Séquence 15 - Graphes

Contents

Vocabulaire à connaître: graphe, relation d'adjacence, sommet, étiquetage d'un graphe, arc, voisin, prédécesseur d'un sommet, successeur d'un sommet, degré entrant d'un sommet, degré sortant d'un sommet, graphe orienté, graphe non orienté, graphe pondéré, chemin, longueur d'un chemin, boucle, cycle, connexité d'un graphe non orienté, composante connexe d'une graphe non orienté, forte connexité d'un graphe orienté, composantes fortement connexes d'un graphe orienté, graphe bi-parti, arbre, forêt, matrice d'adjacence, listes d'adjacence.

Applications des graphes:

- réseau routier
- réseau info
- réseau social (virtuel ou non)
- labyrinthes
- cartographie

- bio-informatique
- algorithmes d'optimisation
- planification, ordonnancement de tâches
- résolution SAT (graphe d'implication 2-SAT)
- graphes de situations en théorie des jeu, jeu de Nim...

Dans les applications concrètes, les graphes sont souvent de très grandes tailles, bien au-delà de 1000 sommets.

I. Structure de données abstraite de graphe - Vocabulaire

I. 1. Graphes orientés (directed graph)

I. 1. a. Définition, représentations, étiquetage

Définition 1 (Graphe orienté)

Un graphe orienté $g = (V, E)$ est défini par un ensemble V de sommets (V pour *vertex*, sommet en anglais) et un ensemble $E \subseteq V \times V$ de couples de sommets appelés **arcs** (E comme *edge*, arc en anglais).

Remarque. E peut être vu comme une relation binaire homogène sur l'ensemble des sommets V, appelée relation d'adjacence.

Notation 1

Lorsque ces ensembles seront finis, ce qui sera le cas pour la quasi-intégralité des situations que nous rencontrerons, on notera n_v le nombre de sommets et n_e le nombre d'arcs.

$$n_v = \text{Card}V, \quad n_e = \text{Card}E$$

Remarque (Important: majoration du nombre d'arcs). Dans le cas où E et V sont finis, on a toujours:

$$n_e \leq n_v^2$$

Cette majoration découle immédiatement de la définition de E comme sous-ensemble de $V \times V$.

Définition 2 (Étiquetage d'un graphe)

Chaque sommet d'un graphe est associé à une donnée, qui peut être de type quelconque, personnalisé, très simple ou complexe. Le type des données peut ne pas être le même pour tous les sommets, mais c'est très souvent le cas.
En général, les graphes sont étiquetés: les n_v sommets se voient attribué chacun un identifiant unique. L'étiquette permet d'identifier de manière unique un sommet et de réaliser l'association avec la donnée attachée à ce sommet, qui souvent stockée à part.

Remarque (Étiquetage par des entiers). Dans ce cours, la plupart des graphes seront étiquetés par des entiers consécutifs $[\![0,\, n_v-1]\!]$. **Attention, l'étiquetage commencera toujours à 0 dans le cadre de ce cours.** Cet étiquetage est très simple et très favorable car il permet de relier un sommet à sa donnée utile (stockée à part, pour des raisons évidentes d'optimisation) par un simple tableau associatif. On notera alors v_i le sommet étiqueté par l'entier i.

Attention. Il n'y a aucune manière naturelle d'étiqueter les sommets d'un graphe par des entiers. En fait, il y a $n_v!$ manières de le faire! Nous verrons cependant qu'il peut exister des étiquetages plus favorables que d'autres pour certains algorithmes.

Remarque (Représentation d'un graphe). Un graphe orienté est généralement représenté en dessinant des nœuds étiquetés par des entiers, et des flèches entre ces nœuds symbolisant les arcs existants.

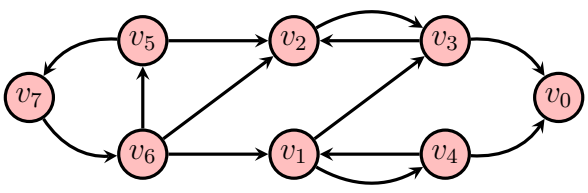
Exemple 1

Le graphe ci-dessous correspond aux ensembles:

$$V = \{v_0,\, v_1,\, v_2,\, v_3,\, v_4,\, v_5,\, v_6,\, v_7\}\ ,$$

et

$$\begin{aligned} E \ &= \{(v_1, v_4),\, (v_1, v_3),\, (v_2, v_3),\, (v_3, v_0),\, (v_3, v_2),\, (v_4, v_0),\\ &\quad (v_4, v_1),\, (v_5, v_2),\, (v_5, v_7),\, (v_6, v_1),\, (v_6, v_2),\, (v_6, v_5),\, (v_7, v_6)\} \end{aligned}$$



On a donc $n_v = 8$ et $n_e = 13$.

I. 1. b. Arcs, chemins, cycles

Définition 3 (Successeur(s), prédécesseur(s), degré sortant et degré entrant d'un sommet)

Si $(u, v) \in E$, alors v est un **successeur** de u et u est un **prédécesseur** de v.
On note $u \to v$ la présence de cet arc de u vers v.
On appelle également **voisins** ou **sommets adjacents** l'ensemble des successeurs d'un sommet.
On appelle **degré sortant** de u, et on note $\delta_+(u) \in \mathbb{N}$ le **nombre** d'arcs $u \to \dots$ sortant de u.
On appelle **degré entrant** de u, et on note $\delta_-(u) \in \mathbb{N}$ le **nombre** d'arcs $\dots \to u$ entrant dans u.

Exemple 2

Sur le graphe exemple ci-dessus:

$$\delta_+(v_0) = 0,\ \delta_-(v_0) = 2$$

$$\delta_+(v_2) = 1,\ \delta_-(v_2) = 3$$

$$\delta_+(v_6) = 3,\ \delta_-(v_6) = 1$$

Définition 4 (Chemin dans un graphe)

Un **chemin** du sommet u au sommet v dans un graphe $g = (V, E)$ est une séquence de sommets $u_0,\ u_1,\ \dots,\ u_n$ de V tels que $u_0 = u$, $u_n = v$ et $(u_i, u_{i+1}) \in E,\ \forall i \in [\![0, n-1]\!]$:

$$u = u_0 \to u_1 \to u_2 \to \dots \to u_{n-1} \to u_n = v$$

Le **longueur d'un chemin** est définie comme le nombre d'arcs traversé de la source u jusqu'à la destination v.

Notation 2

On notera $u \to^* v$ l'existence d'un chemin entre u et v dans le graphe.
La relation $\to^*$ peut être vu comme la clôture transitive de la relation binaire homogène E sur V.

Remarque. Il y a toujours un chemin de longueur 0 entre un sommet et lui-même.
En d'autres termes, la relation binaire E est réflexive.

Définition 5 (Boucle, chemin simple, cycle, graphe acyclique)

On appelle **boucle** un chemin constitué d'un seul arc $u \to u$. Elles sont dessinées comme de petites boucles au dessus des sommets sur les schémas de graphe.
On appelle **chemin simple** un chemin dans lequel tous les arcs traversés sont différents: on ne repasse jamais deux fois par le même arc.
On appelle **cycle** un **chemin simple de longueur strictement positive** de type $u \to^* u$, c'est-à-dire dont la source et la destination sont identiques.
Si un graphe ne possède aucun cycle, on dira qu'il s'agit d'un **graphe acyclique** (DAG pour *directed acyclique graph* en anglais, graphe acyclique orienté).

Remarque. Si le graphe ne contient pas de boucle, on a une majoration un peu plus fine du nombre d'arcs:

$$n_e \leq n_v \times (n_v - 1)$$

On a en effet n_v choix pour le sommet de départ de l'arc, et seulement $n_v - 1$ choix pour le second car on ne peut pas choisir à nouveau le sommet déjà choisi pour le départ de l'arc.

Remarque. Une boucle, avec cette définition, est un cycle de longueur 1.

Remarque. Il arrive souvent que l'on considère que tous les sommets d'un graphe sont bouclés, c'est-à-dire que tous les arcs (u, u) existent, ce qui signifie que la relation d'adjacence définie par E est considérée comme réflexive. Ce sera notamment le cas pour tous les algorithmes de calcul du plus court chemin que nous implémenterons, car chaque sommet sera considéré comme en lien avec lui même, avec une distance 0. Dans d'autres contextes, par exemple en théorie des automates, on précise explicitement la présence ou non de boucles.

I. 1. c. Forte connexité, composantes fortement connexes

Définition 6 (Graphe orienté fortement connexe)

On dit qu'un graphe orienté est fortement connexe si, pour toute paire de sommets (u, v), il existe un chemin $u \to^* v$ menant de u à v dans ce graphe. Autrement dit tout sommet est accessible depuis n'importe quel autre sommet.

Définition 7 (Composantes fortement connexes d'un graphe orienté)

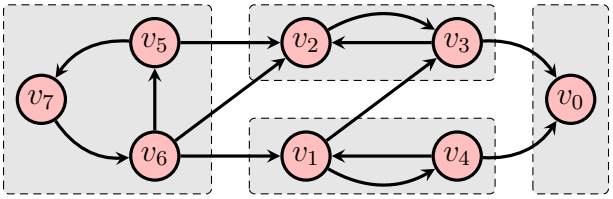
Une composante fortement connexe d'un graphe orienté est un sous-ensemble V_k de V tel que pour tout couple (u_k, v_k) de sommets de V_k, il existe toujours un chemin $u_k \to^* v_k$. et qui est maximal dans V pour la relation d'inclusion.

Propriété 1

Les composantes fortement connexes d'un graphe orienté sont disjointes.

Exemple 3

Reprenons notre graphe orienté exemple, et représentons en gris les composantes fortement connexes:



I. 2. Graphes non-orientés

I. 2. a. Définitions et adaptation des définitions précédentes

Définition 8 (Graphe non orienté)

Un graphe orienté est un graphe pour lequel la relation binaire d'adjacence E est **symétrique**.
Cela signifie que si un arc (u, v) allant du sommet u au sommet v est présent, alors il existe un arc (v, u) qui relie les deux sommets dans l'autre sens.

Remarque. Dans le contexte des graphes non-orientés, on parle parfois de **noeuds** au lieu de parler de sommets, et d'**arêtes** au lieu de parler d'arcs.

Remarques (Qu'est-ce qui change par rapport aux graphes orientés?).

- Il n'y a plus lieu de distinguer prédécesseur et successeur. On dira simplement que deux sommets sont voisins.
- On ne distinguera plus non plus degré entrant et degré sortant. Le degré $\delta(u)$ d'un sommet $u \in V$ sera simplement le nombre de ses voisins.
- La notion de chemin reste exactement la même.
- Il n'y a pas de cycle de longueur 2 dans un graphe non orienté. En effet, le chemin $u \to v \to u$ possède une répétition (deux fois le même arc (u, v) et n'est donc pas considéré comme un cycle car répétition (ce n'est pas un chemin simple)
- On a une nouvelle majoration du nombre d'arcs car dans le dénombrement, on ne distingue plus les arcs (u, v) et (v, u). On a donc au plus $\binom{n}{2}$ arcs, c'est à dire:

$$n_e \leq \frac{n_v\,(n_v - 1)}{2}$$

Les définitions de connexité et de composante connexe ne changent pas mais il faut noter que l'on parle de **forte connexité pour un graphe orienté** et simplement de **connexité pour un graphe non-orienté**. (sans le **fortement**, qui signifie donc en prenant en compte l'orientation).
De même, on parle de **composante fortement connexe pour un graphe orienté** et simplement de **composante connexe pour un graphe non-orienté**.

Définition 9 (Graphe connexe)

On dit qu'un graphe (vu comme un graphe non orienté) est connexe si, pour tout arc (u, v), il existe un chemin $u \to^* v$ menant de u à v dans ce graphe. Autrement dit tout sommet est accessible depuis n'importe quel autre sommet.

Définition 10 (Composantes connexes d'un graphe non-orienté)

Une composante connexe d'un graphe non-orienté est un sous-ensemble V_k de V tel que pour tout couple (u_k, v_k) de sommets de V_k, il existe toujours un chemin $u_k \to^* v_k$. et qui est maximal dans V pour la relation d'inclusion.

Propriété 2

Les composantes connexes d'un graphe non-orienté sont disjointes.

Exemple 4

Si on regarde le graphe exemple en oubliant l'orientation des arcs et en considérant ces arcs comme bidirectionnels, ce graphe devient un graphe non-orienté ayant une seule composante connexe.

I. 2. b. Cas particulier: les arbres généraux

Nous avons étudié les arbres généraux dans un précédent TP. Il est intéressant de comprendre que les arbres généraux tels que nous les avons définis sont un cas particulier de graphe non orienté.

Définition 11 (Arbre, forêt)

Un arbre (général) est un **graphe non orienté non vide acyclique et connexe**. Un ensemble d'arbres est appelé une forêt.

Comparons cette définition à celle que nous avions donnée:

Définition 12 (Arbre enraciné, forêt)

Un **arbre** général est un ensemble de $n \geq 1$ nœuds structurés de la manière suivante :

- un nœud particulier r est appelé la **racine** de l'arbre ;
- les $n-1$ nœuds restants sont partitionnés en $k \geq 0$ sous-ensembles disjoints reliés à la racine qui forment autant d'arbres, appelés **sous-arbres** de r ;
- la racine r est liée à la racine de chacun des k sous-arbres.

Ces deux définitions décrivent bien les mêmes objets, sauf sur un point: la présence d'un sommet particulier appelé **racine**. En toute rigueur, ce que nous avons étudié jusqu'ici sont des arbres dit **enracinés**, c'est-à-dire des arbres pour lesquels un sommet joue un rôle particulier en tant que point de départ de l'arbre.

Propriété 3

Tout arbre (V, E) qui possède n_v sommets contient exactement $n_e = n_v - 1$ arcs.

Démonstration

Nous allons montrer cette propriété par récurrence forte sur le nombre de sommets n_v de l'arbre. La propriété a démontrer s'énonce ainsi:
$\mathcal{P}(n_v)$: Pour un arbre ayant n_v sommets, on a toujours $n_e = n_v - 1$

Initialisation: Un arbre, d'après ces deux définitions, est non vide, il possède toujours au moins 1 sommet. Pour $n_v = 1$, l'arbre a un unique nœud et a bien $n_e = n_v - 1 = 1 - 1 = 0$ arcs. $\mathcal{P}(1)$ est vraie.

Vérification de l'hypothèse de récurrence. On fixe $n_v > 1$ et un arbre $g = (V, E)$ ayant n_v sommets. On suppose que, pour tout arbre ayant $k < n_v$ sommets, la propriété $\mathcal{P}(k)$ est vérifiée. Par définition, g a une racine $v_0 \in V$. Ce sommet est relié par des arcs $(v_1, ...) \in E$ à $p \geq 1$ sous-arbres p est forcément strictement positif car sinon, cela signifierait que $V = \{v_1\}$, que l'arbre n'a qu'un sommet, ce qui contredirait $n_v > 1$. Par ailleurs, comme g est un arbre, il est acyclique, ce qui impose que les sous-arbres $V_1, V_2, \ldots, V_p$ sont disjoints. En effet, s'ils ne l'étaient pas, il existerait un sommet v commun à deux sous-arbres V_i et V_j. On aurait alors un chemin passant par la racine, puis passant par le sous-arbre V_i jusqu'à v, ce qui existe car g est connexe, puis par le sous-arbre V_j à nouveau jusqu'à la racine, encore grâce à la connexité. Ainsi, on aurait réussi à construire un chemin de longueur strictement positive partant de la racine et revenant à la racine, c'est-à-dire un cycle, ce qui est une contradiction car un arbre est, par définition, acyclique.

Le $V_1, V_2, \ldots, V_p$ étant disjoints, on peut maintenant écrire, en notant $n_{e,k}$ le nombre d'arcs du sous-arbre k et $n_{v,k}$ le nombre de ses sommets:

$$n_e = p + n_{e,1} + n_{e,2} + \ldots + n_{e,p}$$

$$n_v = 1 + n_{v,1} + n_{v,2} + \ldots + n_{v,p}$$

On applique maintenant l'hypothèse de récurrence aux sous-arbres $V_1, V_2, \ldots, V_p$:

$$\forall k \in [\![1, p]\!],\ n_{e,k} = n_{v,k} - 1$$

En réinjectant ces relations dans la précédente:

$$n_e = p + (n_{v,1} - 1) + (n_{v,2} - 1) + \ldots + (n_{v,p} - 1)$$

$$\Leftrightarrow n_e = n_{v,1} + n_{v,2} + \ldots + n_{v,p} = n_v - 1$$

On a donc montré que $\mathcal{P}(n_v)$ est vraie

Conclusion: par principe de récurrence forte, $\mathcal{P}(n_v)$ est vraie pour tout arbre ayant n_v sommets.

I. 3. Graphes pondérés (weighted graph)

Un graphe pondéré (orienté ou non) est simplement un graphe pour lequel des informations supplémentaires ont été associées aux arcs. Ces informations peuvent représenter un coût, une distance, ou n'importe quoi d'autre.

L'information supplémentaire associée à chaque arc est, selon le contexte, appelée **distance** ou **poids**.
Souvent, dans ce contexte, la définition de la longueur d'un chemin est adaptée pour prendre en compte le poids (dans ce contexte interprété comme une distance) associé à chaque arc.

Définition 13 (Longueur d'un chemin dans un graphe pondéré)

La longueur d'un chemin dans graphe pondéré est l'accumulation des poids de tous les arcs formant le chemin, le terme accumulation devant être précisé en fonction du type de ces poids.
Souvent, les poids étant de type flottant, la longueur d'un chemin est simplement la somme (flottante) de tous les poids de tous les arcs formant le chemin.

I. 4. Graphes bipartis

Définition 14 (Graphe biparti)

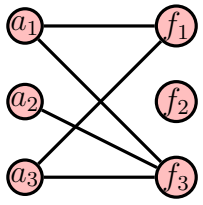
Un graphe biparti (orienté ou non, pondéré ou non) est un graphe $g = (V, E)$ dont les sommets peuvent être séparés en deux ensembles disjoints V_1 et V_2 (chaque sommet appartient soit à V_1, soit à V_2), de sorte que, pour tout arc $(u, v) \in E$, on a:

- $u \in V_1$ et $v \in V_2$
- Ou bien $u \in V_2$ et $v \in V_1$

Autrement dit, tout arc forme un pont entre les deux sous-ensembles V_1 et V_2.

Exemple 5

Ce type de graphes est en fait très courant dans les modélisations concrètes, par exemple si l'on réalise des graphes avec deux sortes de sommets, par exemple des sommets représentant des acteurs, et des sommets représentant des films. On dessine un arc $a - f$ si l'acteur a joue dans le film f Tous les arcs vont naturellement établir des ponts entre les deux sous-ensembles de sommets disjoints représentant d'une part les films, et d'autre part les acteurs.
Un tel graphe biparti ressemble par exemple à ceci (ici le graphe est non-orienté):



II. Implémentations concrètes

II. 1. Cadre

II. 1. a. Choix d'implémentation

Comme pour toute implémentation concrète, on a deux briques de base concrètes pour établir des liens entre nos objets:

- le chaînage par adresse, chaque objet stockant les adresses des autres objets auxquels il est relié
- le lien implicite s'appuyant implicite sur la contiguïté mémoire, avec des tableaux

Nous détaillons et analysons maintenant différentes implémentations concrètes possibles pour la structure de données abstraite de graphe pondéré, orienté ou non.
Toutes les implémentations concrètes ci-dessous permettent de gérer des graphes pondérés et orientés: il est ensuite très facile de limiter ces implémentations pour représenter des graphes non pondérés, ou des graphes non orientés.

Graphe non pondéré: il suffit de donner le même poids à toutes les arcs du graphe lors de leur création. Il est généralement intéressant d'attribuer un poids 1 à chaque arc pour rester cohérent sur la notion de longueur de chemin;

Graphe non orienté: il suffit de toujours créer, lors de la création d'une nouvelle arc, l'arc correspondant au trajet inverse: si l'arc (u, v) est créée, il me suffit de créer automatiquement l'arc (v, u) (symétrisation), c'est-à-dire la clotûre transitive de la relation d'adjacence E.

Pour forcer ces deux comportements, je peux stocker deux booléens supplémentaires dans la structure de donnée du graphe, permettant de maintenir en conséquence les invariants de structure:

directed: booléen indiquant si le graphe est orienté ou non

weighted: booléen indiquant si le graphe est pondéré ou non (facultatif, car il peut suffire de mettre le même poids à toutes les arcs)

II. 1. b. Stockage des données associées aux sommets

En pratique, la donnée associée à un sommet est stockée à part, quelque part dans la mémoire, et le sommet ne contient qu'une référence implicite ou explicite permettant de faire le lien avec sa donnée utile. Généralement, le lien entre un sommet et sa donnée utile est implicite et s'appuie sur l'étiquetage.
Si le graphe est étiqueté par des entiers consécutifs, on crée à par un tableau ayant autant de cases que de sommets, soit n_v. La case d'indice i de ce tableau contient un pointeur vers la donnée associée au sommet étiqueté par l'entier i.
Si le graphe n'est pas étiqueté par des entiers consécutifs mais par des éléments d'un ensemble $\mathcal{E}$, on a deux possibilités (cf cours sur le hachage):

S'il existe un mapping $h : \mathcal{E} \to [\![0, n_v - 1]\!]$, alors on peut toujours procéder avec un tableau associatif. Dans ce cas, la case d'indice $h(e)$ du tableau associatif contient la donnée associée au sommet v_e d'étiquette e. Comme h est injective, chaque contient la donnée associée à un seul sommet.

Sinon, on utilise une table de hachage avec un fonction h non injective mais proche de l'injectivité. Dans ce cas, la case d'indice $h(e)$ du tableau associatif contient une liste de pointeurs vers des données, et parmi cette liste, il y a la donnée associée au sommet v_e d'étiquette e. Comme h est presque injective, la liste associée à une case du tableau est quasiment tout le temps de longueur 0 ou 1, sauf si on a une collision, mais cela est rare si h est une fonction de hachage performante.

Dans la suite, dans un souci de clarté, on se limitera à un étiquetage des sommets par des entiers Tous les sommets du graphe sont étiquetés par des entiers consécutifs de 0 à $n_v - 1$.
Dans le cadre de ce cours, comme nous nous intéressons surtout à la structure de graphe en tant que tel et non dans un but applicatif direct, on ne se préoccupera pas, la plupart du temps, des données utiles associées au sommet. Bien sûr, dans des applications réalistes, ces données sont la plupart du temps essentielles, et le graphe n'existe pour ainsi dire que dans le but de structurer ces données utiles.

II. 1. c. Interface abstraite

Pour implémenter une structure de données de graphe, il faut, au minimum, implémenter les primitives suivantes:

```c
typedef struct wgraph_t wgraph;

// constructeur
wgraph *wgraph_init_no_edge(int nv, bool directed);

// destructeur
void    wgraph_free(wgraph **addr_g);

// accesseurs
list   *wgraph_succ(wgraph *g, int u); // liste des successeurs d'un sommet
void    wgraph_print(wgraph *g);
int     wgraph_number_of_vertices(wgraph *g);
int     wgraph_number_of_edges(wgraph *g);
bool    wgraph_has_edge(wgraph *g, int u, int v);
bool    wgraph_has_vertex(wgraph *g, int u);

// transformateurs
bool    wgraph_add_edge(wgraph *g, int u, int v, double weight);
bool    wgraph_remove_edge(wgraph *g, int u, int v);

// si on autorise un nombre variable de sommets (graphe dynamique):
int     wgraph_add_vertex(wgraph *g);
bool    wgraph_remove_vertex(wgraph *g, int u);
```

init_no_edge: constructeur qui alloue la structure de donnée associée à un graphe ayant initialement un nombre donné n_v de sommet, et sans arcs. L'utilisateur doit également indiqué s'il s'agit d'un graphe orienté ou non.

free: destructeur, libère l'intégralité de l'espace mémoire alloué pour la structure de données.

has_vertex: accesseur, fonction renvoyant la valeur vraie si le graphe possède un sommet d'étiquette i donnée, et faux sinon

has_edge: accesseur, fonction renvoyant la valeur vraie si le graphe possède un arc (v_i, v_j), où l'on a fourni les étiquettes i et j des deux sommets en entrée, et faux sinon

add_edge: transformateur, rajoute un arc (v_i, v_j) entre deux sommets d'étiquettes i et j données, avec un poids w donné. Si l'arc est déjà présent dans le graphe, la fonction se contente de remplacer le poids associé par la nouvelle valeur w fournie et renvoie la valeur faux. Si l'arête n'était pas présente, une nouvelle arête est crée et la fonction renvoie la valeur vrai.

remove_edge: transformateur, supprime un arc (v_i, v_j) entre deux sommets d'étiquettes i et j données. Si l'arc n'est pas présent, aucune suppression n'a lieu et la fonction renvoie la valeur faux. Sinon, l'arc est effectivement supprimé de la structure de donnée et la valeur vrai est renvoyée.

add_vertex: transformateur, rajoute un sommet v_i d'étiquette i donnée au graphe. Si l'étiquette i est déjà présente dans le graphe, la fonctionne fait rien et renvoie la valeur faux. Sinon, le nouveau sommet est crée et la fonction renvoie la valeur vrai.

remove_vertex: transformateur, supprime un sommetv_i d'étiquette i donnée. Supprime également tous les arcs qui ce sommet comme successeur ou prédécesseur. Si l'étiquette i n'est pas présente, aucune suppression n'a lieu et la fonction renvoie la valeur faux. Sinon, le sommet et les arcs impliquant ce sommet sont effectivement supprimées de la structure de donnée et la valeur vrai est renvoyée.

Remarque. Les deux dernières primitives ne sont utiles que si l'on implémente des **graphes dynamiques, dont le nombre de sommets évolue au cours du temps**. Le fait d'autoriser ou non une variation du nombre de sommets au cours de la vie du graphe influence fortement l'implémentation concrète d'une telle structure de données, car, dans le formalisme que nous adoptons, ce sont les sommets qui servent de référence pour désigner les arcs, et non l'inverse.

Remarque. Cette interface abstraite peut facilement se généraliser pour des étiquettes qui ne sont pas des entiers.

II. 2. Implémentation naïve avec deux listes: à oublier

II. 2. a. Principe

Une première idée naïve consisterait à représenter un graphe avec deux listes:

- Une liste de sommets: liste des étiquettes des sommets
- Une liste d'arcs: liste de triplets (i, j, w) où i et j sont les étiquettes des deux sommets et w le poids associé à l'arc

II. 2. b. Analyse de complexité

Complexité temporelle. Cette implémentation est catastrophique en terme de complexité temporelle:

has_vertex: nécessite le parcours de la liste des sommets, donc en $\Theta(n_v)$

has_edge: nécessite d'appeler `has_vertex` pour vérifier que l'arête donnée est bien formée de deux sommets existants, et nécessite le parcours de la liste des arcs jusqu'à trouver l'arête. Au pire, on parcourt toute la liste donc la complexité au pire est en en $\Theta(nv + n_e)$

succ: nécessite de parcourir toute la liste des arcs donc la complexité est en $\Theta(n_e)$, même sans être dans le pire des cas.

add_edge: il faut parcourir la liste des sommets pour valider l'existence des deux étiquettes données en entrée pour former l'arc, puis il faut parcourir la liste des arcs pour vérifier que l'arc n'existe pas déjà avant, éventuellement de l'insérer. Dans le pire des cas, si on doit parcourir les deux listes dans leur intégralité, le coût temporel est en $\Theta(n_v + n_e)$

add_vertex: il faut parcourir toute la liste des sommets pour vérifier que l'étiquette choisie pour le nouveau sommet n'existe pas déjà, et éventuellement ensuite ajouter le sommet (coût constant). La complexité temporelle est donc en $\Theta(n_v)$

remove_edge: il faut au minimum parcourir de la liste des arcs pour trouver l'arc à supprimer. Dans le pire des cas, on parcourt toute la liste des arcs et la complexité temporelle au pire est donc en $\Theta(n_e)$

remove_vertex: il faut parcourir la liste des sommets pour valider l'existence du sommet à supprimer et le supprimer effectivement le cas échéant (suppression à coût constant). Puis il faut parcourir la liste de tous les arcs et supprimer tous les arcs utilisant ce sommet. Le coût temporel est donc au pire en $\Theta(n_v + n_e + \delta_+ u + \delta_-(u))$ si u est le sommet à supprimer.

Cette analyse suffit à montrer que cette première implémentation se révélera peu efficace dès que l'on manipulera des graphes de taille plus importante. En effet, un graphe à n_v sommets peut comporter jusqu'à $\begin{pmatrix} n \\ 2 \end{pmatrix}$ arcs. Les coûts en $\Theta(n_e)$ deviennent alors prohibitifs, de l'ordre de $\Theta(n_v^2)$.

Complexité spatiale. L'un des rares avantage de cette implémentation est peut-être sa complexité spatiale, plutôt bien ajustée à la quantité d'information nécessaire pour représenter le graphe.

Gestion des graphes dynamiques. Autre avantage, cette implémentation est relativement flexible et permet d'implémenter assez naturellement les graphes dynamiques. A noter également que cette implémentation s'adapte aussi assez bien aux graphes dont les sommets sont étiquetés par autre chose que des entiers.

Toutefois, ces quelques avantages ne compensent pas les coûts temporels catastrophiques des primitives de manipulation et cette implémentation est à proscrire sauf pour de très petits graphes.

II. 3. Matrice d'adjacence

II. 3. a. Principe

La première implémentation consiste à représenter le graphe par une matrice $M = (m_{ij})$ de taille $n_v \times n_v$ dont les coefficients sont des poids:

$$m_{ij} = \begin{cases} \text{valeur par défaut} & \text{si l'arc } (v_i, v_j) \text{ n'est pas présent dans le graphe} \\ 0 \leq w < \infty & \text{si l'arc } (v_i, v_j) \text{ est présent dans le graphe} \end{cases}$$

et w représente le poids (positif, fini) de l'arc lorsqu'il existe.
Dans le cas où l'on n'a pas du tout besoin d'implémenter des graphes pondérés, on peut se contenter d'une matrice de booléen telle que:

$$m_{ij} = \begin{cases} \texttt{false} & \text{si l'arc } (v_i, v_j) \text{ n'est pas présent dans le graphe} \\ \texttt{true} & \text{si l'arc } (v_i, v_j) \text{ est présent dans le graphe} \end{cases}$$

Remarque. Dans beaucoup d'applications, notamment pour les algorithmes de calcul de plus court chemin, la relation d'adjacence est considérée comme réflexive (une boucle au dessus de chaque sommet) et les poids (ici des distances) associés à ces boucles sont mis à 0 (`true` pour les graphes non-pondérés), pour signifier qu'un sommet est toujours en lien avec lui même, avec une distance 0 à lui-même dans le cas des graphes pondérés. Dans ce cas, les poids sur la diagonale sont forcés à 0
Toujours dans le cadre des applications type calcul de plus court chemin, lorsque l'arc n'existe pas, on préfère lui attribuer une valeur par défaut correspondant à un poids très lourd, par exemple la valeur réelle positive la plus grande possible représentable en machine, `DBL_MAX` en C.
Toutes les valeurs diagonales sont généralement mises à dans ce cas car un sommet est toujours connecté à lui-même, donc l'arc (i, i) existe toujours.

II. 3. b. Analyse de complexité

Complexité temporelle. Cette implémentation est très performante en terme de complexité temporelle des primitives:

has_vertex: un test $\Theta(1)$

has_edge: une lecture de case dans la matrice, coût constant grâce à l'adressage direct, donc en $\Theta(1)$

succ: pour calculer la liste de tous les successeurs du sommet d'étiquette i, il faut parcourir tout le ligne i de la matrice, on est donc en $\Theta(n_v)$

add_edge: il suffit d'effectuer une écriture dans une case de la matrice, donc en $\Theta(1)$

add_vertex: difficile...

remove_edge: il suffit d'effectuer une écriture dans une case de la matrice pour remettre la valeur par défaut à la case d'indice (i, j) correspondant à l'arc (v_i, v_j) que l'on souhaite supprimer, donc en $\Theta(1)$

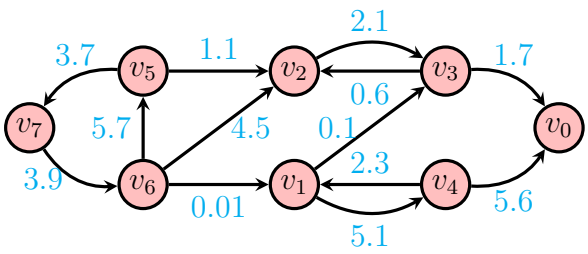
remove_vertex: difficile...

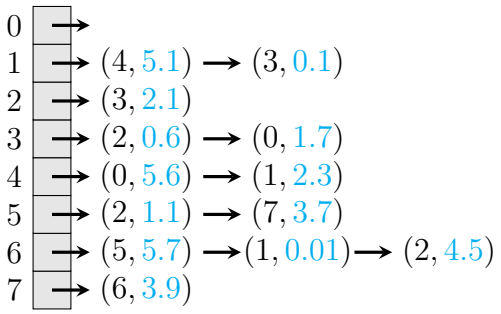
Complexité spatiale. Cette implémentation est gourmande en mémoire puisqu'elle est en $\Theta(n_v^2)$. C'est particulièrement dommage dans le cas de **graphes creux** (*sparse graphs*), qui ont beaucoup de sommets mais peu d'arcs, car cette structure de données utilise de l'espace mémoire pour représenter l'absence d'arc.
Gestion des graphes dynamiques. L'implémentation de graphes dynamiques, ayant un nombre de sommets variable au cours de leur vie, est assez peu naturelle et coûteuse avec cette implémentation, et peut entraîner un surcoût spatial important en créant des lignes et des colonnes fantômes dans la matrice à la place des sommets supprimés, qui ne pourront être réutilisées sauf en s'attelant à une implémentation complexe et temporellement coûteuse d'un mécanisme de recyclage.
L'implémentation concrète par matrice d'adjacence est donc très adaptée pour les graphes pleins, très connectés ou pour les petits graphes.

II. 4. Tableau ou liste d'adjacence

II. 5. Principe

L'idée de cette implémentation est principalement de limiter l'espace mémoire occupé par rapport à l'implémentation par matrice d'adjacence, notamment pour les graphes très creux, en évitant d'utiliser de l'espace mémoire pour indique une absence d'arc.
Pour chaque sommet, on stocke la liste des arcs partant de ce sommet, avec leur poids. Pour un sommet v_i, si l'arc (v_i, v_j) est présent dans le graphe avec le poids w, on stocke simplement le couple (v_j, w) dans une liste associée au sommet v_i.
On a donc un liste de couples du type (v_j, w) pour chaque sommet v_i. Si la liste associée à un sommet v_i est vide, c'est que ce sommet n'est le prédécesseur d'aucun autre sommet (autrement dit qu'il n'a pas de successeurs).





Il existe plusieurs variantes de cette implémentation, cela l'implémentation concrète que l'on choisit:

Choix 1: d'une part pour le stockage correspondant au tableau associatif vertical sur le schéma: liste, tableau ou table de hachage

Choix 2: d'autre part pour le stockage correspondant aux listes horizontales sur le schéma: liste, tableau ou table de hachage

Sur le schéma ci-dessus, on a plutôt représentée une implémentation par tableau (choix 1) de listes (choix 2).

II. 5. a. Analyse de complexité

Pour le choix 1, le choix est assez simple: si l'on implémente les graphes statiques, avec des étiquettes entières consécutives, le tableau est tout indiqué. Si l'on implémente des graphes dynamiques, le tableau est à exclure. La table de hachage est tout indiquée car elle évitera le coût d'accès linéaire à l'élément i avant d'accéder la liste des successeurs de v_i. Ce coût se fera en temps constant grâce aux bonnes propriété de pseudo-injectivité de la fonction de hachage. De plus, une table de hachage permet de gérer le cas d'étiquettes plus générales. Enfin, la table de hachage permet de gérer facilement l'ajout ou la suppression d'un sommet.

Pour le choix 2, le tableau statique est peu astucieux... car on retombe alors sur l'implémentation par matrice d'adjacence et son inconvénient majeur qui est la complexité spatiale en $\Theta(n_v^2)$ et la difficulté à l'adapter aux graphes dynamiques. Pour limiter la complexité spatiale et améliorer le caractère dynamique de la structure, les listes et les les tables de hachage sont tout indiquées pour le choix 2. Toutefois, l'utilisation de listes entrainera un coût linéaire en le nombre de successeurs, donc en $\Theta(\delta_+(v_i))$ chaque fois qu'on l'on devra effectuer une recherche sur les successeurs de v_i, ce qui se produit pour `has_edge`, `add_edge`, `remove_edge`, `remove_vertex`. La encore, stocker ces listes de successeurs sous la forme de tables de hachage permet de garantir un accès en temps à n'importe quel successeur (v_j, w) d'un sommet v_i grâce aux propriétés des tables de hachage, moyennant toutefois un très léger surcoût spatial et en considérant que le calcul de l'empreinte $h(v_j)$ d'un successeur se fait très rapidement, en temps constant.

Ainsi, la solution optimale, en terme de complexité temporelle des primitives, de complexité spatiale et d'adaptabilité aux graphes dynamiques et aux étiquettes non-entières est la solution table de hachage de tables de hachage, pour les deux choix.
Toutefois, un tel niveau d'optimisation n'est pas toujours requis, et on se contentera généralement d'une implémentation par tableau de listes.

II. 6. Bilan global des implémentations évoquées

Faites un joli tableau récapitulatif indiquant, pour chacune des 8 implémentations proposées, la complexité temporelle de toutes les primitives, l'occupation mémoire et l'adaptabilité de l'implémentation aux graphes dynamiques de chaque implémentation concrète.

III. Algorithmique des graphes

III. 1. Parcours en profondeur (Depth-First Search DFS)

L'idée du parcours en profondeur est d'explorer, à partir d'un sommet source choisi v_{source} chaque chemin jusqu'au bout, c'est-à-dire jusqu'à tomber sur un sommet dont tous les voisins ont déjà été visités, avant de remonter le chemin pour trouver un autre chemin alternatif.

III. 1. a. Implémentation récursive

La fonction est naturelle récursive: on appelle récursivement, sur chaque successeur de v_{source}, la fonction d'exploration DFS. Pour que l'exploration se fasse toujours vers des sommets non explorés (pas de retour en arrière) et garantir ainsi la terminaison de l'algorithme, on utilise un tableau de booléens initialisé avec la valeur `false`, nommé `visited`, qui permet de marquer les sommets déjà visités en mettant la case associé à l'étiquette entière du sommet à `true`.

Attention (visité ≠ terminé ≠ sans successeur). Attention, il faut bien différencier les sommets:

- sans successeurs
- visités
- terminés

On a l'implication suivante (*terminé* ⇒ *visité*) mais *visité* n'implique certainement pas *terminé* et *terminé* n'implique certainement pas *sans successeur*.

Algorithme 1 : `dfs`

Donnée : `g`, graphe
Donnée : `source`, étiquette, $0 \leq$ `source` $\leq n_v - 1$

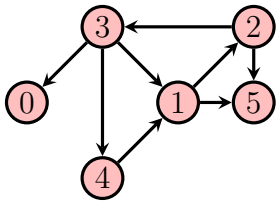
Variable de travail : `visited`, `tableau de booléens global, initialisé à false`
Variable de travail : `lsucc`, `liste locale des successeurs`

1 **Marquer la source comme ayant été visitée** `visited[source] ← true`
2 **Récupérer la liste des successeurs** du sommet d'étiquette `source` `lsucc ← succ(g, source)`
3 **Appeler récursivement et successivement l'algorithme** sur tous les éléments de `lsucc` non déjà visités
4 **Afficher** le sommet `u` comme ayant été traité *(parcours post-fixe, affichage après traitement)* `print u`

Voici une implémentation OCaml de cet algorithme:

```
let dfs_rec g source  =
   let nv = Wgraph.number_of_vertices g in
   let visited = Array.make nv false in
   let rec aux_dfs_rec g u =
     visited.(u) <- true;
     List.iter (fun (v,w) -> if not visited.(v) then
                               (aux_dfs_rec g v)
       ) (Wgraph.succ g u);
      Printf.printf "%d terminé\n" u;
   in
   aux_dfs_rec g source
;;
```

Exercice 1 (Exercice fondamental).
Dérouler l'algorithme, en suivant notamment l'état du tableau `visited` sur le graphe suivant. On supposerons que la liste des successeurs est renvoyée par ordre croissant d'étiquettes des successeurs.



Propriété 4

Un appel à la fonction `dfs_rec g source` termine et détermine exactement l'ensemble des sommets accessibles depuis le sommet étiqueté `source`, c'est-à-dire les sommets v pour lesquels il existe un chemin $v_{\texttt{source}} \to^* v$

Démonstration

Terminaison. On prend comme variant la suite (u_k) représentant le nombre de sommets non marqués lors du k-ième appel récursif. Ce nombre est strictement décroissant à chaque appel récursif k. On a donc une suite strictement décroissante à valeurs dans $\mathbb{N}$, ensemble bien fondé: cette suite ne peut être que finie, et l'algorithme termine.

Démonstration – Suite

Montrons que tous les sommets atteignables depuis $v_{\texttt{source}}$ sont marqués. Nous allons démontrer la propriété suivante par une récurrence faible sur la longueur du chemin.
$\mathcal{P}(n)$: Pour tout sommet u et tout sommet v, si il existe un chemin $u \to^* v$ de longueur n alors `dfs g u` marque le sommet v.

Initialisation. $\mathcal{P}(0)$ est vraie car, pour un sommet u, le seul chemin de longueur 0 depuis u mène à u, et `dfs g u` marque immédiatement le sommet u, d'après l'algorithme.

Hérédité. Supposons $\mathcal{P}(n-1)$ vraie. Et soit u et v deux sommets tels qu'il existe un chemin $u \to^* v$ un chemin de longueur n. On considère un appel à `dfs g u` et on veut montrer que cet appel va marquer v.

On peut décomposer ce chemin en faisant apparaître le deuxième sommet x du chemin:

$$u \to x \to^* v$$

`dfs g u` appelle récursivement `dfs g x`, à moins que cet appel n'ait déjà été effectué. Dans tous les cas, par hypothèse de récurrence, comme $x \to^* v$ est un chemin de longueur $n-1$, cet appel récursif marque le sommet v et $\mathcal{P}(n)$ est donc vraie.

Conclusion. On a donc démontré la propriété $\mathcal{P}(n)$ pour tout $n \in \mathbb{N}$.

En particulier, on a montré que pou $u = v_{\texttt{source}}$ et tout sommet v, s'il existe un chemin $v_{\texttt{source}} \to^* v$ alors `dfs g source` marque le sommet v. Autrement dit, tous les sommets atteignables depuis $v_{\texttt{source}}$ sont marqués.

Montrons que seuls les sommets atteignables depuis $v_{\texttt{source}}$ sont marqués. Nous démontrons la propriété suivante par une récurrence faible sur le nombre d'appels récursifs imbriqués
$\mathcal{P}(k)$: Pour tout sommet u et tout sommet v, si `dfs g u` marque le sommet v au k-ième appel récursif imbriqué, alors il existe un chemin $u \to^* v$ de longueur k.

Initialisation. $\mathcal{P}(0)$ est vraie car, pour deux sommets u et v, si `dfs g u` marque le sommet v avec 0 appel récursif, alors la seul possibilité, d'après l'algorithme, est que $u = v$ et il existe bien un chemin de longueur 0 de u à lui-même.

Hérédité. Supposons $\mathcal{P}(k-1)$ vraie. Et soit u et v deux sommets tels que `dfs g u` marque le sommet v au k-ième appel récursif imbriqué. D'après l'algorithme, `dfs g u` effectue au moins un appel récursif `dfs g x` qui marque le sommet v après $k-1$ appels récursifs. Par hypothèse de récurrence, il existe donc un chemin $x \to^* v$ de longueur $k-1$. Mais par ailleurs, x est forcément un successeur de u. On a donc un chemin $u \to x \to^* v$ de longueur k entre u et v et $\mathcal{P}(k)$ est donc vraie.

Conclusion. On a donc démontré la propriété $\mathcal{P}(k)$ pour tout $k \in \mathbb{N}$.

En particulier, on a montré que, pour $u = v_{\texttt{source}}$ et pour un sommet quelconque v, si `dfs g source` marque le sommet v (au bout d'un certain nombre d'appels récursifs imbriqués), alors il existe un chemin $v_{\texttt{source}} \to^* v$. Autrement dit, l'algorithme ne marque que les sommets atteignables depuis $v_{\texttt{source}}$.

III. 1. b. Complexité du parcours en profondeur récursif et version itérative.

La complexité temporelle de `dfs_rec` est en $\Theta(n_v + n_e)$: coût d'initialisation du tableau `visited` de taille n_v et parcours de chaque arête, en supposant que l'implémentation permet d'obtenir la liste des successeurs en temps constant.[1] La complexité spatiale dans le segment du tas de `dfs_rec` est en $\Theta(n_v)$, correspondant à l'allocation du tableau `visited` de taille n_v et éventuellement à la création des maillons de la liste de successeurs, qui est dans le pire des cas en $\Theta(n_v)$ pour un graphe linéaire.
La complexité spatiale de `dfs_rec` dans le segment de pile peut être très élevée si le graphe possède beaucoup de sommets et s'il existe des chemins très profonds, typiquement dans le cas d'un graphe dégénéré linéaire équivalent à une simple liste.
Pour pallier cet inconvénient, on peut coder cet algorithme de manière itérative, en remplaçant le mécanisme récursif par un pile qui accumule les étiquettes des sommets en cours d'exploration. L'idée est d'utiliser une pile qui va empiler mes étiquettes des sommets situés sur le chemin en cours d'exploration et dépiler lorsque l'on a atteint le bout d'un chemin (sommet terminé) jusqu'à avoir en tête de pile un sommet non terminé, pour lequel il reste des successeurs à explorer.
Les appels récursifs sont donc remplacés par une boucle`while` qui tourne tant qu'il y a un chemin en cours d'exploration, donc tant que la pile d'exploration est non vide.

Algorithme 2 : `dfs_iter`

Donnée **:** `g`, graphe
Donnée **:** `source`, étiquette, $0 \leq$ `source` $\leq n_v - 1$

Variable de travail **:** `visited`, `tableau de booléens initialisé à false`
Variable de travail **:** `s`, `pile d'étiquettes entières`

Empiler l'étiquette `source` **sur** `s` `push(s,source)`
Tant que *la pile* `s` *n'est pas vide* **faire**
  Récupérer l'étiquette `u` **en tête de pile** `u ←top(s)`
  Marquer le sommet correspondant comme visité `visited[u] ← true`
  Récupérer la liste des successeurs du sommet d'étiquette `u`
  `lsucc ← succ(g, u)`
  Repérer dans `lsucc` **le premier successeur** `v` **de** `u` **non déjà visité**
  Si `v` *n'existe pas* **alors**
    Dépiler `u` `pop(s)`
    Afficher le sommet `u` comme ayant été traité `print u` *(parcours post-fixe, affichage après traitement)*
  Sinon
    Empiler `v` `push(s,v)`

[1] Attention, cela est le cas pour une implémentation par liste d'adjacence, mais pas pour une implémentation par matrice d'adjacence, pour laquelle la construction de cette liste est en $\Theta(n_v)$

III. 1. c. Applications du parcours en profondeur

Application 1: repérage des composantes connexes d'un graphe non-orienté.

Le parcours en profondeur permet de calculer les composantes connexes d'un graphe non-orienté. Les composantes connexes sont indiquées à l'aide d'un simple tableau d'entiers `connex_components` de taille n_v. Tous les sommets du graphes appartenant à la même composante connexe se verront attribuer un même entier dans ce tableau. Ainsi, deux sommets d'étiquettes i et j appartiennent à la même composante connexe si ces deux indices sont associés à la même valeur (la même couleur) dans le tableau.
Il suffit de partir d'un premier sommet source et d'effectuer un parcours en profondeur, qui va permettre d'explorer tous les sommets appartenant à la même composante connxe que ce sommet de départ (cf démonstration de la proposition, ci-dessus). Ce parcours permet de colorier tous les sommets de cette première composante connexe avec la même couleur, souvent 1, en marquant tous les sommets visités lors de ce parcours avec la couleur 1 dans le tableau `connex_components` .
Lorsque tous les chemins de cette composante ont été explorés, on regarde s'il reste des sommets non explorés, c'est-à-dire non coloriés, ce qui sera le cas s'il s'agit d'un graphe non orienté non-connexe. On passe alors à la couleur 2 et on repart alors du premier sommet trouvé non encore colorié pour explorer une nouvelle composante connexe et attribuer la couleur de composante connexe 2 à tous les sommets atteints par cette seconde exploration...etc
On effectue cela jusqu'à ce que tous les sommets du graphe, sans exception, aient été visités.
On peut améliorer la complexité spatiale dans le segment de tas d'un facteur 2 utilisant le tableau `connex_components` à la fois pour le coloriage des composantes connexes mais aussi pour le marquage des sommets visités, ce qui permet de supprimer le tableau `visited`. Pour faire cela, il suffit d'adopter une convention adéquate sur la coloration des composantes connexes. On peut par exemple commencer la coloration des composantes connexes à 1 et réserver le 0 pour indiquer qu'un sommet n'a pas été visité.

Attention. Cet algorithme ne permet pas de calculer les composantes fortement connexe d'un graphe orienté.
Nous étudierons en TP/TD l'algorithme de Kosaraju-Sharir qui permet de calculer les composantes fortement connexes d'un graphe orienté. Vous verrez encore d'autres méthodes en 2ème année.

Application 2: repérage de cycles.

Le parcours en profondeur permet de repérer des cycles dans un graphe. Il faut distinguer le cas des graphes non orientés et des graphes orientés.

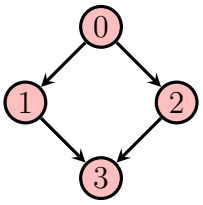
Graphes non-orientés: Le critère est simple:

> *Lors de l'exploration d'un chemin, si je retombe sur un sommet déjà visité, alors le graphe possède un cycle.*

Attention toutefois, pour les graphes non orientés, il ne faut pas compter le chemin $u \to v \to u$ comme un cycle car l'arc est traversé deux fois et un cycle est, par définition, un chemin sans répétition d'arcs. Il faut donc veiller à ce que le sommet

déjà visité sur lequel nous sommes retombés ne soit pas l'avant dernier du chemin en cours d'exploration.

Graphe orienté: c'est moins simple, car on peut avoir ce genre de situations:



Dans cet exemple, le critère utilisé pour les graphes non-orientés ne marche pas, car on peut atteindre un sommet par deux chemins différents sans pour autant avoir un cycle (le graphe ci-dessus n'a pas de cycle). Le bon critère pour les graphes orientés est le suivant:

Si, lors d'un parcours, je retombe sur
un sommet du chemin en cours d'exploration, alors le graphe possède un cycle.

Pour coder il suffit d'enrichir un peu l'algorithme DFS en rajoutant un nouvel état traduisant le fait qu'un sommet est sur un chemin en cours d'exploration. On peut le faire en créant un tableau qui numérote successivement les sommets en cours d'exploration (et les dénumérote quand on fait machine arrière). Ce tableau est initialisé à -1 pour toutes ses valeurs. On initialise un compteur de chemin à 0, puis à chaque fois qu'on l'on avance d'un arc dans le chemin, on incrémente ce compteur et on associe la valeur courante du compteur à ce nouveau sommet. Lorsque l'on arrive sur un sommet impasse, c'est-à-dire un sommet dont tous les voisins ont été visités, on retire ce sommet du chemin en remettant l'entier associé à -1. Le repérage d'un cycle se fait lors de l'analyse des successeurs du sommet courant: si l'un de ces successeurs correspond à un sommet ayant déjà été numéroté (et qui n'est pas le sommet juste avant dans le cas de graphe non-orienté), alors cela signifie que l'on est en train de recroiser le chemin précédemment parcouru: on a trouvé un cycle.
On peut aussi, dans le cas itératif, s'en sortir en créant simplement un type somme permettant de décrire les trois états possibles dans sommet (non visité, sur un chemin en cours d'exploration, et terminé) et stocker des valeurs de ce type dans le tableau `visited`.
Pour gérer le cas de la non-existence de 2-cycles dans les graphes non orientés, on peut simplement dépiler temporairement deux valeurs de la pile d'exploration pour regarder si le sommet sur lequel on est retombé n'est pas l'avant dernier du chemin d'exploration.

Application 3: ordonnancement de tâche et parcours post-fixe comme tri topologique.

Dans les deux algorithmes DFS présentés, on affiche l'étiquette d'un sommet une fois que tous les chemins partants de ce sommet ont été explorés, c'est-à-dire quand toutes les explorations partant de ce sommet sont terminées. Cet affichage est une généralisation du parcours post-fixe vu sur les arbres binaires. La seule différence réside dans le fait que, sur un graphe général, contrairement au cas des arbres binaires, l'ordre post-fixe ainsi obtenu n'est pas unique car:

- Le choix de la source est arbitraire, alors qu'il est imposé (c'est la racine) dans le cas des arbres binaires

- L'ordre exploration des successeurs dépend de la manière dont ces derniers ont été stockés, alors que pour les arbres binaires, qui sont des structures de données positionnelles, on s'est imposé comme convention de toujours visiter le successeur de gauche en premier, puis celui de droite.

Cette non-unicité était déjà apparue lors de la généralisation du parcours post-fixe aux arbres généraux enracinés (cf TP Arbres Généraux).

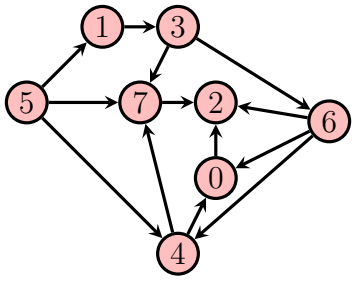
Remarque (Parcours préfixe généralisé). Pour obtenir une généralisation du parcours préfixe vu sur les arbres, il suffit de modifier les algorithmes DFS présentés en affichant l'étiquette du sommet avant de traiter la liste des successeurs.
C'est donc le parcours DFS, avec un affichage avant ou après traitement des successeurs, qui permet de généraliser les deux parcours pré et post-fixe, selon que l'on affiche le sommet courant avant ou après traitement de ses successeurs.
A noter: le parcours infixe n'est défini que pour des arbres binaires, il n'a pas de sens pour les arbres généraux, et encore moins pour les graphes.

Dans le cas de graphe orientés acyclique, l'empilement successifs des étiquettes des sommets terminés post-fixe dans une pile lors le parcours post-fixe fournit ce que l'on appelle un tri topologique. Comme on empile les sommets terminés, la pile obtenue, si on la lit du haut de pile vers le fond de pile, correspond à un **un affichage renversé** par rapport à l'affichage post-fixe direct des algorithmes DFS présentés ci-dessus.

Définition 15 (Tri topologique d'un graphe orienté)

On appelle **tri topologique d'un graphe orienté** une liste ordonnée de ses sommets telle que, pour tout arc $(u, v) \in E$, le sommet u apparaît avant le sommet v dans la liste.
On peut obtenir un tel tri par empilement successifs des étiquettes des sommets terminés dans une pile lors le parcours post-fixe.

Un exemple d'application très utile du tri topologique est celui de l'**ordonnancement de tâches** dépendant les unes des autres. On représente l'ensemble des tâches comme les sommets d'un graphe et on indique qu'une tâche u doit être effectuée avant une tâche v en créant un arc orienté $u \to v$. Voici un exemple de graphe de tâches.



Un tri topologique d'un tel graphe fournit une manière cohérente de réaliser les tâches, en respectant les contraintes liées à leur enchaînement de sorte que lorsque l'on effectue les tâches dans l'ordre du tri topologique, au moment d'effectuer une tâche, toutes les tâches nécessaires ont été effectuées en amont.

Remarques.

- Un trio topologique n'a de sens que sur un graphe orienté acyclique.
- Il n'y a pas unicité du tri topologique: par exemple ici, $[5, 1, 3, 6, 4, 7, 0, 2]$ et $[5, 1, 3, 6, 4, 0, 7, 2]$ sont deux tri topologiques différents de ce graphe.
- Si le graphe possède un cycle, alors il ne peut pas exister de tri topologique, car il existe alors au moins un sommet qui est à la fois avant et après un autre sommet.

Propriété 5 (Ordre post-fixe et tri topologique)

Pour un graphe orienté acyclique, la pile obtenue en empilant les sommets terminés au cours de l'algorithme DFS (pile lue du sommet de la pile, dernier sommet terminé, vers le fond de la pile) est un tri topologique.

Démonstration

Soit un arc $u \to v$ dans un graphe orienté g. On veut montrer que u apparaît avant v dans la liste postfixe, autrement dit que `dfs g u` termine après `dfs g v`.

- Si, c'est l'appel `dfs g u`, qui déclenche `dfs g v`, alors `dfs g v` terminera en premier et v apparaîtra après u dans la liste.
- Si c'est l'appel `dfs g v` qui déclenche `dfs g u`, alors il existe une chemin $v \to^* u$, mais comme $u \to v$, cela signifierait qu'il existe un cycle, ce qui contredit l'hypothèse du graphe orienté acyclique. Ce cas ne peut donc pas arriver.
- Si, ce n'est pas l'appel `dfs g u` qui déclenche `dfs g v`, c'est que le sommet v a déjà été marqué, donc que `dfs g v` a été déclenché par un autre appel précédemment effectué. Cet appel s'est de toute de toute façon terminé avant l'appel à `dfs g u`: v est donc déjà dans la liste et u apparaîtra donc après, une fois l'appel `dfs g u` terminé.

III. 2. Parcours en largeur (breadth-first search BFS)

III. 2. a. Principe

Le parcours en largeur d'un graphe à partir d'un sommet source v_{source} consiste à explorer les sommets d'un graphe en progressant en cercles concentriques, en "tâche d'huile" à partir de la source. On part de la source puis on visite d'abord les sommets à distance d'un arc, puis ceux à distance deux arcs...etc
L'algorithme de parcours en largeur se prête naturellement à une implémentation itérative. On utilise une file pour traiter les sommets dans l'ordre d'élargissement du périmètre d'exploration autour de la source.

Algorithme 3 : `bfs`

Donnée **:** `g`, graphe
Donnée **:** `source`, étiquette, $0 \leq$ `source` $\leq n_v - 1$

Variable de travail **:** `visited`, `tableau de booléens initialisé à false`
Variable de travail **:** `q`, `file d'étiquettes entières`

Initialiser la file de travail `q` avec une file vide — `q ← queue_create()`
Enfiler l'étiquette `source` dans la file `q`
1 — `enqueue(q,source)`
2 **Marquer** le sommet `source` comme visité — `visited[source] ← true`
3 **Tant que** *la file* `q` *n'est pas vide* **faire**
4 **Récupérer et supprimer l'étiquette** `u` **en tête de file** `q` — `u←dequeue(q)`
5 **Afficher** le sommet `u` comme ayant été traité — `print u`
6 **Récupérer la liste des successeurs** du sommet d'étiquette `u`
7 `lsucc ← succ(g, u)`
8 **Pour** *Toutes les étiquettes* `v` *de sommets de* `lsucc`*, successeurs de* `u` **faire**
9 **Si** `v` *n'a pas encore été visité* **alors**
10 **Ajouter** `v` dans à la file `q` — `enqueue(q, v)`
11 **Marquer** le sommet d'étiquette v comme visité — `visited[v] ← true`

La dénomination anglaise, *Breadth-First Search* (BFS) vient de l'anglais *breadth*, la fratrie. Cette image correspond bien au fat que l'on va d'abord visiter les frères avant de s'enfoncer en profondeur dans le graphe. [2]
Les fichiers donnés en annexe de ce cours (dans le dossier *Exemples de déroulés - BFS*) montrent des exemples où l'on déroule pas à pas l'algorithme de recherche en largeur. Allez les voir et assurez vous de bien comprendre le déroulement de l'algorithme.

[2] *breadth* = substantif du verbe *to breed*, éléver des enfants, des animaux. breadth = ceux qui ont été élevés ensemble, la fratrie.

III. 2. b. Correction

Nous allons démontrer la propriété suivante:

Propriété 6

L'algorithme de parcours en largeur `bfs g source` marque exactement tous les sommets de g accessibles depuis le sommet d'étiquette `source`, c'est à dire qu'il marque tous les sommets v tels que $v_{\text{source}} \rightarrow^* v$.

Démonstration

Nous allons faire une preuve par invariant de boucle. Il est très facile de montrer, en s'appuyant sur l'algorithme, l'invariant de boucle suivant, où k correspond au numéro d'itération de la boucle `while`:

$$\mathcal{P}(k) : \forall v_j \in V(\exists\, v_{\texttt{source}} \rightarrow^* v_j \text{ de longueur } k \Leftrightarrow j \text{ est dans la file } \texttt{q})$$

Pour conclure sur la correction de l'algorithme, il suffit de remarquer que l'algorithme se termine quand la file est vide. Or, chaque fois qu'on le retire la tête de file, on marque le sommet correspondant, donc tous les sommets qui se sont trouvé dans la file à un moment donné ont bien été marqués. D'après l'invariant de boucle démontré, ce sont exactement les sommets accessibles depuis $v_{\texttt{source}}$ qui ont été marqués, et eux seuls.

III. 2. c. Analyse de complexité

Pour minimiser la complexité temporelle de notre algorithme, on fera les choix suivants:

Choix d'implémentation pour les files. On utilisera des implémentations performantes de files, par exemple avec la stratégie *ring buffer*, qui nous garantissent des opérations de mise en queue de file et de récupération de tête de file en $\Theta(1)$. Cette implémentation ne pose aucun problème ici car on peut borner la taille de la file par n_v. En effet, grâce au tableau `visited`, on est certains qu'un sommet ne pourra entrer qu'une seule fois dans la file. Dans le pire des cas, tous les n_v sommets sont dans la file au même moment (essayez de trouver un graphe où cela arrive!) . Il nous suffit donc de prescrire une capacité maximale de file à n_v et l'on est certain de ne jamais la dépasser.

Choix d'implémentation pour les graphes. On utilise également une implémentation par liste d'adjacence pour que le coût de récupération de la liste des successeurs d'un sommet, qui a lieu à chaque tour de boucle, se fasse en temps constant. Il s'agit en effet d'une simple récupération d'un pointeur, alors qu'il faut parcourir toute une ligne de la matrice en $\Theta(n_v)$ dans le cas d'une implémentation par matrice d'adjacence

Avec ces choix, l'analyse de la complexité de l'algorithme donne ceci:

Algorithme 4 : `bfs`

Donnée : `g`, graphe (orienté ou non)
Donnée : `source`, étiquette, $0 \leq$ `source` $\leq n_v - 1$

Variable de travail : `visited`
Variable de travail : `q`

```
1  Initialisation de visited à false                                   Θ(n_v)
2  q ← queue_create()                                                  Θ(1)
3  enqueue(q,source)                                                   Θ(1)
4  visited[source] ← true                                              Θ(1)
5  Tant que is_empty(q) est faux faire
6      u←dequeue(q)                                                    Θ(1)
7      print u                                                         Θ(1)
8      lsucc ← succ(g, u)        Θ(1) avec l'implémentation par listes d'adjacence
9      Pour Toutes les étiquettes v de sommets de lsucc, successeurs de u   Θ(δ+(u))
        faire
10         Si visited[v] contient false alors
11             enqueue(q, v)                                           Θ(1)
12             visited[u] ← true                                       Θ(1)
```

Au pire, on a une boucle sur tous les successeurs du sommet courant u en $\Theta(\delta_+(u))$, donc la complexité de la boucle `while` est au pire en $O(n_e)$. Il faut rajouter à cela l'initialisation du tableau `visited`, en $\Theta(n_v)$: la complexité temporelle est donc $C_{temp}(\texttt{bfs}) \in O(n_e+n_v)$.

La complexité spatiale dans le segment du tas est en $\Theta(n_v)$ puisque l'on alloue le tableau `visited`, de taille n_v, et une file `q` implémentée par *ring buffer* avec une capacité maximale n_v.

La complexité spatiale dans le segment de pile est en $\Theta(1)$ car l'algorithme est purement itératif.

III. 2. d. Variantes d'implémentation

Il existe plusieurs variantes d'implémentation de cet algorithme de parcours en largeur. On peut notamment remarquer qu'il n'est pas vraiment nécessaire d'utiliser une pile. La seule vraie contrainte étant de traiter les sommets à distance k arcs avant les sommets situés à distance $k+1$ arcs. Pour cela, on peut procéder avec deux piles: l'une `s_cur` pour les successeurs à distance k, et l'autre `s_nxt` pour les successeurs à distance $k+1$. Quand la pile `s_cur` est vide, on copie `s_nxt` dans `s_cur` et on vide `s_nxt`. L'algorithme se termine quand les deux piles sont vides.

IV. Calcul de plus court chemin dans un graphe pondéré

Les deux algorithmes ci-dessous s'appliquent à des graphes simples (sans boucles ni multi-arêtes) pondérés, orientés ou non. **Les pondérations des arcs sont des nombres réels et sont supposés être positifs ou nulles**. Ces pondérations sur les arcs sont interprétées comme des distances entre les sommets du graphe. Pour une arc $(u, v) \in E$, la pondération sera donc notée $d(u, v)$ au lieu de $w(u, v)$ et on fait l'hypothèse suivante:

$$\forall (u, v) \in E \, d(u, v) \in \mathbb{R}^+$$

On définit alors la distance entre deux sommets de la manière suivante:

Définition 16 (Distance entre deux sommets)

Soit u et v deux sommets d'un graphe $g = (V, E)$. S'il existe un chemin $u \to^* v$ entre u et v de la forme:

$$u = x_0 \to x_1 \to \ldots \to x_{p-1} \to x_p = v$$

alors la distance entre u et v est définie par:

$$\text{dist}\,(u, v) = \sum_{i=0}^{p-1} d(x_i, x_{i+1}) \geq 0$$

S'il n'existe pas de chemin entre u et v, cette distance est considérée comme étant infinie:

$$\text{dist}\,(u, v) = +\infty$$

Enfin, on considérera que la relation d'adjacence est réflexive et qu'un sommet est toujours à distance 0 de lui-même:

$$\text{dist}\,(u, u) = 0$$

Pour tout sommet u, on considérera donc qu'il existe un chemin $u \to u$ associée à la distance 0., donc un plus court chemin.

IV. 1. Algorithme de Dijkstra de calcul de plus court chemin dans un graphe pondéré

IV. 1. a. Présentation de l'algorithme

L'algorithme de Dijkstra s'appuie sur un parcours en largeur (BFS) pour tenter de trouver le chemin le plus court depuis une source unique, vers tous les autres sommets du graphe. Cet algorithme prend en entrée un graphe `g` et l'étiquette `source` du sommet choisi comme sommet de départ, et calcule n_v plus courts chemins vers l'ensemble des sommets du graphe.
L'algorithme fonctionne avec une **file de priorité**. Cette file contient des couples clé-valeur (d, v): la valeur v correspond à l'étiquette entière d'un sommet, la clé k, qui sert déterminer la priorité de cet élément par rapport aux autres éléments de la file, correspond à la distance la plus courte actuellement calculée entre le sommet d'étiquette v et la source.

Un tableau de flottants de taille n_v, `dist`, est mis à jour de sorte que `dist[v]` contient, à un moment donné de l'algorithme, la plus courte distance calculée entre la source et le sommet d'étiquette `v` à ce stade de l'exploration en largeur.
Comme on peut atteindre un sommet par plusieurs chemins, un test est effectué lors de l'arrivée sur un sommet v par un nouveau chemin en venant de u. La distance minimal entre la source et v en passant par u est égale à dist[u] + d(u, v). On doit donc comparer cette nouvelle distance à la longueur du plus court chemin actuellement connue dist[v] entre la source et v. Si cette nouvelle distance est inférieure, alors le nouveau chemin passant par u est plus court: on met donc à jour la distance du plus court chemin connu entre la source et v.

Algorithme 5 : `dijkstra`

Donnée **:** `g`, graphe (orienté ou non) pondéré positivement
Donnée **:** `source`, étiquette, $0 \leq$ `source` $\leq n_v - 1$

Variable de travail **:** `dist_min`, tableau de flottants de taille n_v, résultat de l'algo
1 ¨
Variable de travail **:** `q`, file de priorité de couples (clé = distance à la source, valeur = étiquette du sommet)

2 Allocation de `dist_min` + initialisation avec des distances infinies `DBL_MAX` $\Theta(n_v)$
3 `dist_min[source]` $\leftarrow 0.$
4 `q` $\leftarrow$ `pqueue_create()` $\Theta(1)$
5 `enqueue(q, 0., source)` $\Theta(1)$
6 **Tant que** *`is_empty(q)` est faux* **faire**
7 (`dist_min[u]`, `u`) $\leftarrow$ `dequeue(q)` (récupère l'élément de priorité max, donc ici à distance minimale de la source) $\Theta(1)$
8 `lsucc` $\leftarrow$ `succ(g, u)` $\Theta(1)$ avec l'implémentation par listes d'adjacence
9 **Pour** *tous les arcs* $(d(u, v), v)$ *de* `lsucc` $\Theta(\delta_+(u))$ **faire**
10 **Si** *`dist_min[u]`* + $d(u, v) <$ *`dist[v]`* **alors**
11 `dist_min[v]` $\leftarrow$ `dist_min[u]` + d(u, v) $\Theta(1)$
12 `enqueue(q, dist_min[v], v)` $O(\log(|q|))$
13 Renvoyer le tableau `dist_min`

Les fichiers donnés en annexe de ce cours (dans le dossier *Exemples de déroulés - Dijkstra*) montrent des exemples où l'on déroule pas à pas l'algorithme de Dijkstra. Allez les voir et assurez vous de bien comprendre le déroulement de l'algorithme.

Remarque. On note un petit défaut d'implémentation: la distance minimale courante entre un sommet et la source peut être stockée en doublon: une fois dans le tableau `dist`, et une fois dans la file comme clé associée à cette étiquette... Dans l'idéal, il faudrait que le tableau de clés de la file de priorité soit directement connecté au tableau `dist`... mais c'est compliqué!

IV. 1. b. Reconstitution des plus courts chemins.

Sous cette forme, l'algorithme renvoie seulement les plus courtes distances entre la source et tous les sommets du graphe, mais ne donne pas les suites de sommets correspondant à ces plus courts chemins. Pour pourvoir reconstituer ces plus courts chemins, il suffit de mettre à jour un tableau `previous_vertex` de taille n_v, de sorte que `previous_vertex[v]` contient toujours le prédécesseur de v situé sur le plus court chemin qui a permis d'arriver jusqu'à v, et -1 si v n'a pas encore été visité, ou s'il est inaccessible depuis la source. On renvoyant ce tableau en plus du tableau `dist_min`, il est alors possible de reconstituer le plus court chemin en partant de la destination ciblée v et en reconstituant le chemin à l'envers: on retrouve en effet à chaque fois le prédécesseur d'un sommet sur le chemin le plus en sautant de case en case dans le tableau `previous_vertex` jusqu'à arriver au sommet source si le chemin existe.

IV. 1. c. Preuve de correction totale

Terminaison. La preuve de terminaison est un peu plus complexe que celle de BFS. En effet, dans l'algorithme de Dijkstra, il est possible, qu'un sommet soit présent plusieurs fois dans la file de priorité, avec des distances minimales associées (clés) différentes (voir l'exemple déroulé ci-dessus). Cependant, une fois que l'on a atteint un sommet v par son plus court chemin depuis la source, il entra dans la file avec le couple $(d_{min}(v_{source}, u), u)$. Ce couple sera forcément extrait avant tout autre couple $(d(v_{source}, u), u)$ qui pourrait déjà se trouver dans la pile car $d_{min}(v_{source}, u) \leq d(u_{source}, v)$ par définition. Les distances minimales vers les successeurs v à partir de ce sommet seront donc éventuellement mises à jour avec la distance $d_{min}(v_{source}, u) + d(u, v)$. Ainsi, lorsque un éventuel $(d(v_{source}, u), u)$ anciennement présent et associé au même sommet u mais avec une autre distance sera dépilé, aucun successeur ne sera mis à jour car $d_{min}(v_{source}, u) + d(u, v) < d(v_{source}, u) + d(u, v)$. Ainsi, pour chaque sommet v_j , il existe une itération k_j de l'algorithme (celle où on a atteint ce sommet par le plus court chemin), à partir de laquelle plus aucun successeur de ce sommet ne pourra être empilé à partir de ce sommet. A l'itération $k = \max(k_0, \ldots, k_{n_v-1})$, plus aucun sommet ne peut amener un nouvel empilement d'un successeur, et à partir de cette itération, la file ne peut plus grossir. Comme on continue à retirer l'élément prioritaire de la file, la file va se vider jusqu'à ce que la condition d'arrêt de la boucle `while` soit atteinte, et l'algorithme termine.

Correction. Comme il s'agit d'un algorithme itératif, la preuve se fait là encore en exhibant des invariants de boucle.

IV. 1. d. Analyse de complexité

Complexité temporelle. Les complexités temporelles des différentes instructions effectuées dans l'algorithme sont fournies en annotation du précédent algorithme. La file contient au pire n_e éléments car l'algorithme visite chaque arc au plus une fois, le pire cas se produisant pour un graphe complet. En considérant une implémentation de file de priorité par tas, le coût de `dequeue` est constant (racine du tas...) et le coût d'insertion est au pire logarithmique en la taille de la file de priorité, donc en $\Theta(\log|q|)$. On a donc un coût au pire (majoration grossière), pour chaque tour de boucle, en $O(\log(n_e))$, c'est-à-dire en $O(\log(n_v))$ car on a toujours $n_e \leq n_v^2$. Au final, la complexité temporelle au pire est en $O(n_e \log(n_v))$, pour ces choix d'implémentation[3]

[3]On peut faire un peu mieux avec des tas de Fibonacci... mais c'est une autre histoire.

Complexité spatiale. La complexité spatiale dans le segment du tas est, comme pour l'algorithme BFS, en $\Theta(n_v)$.
La complexité spatiale dans le segment de pile est en $\Theta(1)$ car l'algorithme est purement itératif.

IV. 2. Algorithme de Floyd-Warshall de calcul de plus court chemin dans un graphe pondéré

IV. 2. a. Présentation de l'algorithme

L'algorithme de Floyd-Warshall s'appuie sur une stratégie d'énumération des sommets intermédiaires - on notera k l'étiquette d'un sommet intermédiaire - susceptibles de constituer une étape entre deux sommets u et v qui raccourcit le plus court chemin précédemment trouvé.
A la différence de l'algorithme de Dijkstra, l'algorithme de Floyd-Warshall prend en entrée le graphe considéré et calcule tous les plus courts chemins vers l'ensemble des sommets pour toutes les sources possibles. Il calcule donc $(n_v)^2$ plus courts chemins.
Autre différence avec l'algorithme de Dijkstra, l'implémentation de graphe par matrice d'adjacence est beaucoup plus adaptée pour cet algorithme car la principale opération est une opération de recherche d'un arc (u, v), qui s'effectue en $\Theta(1)$ dans le cas des matrices d'adjacence (simple lecture d'une case dans la matrice), alors qu'elle s'effectue en $\delta_+(u)$ avec les listes d'adjacence.

Algorithme 6 : `floyd-warshall`

Donnée **:** g, graphe (orienté ou non) pondéré positivement

Variable de travail **:** `dist_min`, matrice de flottants de taille $n_v \times n_v$, résultat
Variable de travail **:** d, distance, flottant

```
1  Allocation de la matrice dist_min
2  Copie de la matrice d'adjacence de g dans dist_min                    Θ(n_v^2)
3  Boucle triplement imbriquée en Θ(n_v^3)
4  Pour k étiquette de sommet intermédiaire allant de 1 à n_v faire
5      Pour i étiquette de sommet allant de 1 à n_v faire
6          Pour j étiquette de sommet allant de 1 à n_v faire
7              d ← dist_min[i][k] + dist_min[k][j]
8              Si d < dist[i][j] alors
9                  dist_min[i][j] ← d
10 Renvoyer la matrice dist_min
```

L'algorithme de Floyd-Warshall est naturellement itératif et réclame naturellement une implémentation concrète de graphe par matrice d'adjacence, qui permet de rendre le coût d'initialisation de `dist_min` en $\Theta(n_v^2)$, alors que cette initialisation serait en $O(n_v^2 + n_e)$ avec une implémentation par liste d'adjacence.
Là où l'algorithme de Dijkstra fonctionne en parcourant des chemins, et donc en découvrant des sommets, l'algorithme de Floyd-Warshall, lui, tire des arcs. Le déroulé de l'algorithme joint à ce cours illustre cette interprétation de l'algorithme.

IV. 2. b. Preuve de correction totale

La terminaison de l'algorithme est évidente, car il n'y a que des boucles `for`.

Pour la preuve de correction, comme il s'agit d'un algorithme itératif, nous allons utiliser un invariant sur la boucle externe sur `k`:
$\mathcal{P}(k)$: Pour tout graphe $g = (V, E)$ pondéré positivement, et pour tout couple de sommet $(u, v) \in V^2$, la valeur `dist_min[u][v]` à la fin du k-ième tour de boucle contient la distance du plus court chemin $u \to^* v$ n'empruntant que des sommets intermédiaires dans $V_k = \{v_0, \dots v_{k-1}\}$, s'il existe, et $+\infty$ s'il n'existe pas.

Démonstration

Au début du tout premier tour de boucle $k = 0$: comme $V_0 = \emptyset$, on regarde les plus courts chemins n'empruntant aucun sommet intermédiaire, c'est-à-dire... les chemins réduits à un arc existant dans le graphe. Or, l'algorithme initialise la matrice `dist_min` en copiant les valeurs de la matrice d'adjacence du graphe Donc, pour tout couple $(v_i, v_j) \in V^2$, `dist_min[i][j]` contient bien la distance du plus court chemin n'empruntant aucun sommet intermédiaire (c'est-à-dire les distances associées à des arcs existants d'arc) et $+\infty$ sinon. Donc $\mathcal{P}(0)$ est vraie.

Préservation de l'invariant. Soit g un graphe pondéré positivement. On suppose la propriété $\mathcal{P}(k)$ vraie à la fin de l'itération k (et donc au tout début de l'itération $k+1$) et on souhaite montrer que $\mathcal{P}(k+1)$ est vraie à la fin de l'itération $k+1$.
Soit $(v_i, v_j) \in V^2$ quelconque. On veut montrer que, à la fin de l'itération $k+1$, `dist_min[i][j]` contient la distance du plus court chemin entre v_i et v_j ne faisant intervenir que les $k+1$ premiers sommets du graphe $V_{k+1} = \{v_0, \dots v_k\}$, s'il existe, et $+\infty$ sinon.

- S'il existe au moins un chemin entre v_i et v_j ne faisant intervenir que les $k+1$ premiers sommets de graphe $V_{k+1} = \{v_0, \dots v_k\}$, on considère le plus court parmi ces chemins. Deux cas sont possibles:
  - Si ce plus court chemin ne fait pas intervenir le sommet v_k, alors,par hypothèse de récurrence, `dist_min[i][j]` contient déjà le plus court chemin ne faisant intervenir que les $k+1$ premiers sommets du graphe. Le passage par v_k augmente alors la longueur du chemin et donc le test `if` est faux. Ainsi, `dist[i][j]` n'est pas modifiée par l'algorithme et la distance reste bien la distance du plus chemin ne faisant intervenir que les $k+1$ premiers sommets de graphe .
  - Sinon, v_k apparaît comme sommet intermédiaire dans le plus court chemin entre v_i et v_j ne faisant intervenir que les $k+1$ premiers sommets du graphe, et ce plus court chemin s'écrit donc:

    $$v_i \to x_1 \to \dots x_p \to v_k \to y_1 \to \dots \to v_j$$

    Ce chemin le plus court est sans cycle car le graphe est pondéré positivement, et tout cycle va augmenter la longueur du chemin. Ainsi, v_k n'apparaît qu'une seule fois dans ce chemin. Dans ce plus court chemin, tous les sommets autres que v_k ont donc des étiquettes strictement inférieures à k.
    Par définition de la longueur d'un plus court chemin, la longueur de ce chemin est $d^{\min}(v_i, v_k) + d^{\min}(v_k, v_j)$. Or, par l'invariant de boucle dist_min[i][k] contient déjà la valeur $d^{\min}(v_i, v_k)$ et dist_min[k][j] contient la valeur $d^{\min}(v_k, v_j)$. Dans l'algorithme, la condition du `if` est validée et l'algorithme met bien à jour la distance dist_min[i][j] avec la distance du plus court chemin entre v_i et v_j ne faisant intervenir que les $k+1$ premiers sommets.
- S'il n'existe pas du tout de chemin entre u et v ne faisant intervenir que les sommets de $V_{k+1} = \{v_0, \dots v_k\}$, alors il n'en existait pas non plus ne faisant intervenir que les sommets de $V_k = \{v_0, \dots v_{k-1}\}$. L'algorithme ne modifie pas la valeur de dist_min[i][j] qui était et reste à $+\infty$.

Dans tous les cas, on a montré que $\mathcal{P}(k+1)$ est vraie à la fin de l'itération

IV. 2. c. Reconstitution des plus courts chemins

On peut reconstituer tous les plus courts chemins entre deux sommets u et v quelconques stockant, pour tout couple de sommets v_i et v_j l'étiquette k du dernier (le plus récent) sommet intermédiaire qui a permis de raccourcir la distance entre v_i et v_j.
On peut ensuite reconstruire le plus court chemin en redécouvrant récursivement tous les sommets intermédiaires empruntés, jusqu'à ce qu'il n'y ait plus de sommet intermédiaire, et que l'on tombe sur un arc initial du graphe.
Si v_k est le sommet intermédiaire le plus récent entre v_i et v_j:

$$v_i \rightarrow^* v_k \rightarrow^* v_j$$

- On redécoupe récursivement le chemin $v_i \rightarrow^* v_k$ en appelant la fonction de création de chemin le plus court entre v_i et v_k... qui va utiliser le sommet intermédiaire k' entre v_i et v_k ...etc
- On redécoupe récursivement le chemin $v_k \rightarrow^* v_j$ en appelant la fonction de création de chemin le plus court entre v_k et v_j... qui va utiliser le sommet intermédiaire k'' entre v_k et v_j ...etc

On peut assez facilement transformer cette fonction récursive non terminale en fonction itérative à l'aide d'une pile, pour éviter l'empilement de blocs d'activation et l'augmentation de la complexité spatiale dans le segment de pile qui en résulte.
Cette implémentation est proposée en TP.

IV. 2. d. Analyse de complexité

Complexité temporelle. La complexité temporelle de l'algorithme est en $\Theta(n_v^3)$ si l'on utilise une implémentation par matrice d'adjacence, pour laquelle la récupération du poids associé à une arête donnée se fait $\Theta(1)$ dans l'initialisation de `dist_min`. Cette complexité peut sembler effrayante... Mais n'oublions pas que l'algorithme de Floyd-Warshall calcule plus de choses que `Dijkstra` (tous les chemins les plus courts, depuis toutes les sources possibles vers toutes les destinations possibles), on ne peut donc pas totalement le comparer à Dijkstra.
Cet algorithme est parfois à privilégier dans le cas:

- où l'on manipule des graphes de taille raisonnable (< 10000 sommets sur une architecture moderne, pour que les matrices passent en RAM)
- où l'on manipule des graphes pleins, c'est-à-dire ayant beaucoup d'arcs: dans ce cas, l'implémentation par listes d'adjacence devient aussi coûteuse spatialement que celle par matrice d'adjacence, et la complexité temporelle de Dijkstra, en $O(n_e \log(n_v))$ peut se rapprocher de celle de Floyd-Warshall en $O(n_v^3)$ (comme $n_e \leq n_v^2$ Floyd Warshall restera toujours plus coûteux asymptotiquement)
- où l'on a besoin de **tous** les plus courts chemins entre deux sommets quelconques. Dans ce cas, on devrait faire n_v appels à Dijkstra avec les n_v sommets comme source. On a alors une complexité temporelle en $O(n_v \times n_e \times \log(n_v))$. Sur des graphes pleins, pour lesquels $n_e \in \Theta(n_v^2)$, on peut dépasser la complexité de Floyd-Warshall...

Dans le cas où l'on souhaite obtenir tous les plus courts chemins, que le graphe est plein et de taille raisonnable, tenant en RAM, il faut penser à Floyd-Warshall, qui a de plus le mérite d'une très grande simplicité d'implémentation.

Complexité spatiale. La complexité spatiale dans le segment de tas est en $\Theta(n_v^2)$ car il faut allouer une matrice `dist` de taille $n_v \times n_v$, et éventuellement une autre matrice d'entiers de même tailles pour `interm` si l'on souhaite reconstruire les chemins intermédiaires.
La complexité spatiale dans le segment de pile est en $\Theta(1)$ car l'algorithme est purement itératif.

IV. 3. L'an prochain...

L'an prochain, vous verrez deux algorithmes importants sur le même thème:

L'algorithme A* (*A-star*): il s'agit d'une adaptation de Dijkstra dans le cas où où l'on n'a pas besoin d'avoir tous les plus courts chemins depuis la source vers tous les sommets, mais seulement vers un autre sommet. Pour cibler uniquement ce chemin, on introduit une pénalisation sur certains chemins (ceux qui ne semblent pas aller dans la direction de la source essentiellement) grâce à des heuristiques bien choisies

L'algorithme de Kruskal: il s'agit de trouver un arbre couvrant de poids minimal.