite valeur de d. On sait donc déjà qu'il termine et qu'il explore $acc(s) = \{ x \in S \mid s \rightsquigarrow x \}$.

Comme on ne modifie d que pour mettre des nombres $\neq \infty$, on est assuré qu'à la fin de l'algorithme les sommets dans $acc(s)$ ont une valeur dans $\mathbb{R}_+$ dans le tableau d et que les autres ont ∞. La force de l'algorithme de Dijkstra découle du contenu final de d qui répond à la question voulue :

Théorème VI.2 A la fin de l'algorithme, si $s \rightsquigarrow x$, $d(x) = dist(s, x)$.

On peut implémenter directement cet algorithme :

```ocaml
let rec extrait_plus_petit l d =
  match l with
  | [] -> failwith "impossible"
  | [x] -> (x, [])
  | x::q ->
    let (y, q') = extrait_plus_petit q d in
    if d.(y) < d.(x)
    then (y, x :: q')
    else (x, q)

let dijkstra g src =
  let n = Array.length g in
  let inf = max_int in
  let d = Array.make n inf in
  d.(src) <- 0;
  let marques = Array.make n false in
  let a_traiter = ref [ src ] in
  while !a_traiter <> [] do
    let x, q = extrait_plus_petit !a_traiter d in
    a_traiter := q;
    if not marques.(x)
    then begin
      marques.(x) <- true;
      List.iter (fun (y, poids) ->
        let n_d = d.(x) + poids in
        if n_d < d.(y)
        then begin
          d.(y) <- n_d;
          a_traiter := y :: !a_traiter
        end
      ) g.(x)
    end
  done;
  d
```

OCaml

```
ERROR: src/structuresdonnees/../../snippets/str c ures/d
```

```python
def extrait_plus_petit(t, d):
  x = t[0] # recherche du d[x] min dans t
  for i in t[1:]:
    if d[i] < d[x]:
      x = i
  t.remove(x) # on le retire
  return x # on le renvoie

def dijkstra(g, src):
  n = len(g)
  inf = float('inf')
  d = [ inf ] * n
  d[src] = 0
  a_traiter = [ src ]
  marques = [ False ] * n
  while a_traiter != []:
    x = extrait_plus_petit(a_traiter, d)
    if not marques[x]:
      marques[x] = True
      for y, poids in g[x]:
        n_d = d[x] + poids
        if n_d < d[y]:
          d[y] = n_d
          a_traiter.append(y)
  return d # distandes finales
```

Python

Remarque La complexité de la recherche sommet de valeur d minimale est en pire cas en $O(|S|)$ comme on ajoute autant de sommets que d'arêtes en pire cas, on a une complexité totale en $O(|S| \times |V|)$.

VI.3.i Utilisation d'une file de priorité

MP2I- S2 OI- Spé

On peut améliorer l'algorithme précédent avec une **file de priorité** où la priorité de x est $d(x)$ et où on suppose qu'on extrait la priorité minimale. Cela permet ainsi d'obtenir une complexité en $O(|A| \log |S|)$.

Cependant, il est alors nécessaire de modifier la priorité ce qui est une opération assez pénible. A la place, on peut ajouter une autre fois le sommet. Au pire, on fera grossir la file (qui reste de longueur $O(|A|)$) et quand on retirera un sommet déjà traité, celui-ci sera ignoré.

On obtient ainsi l'algorithme suivant :

Algorithme - DIJKSTRA

- Entrées :
  - ⋆ Un graphe pondéré $G = (S, A, \pi)$ où $\pi : A \to \mathbb{R}_+$
  - ⋆ Un sommet $s \in S$
- ⋆ On crée un tableau `d` avec $d(s) = 0$ et $d(x) = \infty$ pour $x \neq s$.
  - ⋆ On crée une file de min-priorité `a_traiter` et on ajoute `s` avec priorité `0`
  - ⋆ Les sommets sont initialement non marqués
  - ⋆ Tant que `a_traiter` est non vide
    - ○ On retire le sommet `x` de `a_traiter`
    - ○ Si `x` est non marqué
      - ☐ On marque `x`
      - ☐ Pour chaque arête $x \to y$ où $d(x) + \pi(x \to y) < d(y)$
        - On pose $d(y) = d(x) + \pi(x \to y)$
        - On insère y avec priorité $d(y)$

VI.4 Relaxation

MP2I- S2

Pour montrer la correction de Dijkstra et le lien avec la présence d'arêtes positives, on va présenter ici un cadre plus général.

VI.4.i Arêtes tendues

Définition VI.4 On dit qu'une arête $x \to y$ est **tendue** si

$$d(y) > d(x) + \pi(x \to y)$$

Quand on pose ensuite $d(y) = d(x) + \pi(x \to y)$ on dit qu'on **relâche l'arête**.

Ainsi, une arête est tendue quand on peut l'emprunter pour améliorer l'estimation du plus petit chemin de s à y.

Théorème VI.3 Si aucune arête n'est tendue, alors $d(x) = dist(s, x)$ (avec $= \infty$ si inaccessible).

■ **Preuve**

Par l'absurde, considérons x tel que $d(x) > dist(s, x)$ où $dist(s, x)$ est minimale parmi les contre-exemples, alors $s \rightsquigarrow x$ et soit φ un chemin de longueur minimale.

Soit y le prédécesseur de x dans ce chemin, on a donc un chemin $\varphi = \psi y \to x_0$ où $\psi : s \rightsquigarrow y$. Nécessairement, le chemin ψ et minimal car restriction d'un chemin minimal. Donc, son poids est $\pi(\psi) = dist(s, y) = d(y)$ par hypothèse de minimalité de x.

On a alors $dist(s, x) = d(y) + \pi(y \to x) < d(x)$. L'arête $y \to x$ est donc tendue.

Contradiction.

■

On en deduit un pseudo-algorithme appelé *méthode de relaxation* :

Algorithme - RELAXATION
- Entrées :
  - ⋆ Un graphe pondéré $G = (S, A, \pi)$ sans cycle de poids négatif
  - ⋆ Un sommet $s \in S$
- ⋆ On pose d tel que $d(s) = 0$ et $\forall x \in S\backslash\{s\}, d(x) = \infty$.
  - ⋆ Tant qu'il existe une arête e tendue
    - ○ On relâche l'arête e

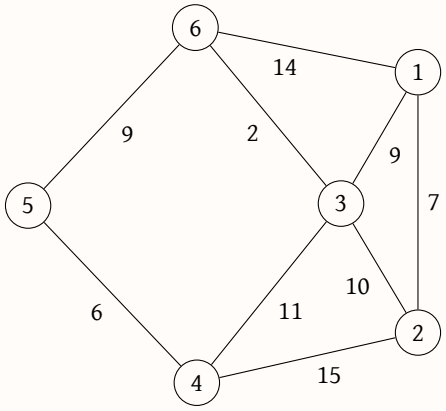
Remarque
- On montre par récurrence sur le nombre d'itérations que si $d(x) < \infty$ alors $d(x)$ est le poids d'un chemin simple de s à x. Comme ces chemins simples sont en nombre finis, leur poids aussi et donc d ne peut prendre qu'un nombre fini de valeurs, comme d change à chaque itération, la boucle termine.
- En présence d'un cycle négatif, on pourrait continuellement relacher les arêtes du cycle et donc l'algorithme ne termine pas.
- En sortie de l'algorithme on a donc $dist(s, x) \leq dist(s, y) + \pi(y \to x)$ pour toute arête $y \to x$. Or, on a l'égalité pour un plus court chemin. On en déduit donc l'égalité **très importante** suivante :

$$\forall x \in S\backslash\{s\}, dist(s, x) = \min\{\ dist(s, y) + \pi(y \to x) \mid y \in S\ \}$$

En fait, si on considère l'opération Φ définie par :

$$\forall x \in S\backslash\{s\}, \Phi(d)(x) = \min\{\ d(y) + \pi(y \to x) \mid y \in S\ \}$$

On remarque que $x \mapsto dist(s, x)$ est un point fixe de Φ.
On pourrait ainsi le calculer comme la limite de la suite stationnaire $d_{n+1} = \Phi(d_n)$ où $d_0(s) = 0$ et $\forall x \neq s, d_0(x) = \infty$.
Si on considère le graphe



et que l'on représente la suite d sous une forme de tableau on obtient :

n	$d_n(1)$	$d_n(2)$	$d_n(3)$	$d_n(4)$	$d_n(5)$	$d_n(6)$
0	0	∞	∞	∞	∞	∞
1	0	7	9	∞	∞	14
2	0	7	9	20	23	11
3	0	7	9	20	20	11
4	0	7	9	20	20	11
...						

On peut montrer qu'en l'absence de cycles négatifs, on converge en moins de $|S|-1$ étapes car à chaque étape on rajoute un sommet dans un chemin simple réalisé par l'estimation. L'algorithme obtenu s'appelle l'algorithme de **Bellman-Ford**.

VI.4.ii Algorithme de Dijkstra comme relaxation

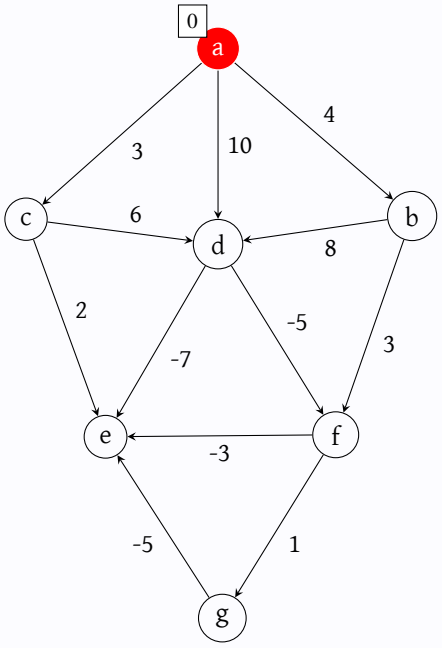
Quand on retire un sommet x de la file, si $e = x \to y$ alors soit l'arête est non tendue et alors $d(y) \leq d(x) + \pi(x \to y)$ et par hypothèse cela signifie qu'on a un chemin simple de poids $d(y)$ qui est meilleur que de passer par x. Cela signifie notamment que y a déjà été traité à un moment par l'algorithme. Soit l'arête est tendue et passer par x améliore le chemin vers y. On relâche l'arête. On a alors deux possibilités : soit y est dans la file, et on change sa priorité soit on l'insère dans la file.

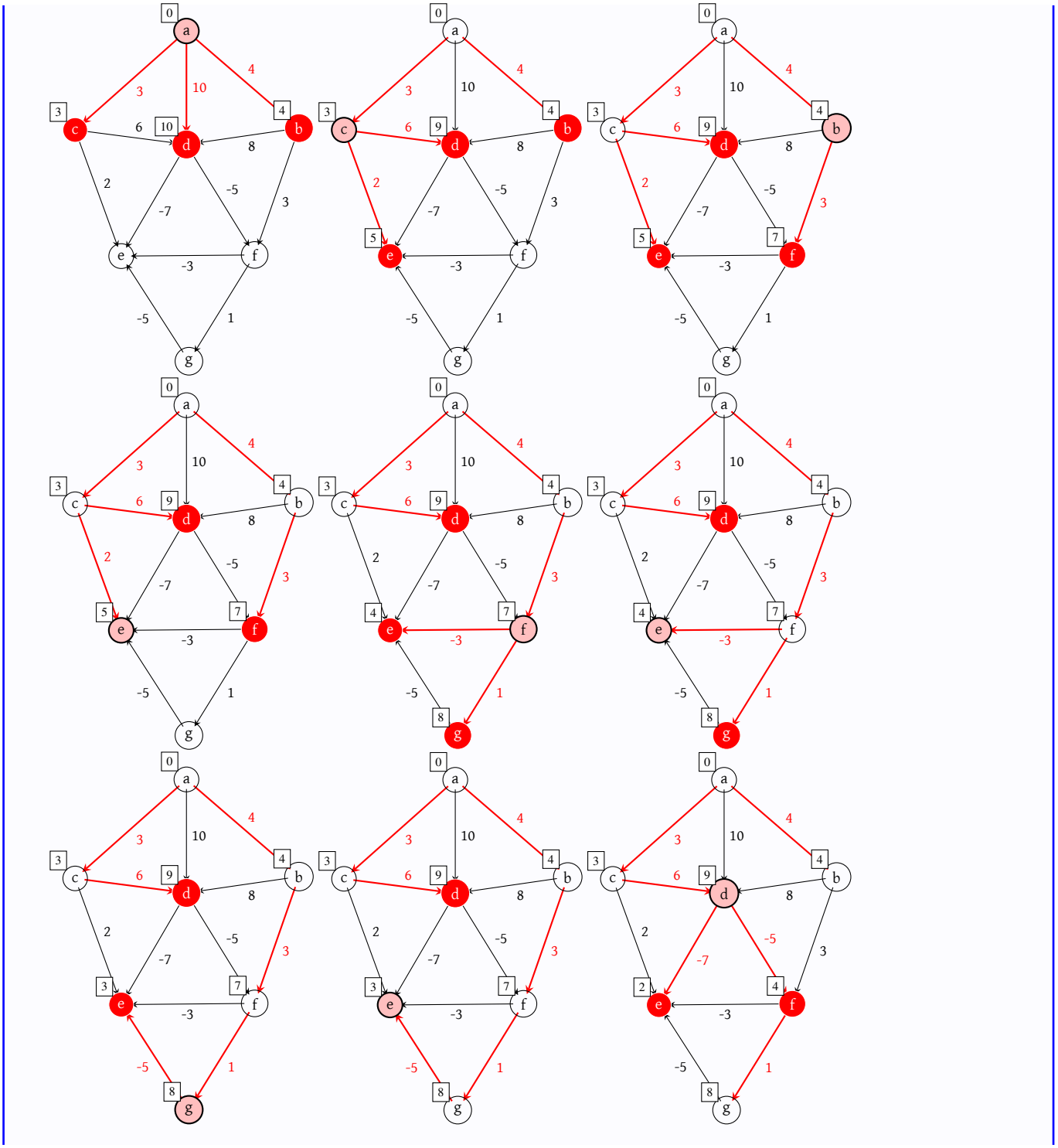
En procédant ainsi, on obtient un algorithme qui implémente le pseudo-algorithme précédent :

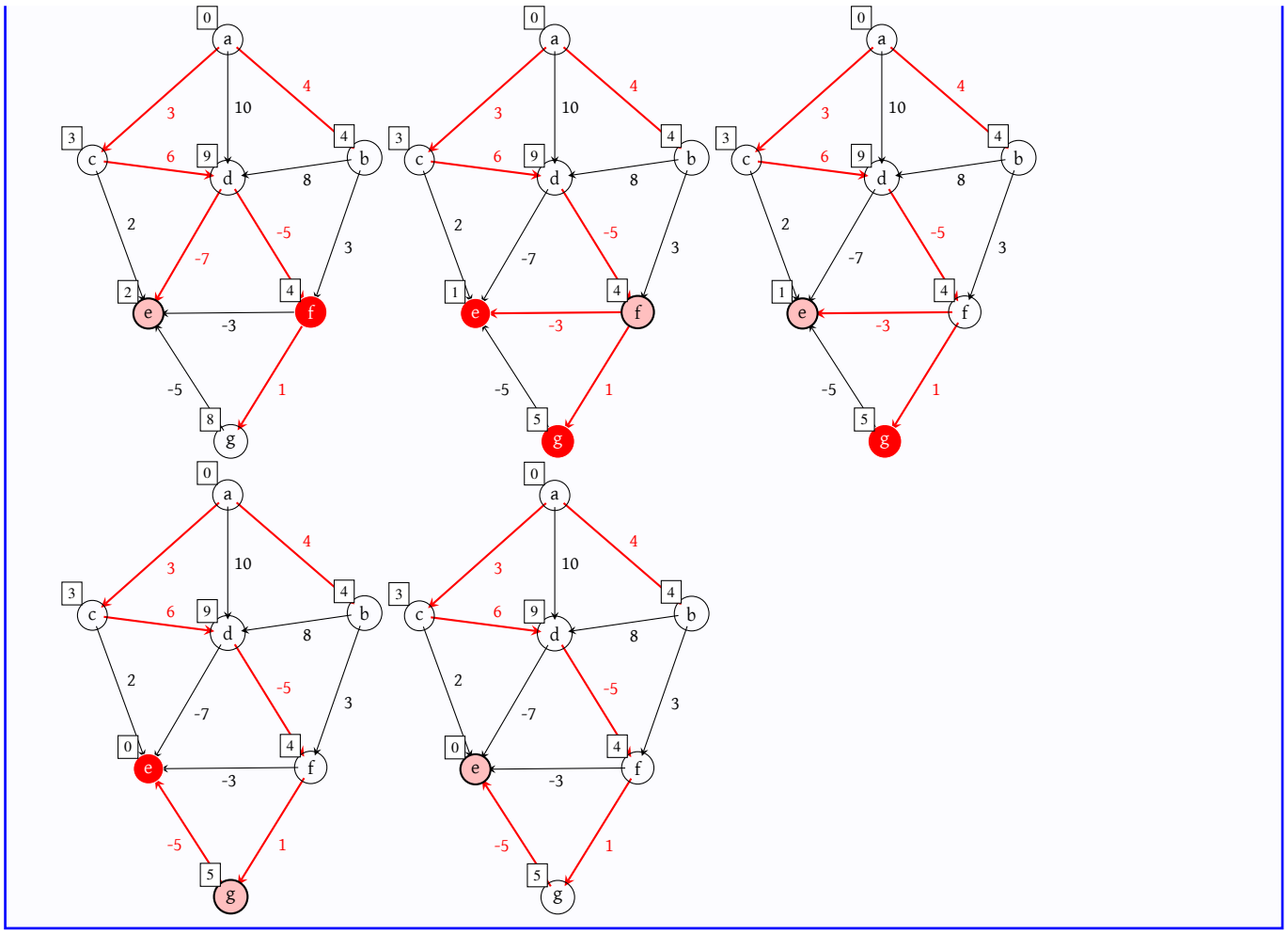
Algorithme - PROTODIJKSTRA
- Entrées :
  - ⋆ Un graphe pondéré $G = (S, A, \pi)$ sans cycle négatif
  - ⋆ Un sommet $s \in S$
- ⋆ On crée un tableau `d` avec $d(s) = 0$ et $d(x) = \infty$ pour $x \neq s$.
- ⋆ On crée une file de min-priorité `a_traiter` et on ajoute `s` avec priorité `0`
- ⋆ Tant que `a_traiter` est non vide
  - ○ On retire le sommet `x` de `a_traiter`
  - ○ Pour chaque arête $x \to y$ tendue vis-à-vis de d
    - ☐ On relâche l'arête dans d
    - ☐ Si y est dans `a_traiter`
      - Alors, on remplace sa priorité par la nouvelle valeur de $d(y)$
      - Sinon, on insère y avec priorité $d(y)$

Exemple L'exemple suivant permet d'observer l'algorithme sur un graphe comportant des poids négatifs. On a indiqué en rouge les sommets sur la file et en rose le sommet actuellement traité. Les arêtes de parentés sont présentées comme dans l'exemple précédent à mesure qu'elles sont modifiées.

On va ainsi partir du graphe suivant et du sommet source a :




a
b
c
d
e
f
g
0
3
4
3
10
4
3
6
10
8
2
-5
-7
3
-3
-5
1
0
3
4
3
10
4
3
6
9
8
2
-5
-7
3
5
-3
-5
1
0
3
4
3
10
4
3
6
9
8
2
-5
-7
3
5
7
-3
-5
1
0
3
4
3
10
4
3
6
9
8
2
-5
-7
3
5
7
-3
-5
1
0
3
4
3
10
4
3
6
9
8
2
-5
-7
3
4
7
-3
-5
1
0
3
4
3
10
4
3
6
9
8
2
-5
-7
3
4
7
-3
-5
1
0
3
4
3
10
4
3
6
9
8
2
-5
-7
3
3
7
-3
-5
1
0
3
4
3
10
4
3
6
9
8
2
-5
-7
3
3
7
-3
-5
1
0
3
4
3
10
4
3
6
9
8
2
-5
-7
3
2
4
-3
-5
1



Contrairement à cet exemple, on constate, comme on l'a fait dans le paragraphe précédent, qu'avec des poids positifs l'algorithme particulièrement bien et d'une manière assez proche du parcours en largeur. Notamment, un sommet n'est placé qu'une fois sur la file de priorité. On constate donc une vague de traitement qui progresse linéairement sur le graphe.

On peut donc se permettre de ne pas avoir à implémenter l'opération assez pénible de mise à jour de la priorité.

Remarque Cette opération n'est pas vraiment compliquée en raison du tas car il suffit de changer la valeur et de faire remonter l'élément le long de sa branche en direction de la racine en cas d'inversion de priorité. Le problème est qu'on a besoin de préserver un lien entre les sommets et leur emplacement dans le tas afin de savoir où est le nœud qui le contient. Le plus simple est de rajouter un dictionnaire, par exemple avec une table de hachage, mais cela alourdit conséquemment l'implémentation du tas car il faut maintenir l'intégrité du dictionnaire.

A la place, on va ajouter un sommet avec une autre priorité, meilleure, et il sera ainsi retiré avant. Pour ne pas traiter deux fois un sommet, on adopte alors une notion de marquage comme dans les parcours précédents. On en déduit l'algorithme de Dijkstra :

VI.5 Floyd-Warshall

Pour résoudre le problème PlusCourtCheminToutCouple dans le cas d'un graphe orienté, l'algorithme de Floyd-Warshall va considérer une énumération $x_1, \ldots, x_n$ des sommets du graphe et construire une famille de matrices $dist^{(k)}$ où $dist^{(k)}(i,j)$ indique le poids d'un plus court chemin de i à j **dont les sommets intermédiaires** sont dans $x_1, \ldots, x_k$ lorsqu'un tel chemin existe, et ∞ sinon.

On a donc $dist^{(0)}(x,y) = \begin{cases} 0 \text{ si } x = y \\ \pi(x \to y) \text{ si } x \to y \in A \\ \infty \text{ sinon} \end{cases}$.

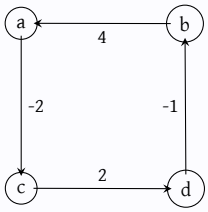
On remarque rapidement, **en l'absence de cycle négatif**, que ces matrices vérifient l'égalité, pour $k > 0$:

$$dist^{(k)}(i,j) = \min\left(dist^{(k-1)}(i,j), dist^{(k-1)}(i,k) + dist^{(k-1)}(k,j)\right)$$

En effet, un plus court chemin étant nécessairement élémentaire, soit il ne passe pas par k, soit il passe exactement une fois par k.

On en déduit un algorithme consistant à calculer ces $n+1$ matrices et les calculer avec trois boucles for imbriquées.

Exemple Si on considère le graphe



On va avoir la suite de matrices suivantes, dans l'ordre d'énumération alphabétique des sommets :

$$dist^{(0)} = \begin{pmatrix} 0 & \infty & -2 & \infty \\ 4 & 0 & \infty & \infty \\ \infty & \infty & 0 & 2 \\ \infty & -1 & \infty & 0 \end{pmatrix}$$

$$dist^{(1)} = \begin{pmatrix} 0 & \infty & -2 & \infty \\ 4 & 0 & 2 & \infty \\ \infty & \infty & 0 & 2 \\ \infty & -1 & \infty & 0 \end{pmatrix}$$

$$dist^{(2)} = \begin{pmatrix} 0 & \infty & -2 & \infty \\ 4 & 0 & 2 & \infty \\ \infty & \infty & 0 & 2 \\ 3 & -1 & 1 & 0 \end{pmatrix}$$

$$dist^{(3)} = \begin{pmatrix} 0 & \infty & -2 & 0 \\ 4 & 0 & 2 & 4 \\ \infty & \infty & 0 & 2 \\ 3 & -1 & 1 & 0 \end{pmatrix}$$

$$dist^{(4)} = \begin{pmatrix} 0 & -1 & -2 & 0 \\ 4 & 0 & 2 & 4 \\ 5 & 1 & 0 & 2 \\ 3 & -1 & 1 & 0 \end{pmatrix}$$

Une première version consisterait à calculer les matrices ce qui prend $O(|S|^3)$ espaces. On peut optimiser cela avec une paire de matrices qui changent de rôle, mais on peut faire mieux. En effet, on a $d^{(k)}(i,k) = d^{(k-1)}(i,k)$ car le chemin associé est élémentaire et il ne peut passer qu'une fois par k. De même, pour $d^{(k-1)}(k,j)$. On en déduit donc qu'on peut effectuer le calcul *en place* dans une seule matrice. On obtient alors l'algorithme suivant :

Algorithme - FLOYD-WARSHALL

- Entrées :
  - ⋆ Un graphe pondéré $G = (S, A, \pi)$ et une énumération $x_1, \ldots, x_n$ de S
- ⋆ On initialise une matrice d telle que $d(i,j) = \pi(i \to j)$ si $i \to j \in A$ et $= \infty$ sinon
  - ⋆ Pour k allant de 1 à n

- Pour i allant de 1 à n
  - Pour j allant de 1 à n
    - $d(i,j) = \min(d(i,j), d(i,k) + d(k,j))$

La matrice renvoyée vérifie ainsi $d(i,j) = dist(x_i, x_j)$.

Remarque Il est aussi possible d'imposer que les plus courts chemins soient de longueur au moins 1, ce qui va mettre des ∞ sur la diagonale dans $dist^{(0)}$ et qui permettra, en fin d'algorithme d'en déduire la présence de cycle ainsi que le poids d'un poids plus court cycle issu de i dans $d(i,i)$.

Il y a de nombreux contextes dans lesquels la détermination d'une matrice des plus petits poids est suffisante. Cependant, si on a également besoin des chemins eux-mêmes, il est possible de construire une matrice de parenté P où P_{ij} indique le prédécesseur de j dans un plus court chemin de i à j. On utilisera le symbole $\perp$ quand un tel chemin n'existe pas.

Pour calculer P il suffit d'adapter l'algorithme précédent en considérant les $P^{(k)}$ et en posant :

$$P_{ij}^{(0)} = \begin{cases} i & \text{si } i \to j \in A \\ \perp & \text{sinon} \end{cases}$$

Puis,

$$\text{Pour } k \geq 1, P_{ij}^{(k)} = \begin{cases} P_{ij}^{(k-1)} & \text{si } d_{ij}^{(k)} = d_{ij}^{(k-1)} \\ P_{kj}^{(k-1)} & \text{sinon} \end{cases}$$

Ce qui revient à changer le prédécesseur selon la résolution du min.

Remarque
- En appliquant Floyd-Warshall a un graphe quelconque, on peut détecter la présence de cycles de poids négatifs. En effet, si un tel cycle existe, on aura nécessairement $d(i,i) < 0$ à la fin.
- Floyd-Warshall a une application importante dans le cas du calcul de la fermeture transitive d'un graphe. En effet, il suffit de considérer la matrice d'adjacence au départ et d'appliquer Floyd-Warshall pour en déduire la matrice d'adjacence de la fermeture transitive où les arêtes sont en plus pondérés par la longueur du chemin qui le réalise.