Les Graphes

Chapitre 2

1 Les graphes . . . 1
1.1 Graphes orientés . . . 1
1.2 Graphes non orientés . . . 3
2 Implémentations . . . 5
2.1 Par matrices d'adjacence . . . 5
2.2 Par listes d'adjacences . . . 7
2.3 Passage de l'une des implémentation à l'autre . . . 9
2.4 Occupation mémoire . . . 10
3 Parcours d'un graphe . . . 11
3.1 Généralités . . . 11
3.2 Parcours en profondeur (Depth First Search) . . . 14
3.3 Parcours en largeur (Breadth First Search) . . . 16
3.4 Quelques exemples . . . 18
4 Chemin de plus faible poids dans un graphe pondéré . . . 21
4.1 Graphes pondérés . . . 21
4.2 Recherche d'un chemin de plus faible poids . . . 22
4.3 Algorithme de Floyd-Warshall . . . 23
4.4 Algorithme de Dijkstra . . . 26

Nous allons étudier les graphes

1 Les graphes

1.1 Graphes orientés

Définition 1.1.1

Un graphe (orienté) G est la donnée d'un ensemble fini non vide S et d'une partie de A de $S \times S$.

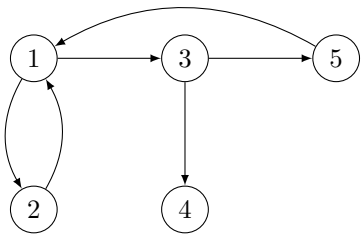
- *Les éléments de S s'appellent les sommets ou noeuds*
- *Les éléments de A s'appellent les arêtes ou arcs*
- *De manière usuelle on note les graphes (S, A) ou (V, E) où V désigne les sommets (vertex en anglais) et E désigne les arêtes (edges en anglais).*

Remarques :

1. Pour les problèmes de complexité on s'intéressera au nombre de sommets $|S|$ et au nombres d'arêtes $|A|$.

Exemples :

1. Le graphe suivant correspond à $S = \{1, 2, 3, 4, 5\}$ et $A = \{(1, 2); (1, 3); (2, 1); (3, 4); (3, 5); (5, 1)\}$



2. Les arbres sont des cas particuliers de graphes (voir plus loin)

3. Un graphe peut être utilisé pour modéliser un plan de route / de métro.
4. Le World Wide Web peut être modélisé par un graphe. Les sommets sont les pages web et les arêtes les liens hypertextes

Définition 1.1.2 (Sous-graphe)

Soit $G = (S, A)$ un graphe. On appelle sous-graphe de G un graphe $G' = (S', A')$ où $S' \subset S$ et $A' \subset A \cap (S' \times S')$.

Définition 1.1.3

Soit $G = (S, A)$ un graphe.

- *Soit x un sommet du graphe, tous les sommets y tels que (x, y) soit une arête du graphe s'appellent les voisins de x. On note souvent V_x l'ensemble des voisins de x.*
- *Soit x un sommet. Son degré sortant est le nombre de ses voisins. Son degré entrant est le nombre de sommets dont x est un voisin.*

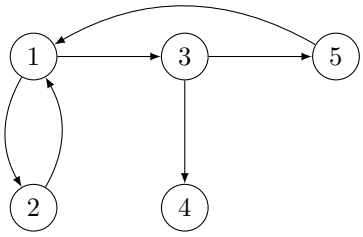
Remarque : On trouve aussi le terme **arité** pour désigner le degré.

Définition 1.1.4

Soit $G = (S, A)$ un graphe.

- *Soit x et y deux sommets du graphe. Un chemin de x à y est une suite $(x_0, \dots, x_n)$ de sommets telle que $x_0 = x$, $x_n = y$ et pour tout $k \in [\![0 ; n-1]\!]$, (x_k, x_{k+1}) soit une arête du graphe.*
- *Soit $(x_0, \dots, x_n)$ un chemin. Si $n > 0$ et $x_0 = x_n$ on dit que c'est un cycle.*
- *Soit $(x_0, \dots, x_n)$ un chemin. Il est dit que longueur n. C'est le nombre d'arêtes du chemin.*
- *Soit x et y deux sommets, la distance de x à y est la plus petite longueur d'un chemin de x à y.*

Exemple : On reprend notre graphe

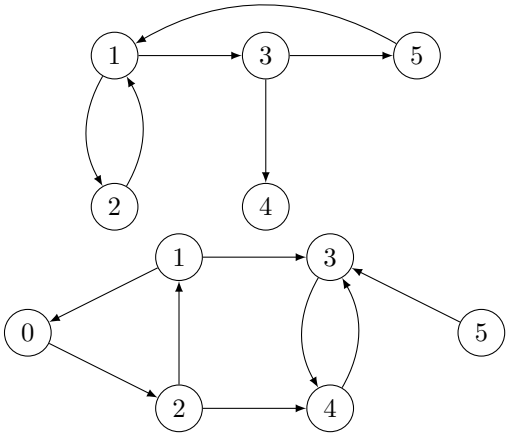


La distance de 1 à 5 est 2, par contre celle de 5 à 1 est 1.
Remarque : Un chemin a au moins une arête. Par contre, s'il existe une arête (x_0, x_0), qui « boucle » sur le sommet x_0, on peut considérer le cycle (x_0, x_0).

Définition 1.1.5

- *Un graphe est dit fortement connexe si pour tout couple (x, y) de sommets du graphe (avec $x \neq y$) il existe un chemin de x vers y **et** un chemin de y vers x.*
- *On appelle composante fortement connexe tout sous-graphe fortement connexe maximal.*

Exercice : Déterminer les composantes fortement connexes des graphes suivants



1.2 Graphes non orientés

On peut décider de ne plus orienter les arêtes du graphe.

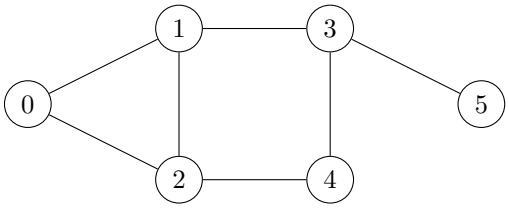
Définition 1.2.6

On appelle graphe (non orienté) la donnée d'un ensemble fini non vide S et d'une partie A de l'ensemble des parties de cardinal 2 de S (noté $\mathscr{P}_2(S)$)

Remarques :

1. Il n'est pas toujours très aisé de travailler avec des parties. On peut garder la représentation des arêtes par des couples. Il faut alors assurer que si $(x, y) \in A$ alors $(y, x) \in A$. C'est cette définition que nous prendrons.
2. Notons que cela signifie en particulier, que l'on ne considère pas de boucles sur un sommet. Contrairement au cas des graphes orientés, dans le cas d'un graphe non-orienté on ne peut pas avoir une arête qui part d'un sommet pour arriver à ce même sommet.

Exemple :



On peut reprendre la majorité des définitions dans ce cas :

Définition 1.2.7

Soit $G = (S, A)$ un graphe non orienté.

- *Soit x un sommet du graphe, un sommet y est un voisin de x si $(x, y) \in A$.*
- *Le degré (ou arité) d'un sommet est le nombre de ses voisins. Il n'y a plus lieu de séparer degré entrant et sortant.*
- *Les notions de chemins, cycles, longueur d'un chemin et distance entre deux nuds se généralisent de manière immédiate* [a]
- *Le graphe est dit connexe si pour tout couple (x, y) de sommets du graphe il existe un chemin de x vers y. (on ne parle plus de fortement connexe).*
- *On appelle composante connexe tout sous graphe connexe maximal.*

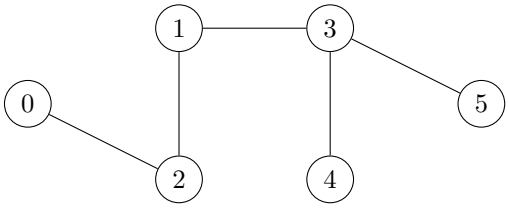
a. il faut juste ajouter qu'un chemin ne peut pas faire demi-tour, c'est-à-dire que si $(x_0, x_1, \ldots, x_n)$ est un chemin alors pour tout $i \leqslant n-2$, $x_i \neq x_{i+2}$.

Définition 1.2.8 (Arbre)

Soit $G = (S, A)$ un graphe (non orienté) non vide. On dit que c'est un arbre s'il est sans cycle et connexe.

Remarque : La définition ci-dessus est un peu différente de la définition du chapitre 1. Ici on parle d'arbres qui ne sont pas enracinés. Par contre, on peut alors choisir n'importe quel sommet x_0 pour la racine de l'arbre, on se retrouve alors avec un arbre enraciné comme au chapitre 1.

Exemple : Si on considère la graphe suivant (qui est un arbre)



En prenant 1 comme racine

En prenant 4 comme racine

2 Implémentations

Nous allons présenter les deux implémentations les plus classiques des graphes. Nous travaillerons avec des graphes orientés en précisant les modifications à apporter pour travailler avec des graphes non orientés.

2.1 Par matrices d'adjacence

On considère un graphe $G = (S, A)$. On peut le représenter par une matrice $M \in \mathscr{M}_n(\mathbf{Z})$ où $n = |S|$

Définition 2.1.9 (Matrice d'adjacence)

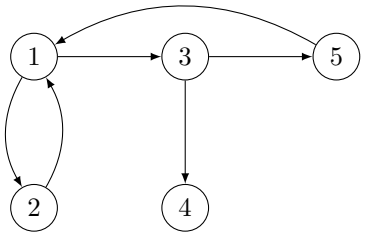
Si on note $S = \{x_1, \ldots, x_n\}$ on appelle matrice d'adjacence du graphe la matrice $M = (m_{ij})$ où

$$m_{ij} = \begin{cases} 1 & si\ (x_i, x_j) \in A \\ 0 & sinon \end{cases}$$

Remarque : Dans le cas d'un graphe orienté, les éléments diagonaux peuvent valoir 1 s'il y a une boucle sur le sommet en question. Dans le cas d'un graphe non-orienté, comme ces boucles sont interdites, les éléments diagonaux valent 0.

Exercices :

1. On reprend notre graphe exemple



Sa matrice d'adjacence est

2. Pour le graphe

Sa matrice d'adjacence est

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

Est-il fortement connexe ?

On peut donc implémenter un graphe par

ATTENTION

Dans la description mathématique on numérote les sommets de 1 à n, dans la description informatique on les numérote de 0 à $n-1$.

Exercice : Ecrire les fonctions `ajoute_arete g i j` et `supprime_arete g i j` qui ajoute ou supprime l'arête (x_i, x_j).

Les graphes non orientés sont simplement des graphes où la matrice est symétrique.

Exercice : Ecrire une fonction `symetrique : int array array-> bool` qui renvoie un booléen selon que la matrice est symétrique ou non. On essayera d'être le plus efficace possible

2.2 Par listes d'adjacences

On considère un graphe $G = (S, A)$. On peut le représenter par un tableau de listes.

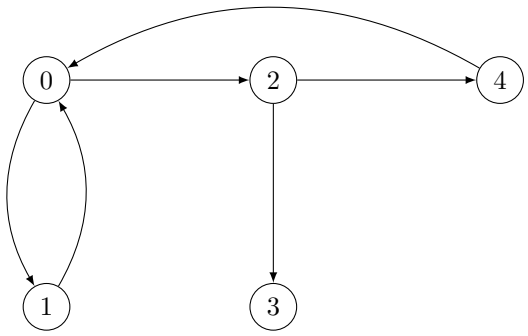
```
type graphe = int list array
```

Définition 2.2.10 (Listes d'adjacences)

Si on note $S = \{x_0, \ldots, x_{n-1}\}$ on appelle listes d'adjacence du graphe le tableau `t` *de listes où pour tout i compris entre 0 et $n-1$,* `t.(i)` *est la liste des voisins de x_i.*

Exemple :

On reprend notre graphe exemple



Ses listes d'adjacences sont

Exercice :
Déterminer le graphe donné par les listes d'adjacences

i	0	1	2	3	4	5
`t.(i)`	[2]	[0; 3]	[1; 4]	[4]	[3]	[3]

Exercice : Ecrire les fonctions `ajoute_arete g i j` et `supprime_arete g i j`

Remarque : On peut représenter un graphe non orienté en s'assurant que si i est dans `t.(j)` alors j est dans `t.(i)`.

Exercice : Ecrire une fonction `valid_non_oriente t` qui prend un graphe (donné par listes d'adjacences) et renvoie un booléen selon qu'il représente bien un graphe non orienté ou non

2.3 Passage de l'une des implémentation à l'autre

Il faut savoir écrire les fonctions `mat_to_liste` et `liste_to_tab` pour passer d'une implémentation à l'autre.

2.4 Occupation mémoire

On dispose d'un graphe avec $n = |S|$ sommets et $m = |A|$ arêtes.

— La représentation en mémoire du graphe par listes d'adjacence nécessite $|S|$ listes dont la somme des longueurs est $|A|$ (ou $2|A|$ pour un graphe non-orienté). La complexité en espace est donc

— La représentation en mémoire du graphe par matrice d'adjacence nécessite une matrice de taille $|S|^2$. La complexité en espace est donc

Dans la majorité des cas, sauf pour les graphes très denses (avec beaucoup d'arêtes), on préférera les listes d'adjacences.

3 Parcours d'un graphe

3.1 Généralités

On a vu le parcours des arbres, dans le cas des graphes c'est un peu plus compliqué car il n'y a pas de sens évident et **surtout il faut éviter de se retrouver piéger dans un éventuel cycle du graphe.**

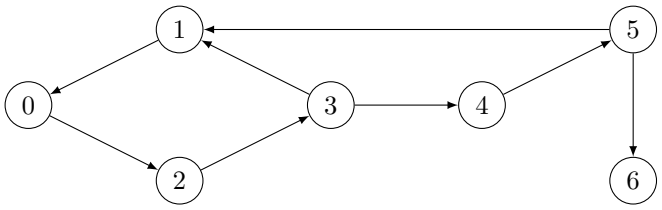
Définition 3.1.11 (Parcours d'un graphe)

Un parcours d'un graphe $G = (S, A)$ est une liste sans répétitions $(x_1, \dots x_p)$ de sommets du graphe telle que

$$\forall i \in [\![1\,;\, p-1]\!], \exists j \in [\![1\,;\, i]\!], (x_j, x_{i+1}) \in A.$$

Remarques :

1. Faire attention que lors d'un parcours on ne demande pas que (x_i, x_{i+1}) soit une arête. Par exemple pour le graphe

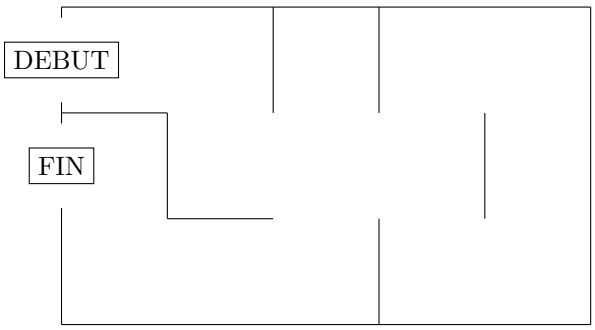


On pourra parcourir en faisant $0, 2, 3, 1, 4, 5, 6$ ou $0, 2, 3, 4, 5, 6, 1$.

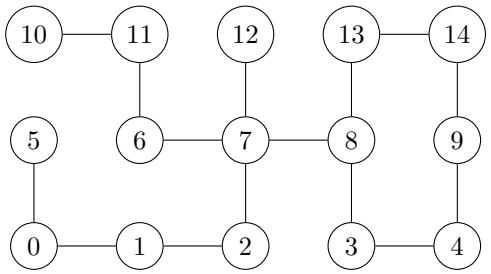
2. Il y a essentiellement deux manières de voir les parcours.
- On se donne un sommet x_1 qui sera l'origine et on parcourt le graphe tant que l'on peut ajouter des sommets.
- On se donne un sommet x_1 qui sera l'origine et un sommet x_p qui sera l'arrivée et on parcourt le graphe tant que l'on n' est pas arrivé à x_p. Il se peut que l'on ne puisse pas y arriver. On dit alors que le parcours à échoué.

3. Nous allons avoir différents parcours qui se caractérisent par l'ordre dans lequel on parcourt les sommets.

Exemple : Un exemple classique de parcours de graphe est le problème d'un labyrinthe. On considère le labyrinthe suivant :



On peut associer à ce labyrinthe un graphe (non orienté car les chemins peuvent être parcourus dans les deux sens).



Trouver une solution à ce labyrinthe, revient à déterminer un parcours du graphe qui commence au sommet 10 et qui arrive au sommet 5.

ATTENTION

Le principe général des parcours est de partager les sommets du graphe en trois.

—

—

—

On applique alors un algorithme du type : tant que la frontière n'est pas vide, on choisit un sommet de la frontière, s'il n'a pas déjà été visité, il passe dans les sommets visités et on ajoute à la frontière tous ses voisins.

En notant F les sommets de la frontière, Visit les sommets visités et, pour tout sommet x, V_x ses voisins :

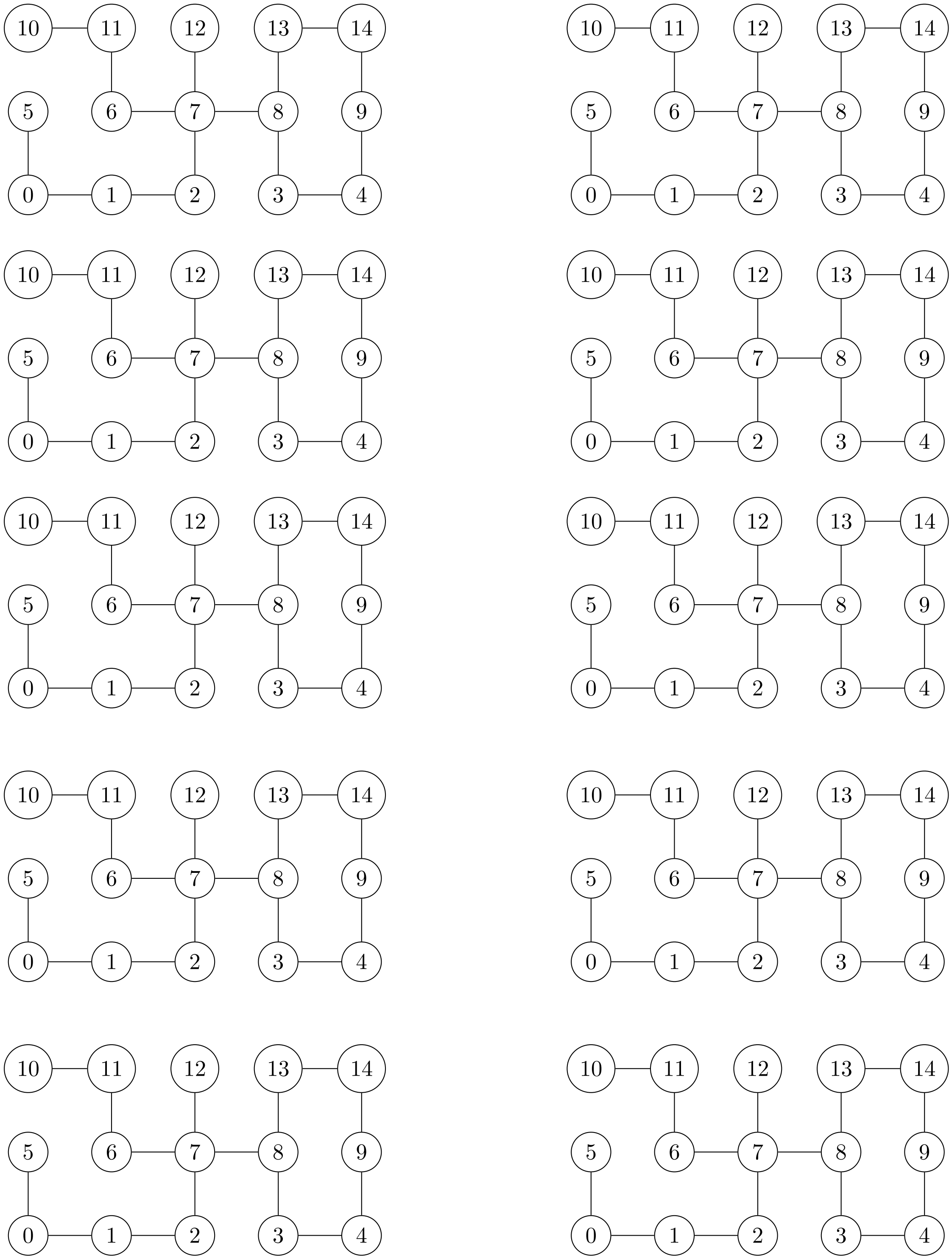
Algorithme 1 : parcours d'un graphe

Entrée Graphe $G = (S, A)$; sommet de départ s
:

```
1 F ← {s}
2 Visit ← ∅
3 tant que F ≠ ∅ faire
4 |   Choisir un sommet x dans F (et l'enlever)
5 |   si x ∉ Visit alors
6 |   |   Ajouter les sommets de V_x à F
7 |   |   Ajouter x à Visit
8 |   fin
9 fin
```

Remarque : On peut aussi se contenter d'ajouter à la frontière seulement les voisins qui n'ont pas été visités.

Exemple : Si on reprend le graphe précédent



3.2 Parcours en profondeur (Depth First Search)

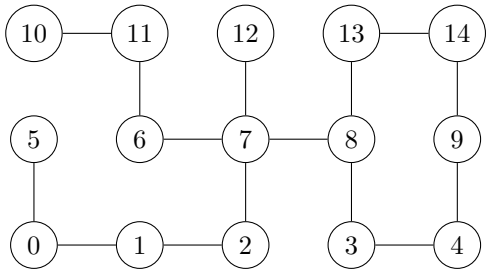
Le principe est de parcourir **en premier un voisin du sommet où l'on arrive.**
On peut l'implémenter de deux façons.

1. On peut se contenter d'une simple fonction récursive qui se rappelle sur tous les voisins non visités du sommet où l'on se situe
2. On peut réaliser ce parcours en stockant les sommets à visiter dans une pile (LIFO).

— On initialise
  — la liste des sommets déjà traités (Visit dans l'algorithme précédent) à la liste vide.
  — la **pile** des sommets à traiter (F dans l'algorithme précédent) en mettant dedans le sommet d'origine.

— Tant que la pile des sommets à traiter n'est pas vide, on traite le sommet de la pile (s'il n'a pas été déjà visité). Pour cela :
  — on l'enlève de la pile
  — on l'insère dans la liste des sommets traités
  — on ajoute tous ses voisins à la pile des sommets à traiter

Voici, dans le cas de notre exemple comment évolue la liste F et la pile Visit. Dans la colonne « sommets à traiter », on indique la pile des sommets à traiter où le sommet de la pile est à droite. Par commodité, on insérera dans la pile à chaque étape les voisins par ordre croissant



sommets traités	sommets à traiter
	10
10	11
10, 11	6

Remarques :

1. Dans un parcours en profondeur on traite en priorité les voisins du dernier sommet visité. Cela correspond au parcours en profondeur d'un arbre
2. Dans l'exemple ci-dessus on cherche à parcourir tout le graphe, comme dit précédemment, on peut s'arrêter dès qu'un noeud est atteint.
3. C'est le parcours en profondeur qui modélise une navigation internet

On veut implémenter le parcours en profondeur.

— On prendra l'implémentation des graphes par les listes d'adjacentes.

— On va garder en mémoire les sommets visités dans un tableau `visit` de type `bool array`.

Implémentons l'algorithme récursif. Il n'y a pas lieu de s'occuper précisément la frontière.

On va réaliser un parcours en profondeur, sans rien faire de particulier et en s'arrêtant uniquement quand tous les sommets auront été visités.

Proposition 3.2.12 (Complexité du parcours en profondeur)

On suppose que les opérations pour enpiler et dépiler se font en temps constant. On voit que lors du parcours en profondeur tous les sommets sont visités une fois et chaque arête est considérée une fois (ou deux fois dans le cas d'un graphe non orienté).

— On a donc une complexité temporelle

— La complexité en espace est de

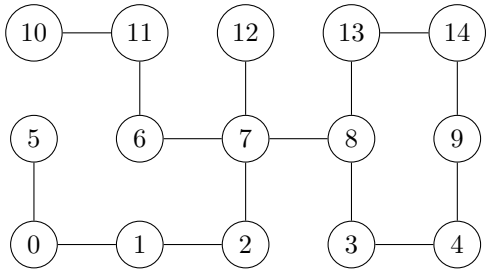
3.3 Parcours en largeur (Breadth First Search)

On veut maintenant étudier un parcours où l'on explore **tous les voisins** d'un sommet avant de passer aux voisins de ses voisins.

Le principe est d'utiliser une file (de type FIFO) pour stocker la frontière

— On initialise
  — la liste des sommets déjà traités (Visit dans l'algorithme précédent) à la liste vide.
  — la **file** des sommets à traiter (F dans l'algorithme précédent) en mettant dedans le sommet d'origine.
— Tant que la file des sommets à traiter n'est pas vide, on traite la tête de la file (s'il n'a pas été déjà visité). Pour cela :
  — on l'enlève de la file
  — on l'insère dans la liste des sommets traités
  — on ajoute tous ses voisins à la queue de la file des sommets à traiter

Voici, dans le cas de notre exemple comment évolue la liste F et la file Visit. Dans la colonne « sommets à traiter », on indique la file des sommets à traiter où on insère dans la file à droite et on extrait à gauche. Par commodité on insérera à chaque étape les voisins par ordre croissant.



sommets traités	sommets à traiter
	10
10	11
10, 11	6

Remarques :

1. Dans un parcours en largeur on traite **tous les voisins** d'un sommet avant de passer aux voisins d'un autre sommet. Cela correspond au parcours en largeur d'un arbre.
2. Dans l'exemple ci-dessus on cherche à parcourir tout le graphe, comme dit précédemment, on peut s'arrêter dès que un noeud est atteint.
3. C'est ce genre de parcours que l'on utilise si on veut regarder en priorité les noeuds « proches » du sommet de départ.

On veut implémenter le parcours en largeur.

— On prendra encore l'implémentation des graphes par les listes d'adjacentes.

— On va garder en mémoire les sommets visités dans un tableau `visit` de type `bool array`.

— On implémentera les files par une **référence de liste**. Les constructeurs sont alors

  — `let creeFile ()`

  — `let enfile f a =` qui permet d'enfiler un entier

  — `let estVide f =`

  — La fonction defile est

  `let defile f =`

On va avoir aussi besoin d'une fonction permettant de rajouter dans la file tous les éléments d'une liste.

```
let rec enfileliste  f l = match l with
  |[] -> ()
  |t::q -> enfile f t; enfileliste  f q;;
```

Le parcours en largeur est alors :

Proposition 3.3.13 (Complexité du parcours en largeur)

On suppose que les opérations pour enfiler et défiler se font en temps constant. On voit que lors du parcours en largeur tous les sommets sont visités une fois et chaque arête est aussi visitée une fois.

— On a donc une complexité temporelle

— La complexité en espace est de

3.4 Quelques exemples

3.4.1 Recherche des composantes connexes

On se donne un graphe g non orienté (il n'y a donc pas de différences entre connexe et fortement connexe) et on veut savoir s'il est connexe et si ce n'est pas le cas faire la liste de ses composantes connexes. Avant cela on va s'intéresser à la recherche de l'existence d'un chemin entre deux sommets.

— On se donne deux sommets i et j et on veut savoir s'il existe un chemin qui va de i à j. L'idée est de parcourir le graphe en partant de i jusqu'à atteindre (ou pas) j. On réalise un parcours en profondeur (recursif).

— Ecrivons maintenant une fonction qui renvoie la composante connexe d'un sommet i. Il suffit juste d'ajouter un accumulateur qui garde la liste de sommets visités. Faire attention à n'ajouter les sommets que la première fois.

— Pour finir on peut obtenir la liste de toutes les composantes connexes en relançant la fonction précédente sur un sommet qui n'est pas encore dans les composantes connexes déjà établies.

3.4.2 Recherche du plus court chemin

On se donne deux sommets i et j et on veut trouver le plus court chemin entre les deux (s'il en existe un).

Cette fois, le parcours en profondeur est moins utile.

On voit que le parcours en largeur parcourt le graphe en visitant d'abord les nuds de distance 1, puis ceux de distance 2 et ainsi de suite. L'idée est donc de réaliser un parcours en largeur.

Une fois que l'on aura atteint le sommet cible, on voudra reconstruire le chemin.

Pour cela, on va garder en mémoire, pour chaque nud atteint, le noeud précédent dans le chemin le plus court (c'est-à-dire le nud d'où l'on vient lors que l'on considère ce nud **pour la première fois**).

De ce fait, la file contient en permanence les nuds à visiter ainsi que les nuds qui les précédent.

On utilise aussi un tableau `prec` tel que `prec.(i)` contient le numéro du sommet qui précède i.

4 Chemin de plus faible poids dans un graphe pondéré

Dans de nombreux cas, ce n'est pas le nombre d'arêtes traversées qui sera utile, certaines arêtes pouvant « compter plus que d'autres ».

4.1 Graphes pondérés

4.1.1 Définition

Définition 4.1.14

On appelle graphe pondéré un graphe $G = (S, A)$ muni d'une application $p : A \to X$ où X est l'ensemble des valeurs (la plupart du temps $\mathbf{R}$)

Remarque : Cela revient à affecter un « poids » à chaque arête.

Définition 4.1.15

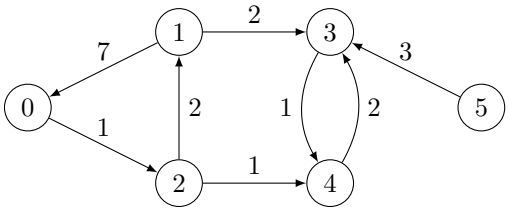
Soit $G = (S, A, p)$ un graphe pondéré. Soit $x_0, x_1, \ldots, x_n$ un chemin du graphe, le poids du chemin est la somme des poids des arêtes qui constituent le chemin.

Remarque : Dans le cas où notre graphe représente des routes, la pondération sera la longueur ou le temps nécessaire au parcours de la route. Dans ce cadre, on voit que la recherche d'un chemin de poids minimum entre deux nuds est crucial. Nous y reviendrons.

ATTENTION

Ne pas confondre la longueur (nombre d'arêtes) d'un chemin et son poids (somme des poids des arêtes).

Exemple :



Pour aller de 0 à 3 on a un chemin de poids 4 et un chemin de poids 5.

4.1.2 Implémentation par matrice d'adjacence

On peut modifier l'implémentation d'un graphe par une matrice d'adjacence afin de traiter le cas des graphes pondérés. Il y a essentiellement deux manières pour faire cela :

4.1.3 Implémentation par listes d'adjacences

Il y a là encore plusieurs manières de représenter un graphe pondéré avec des listes d'adjacences.

— On peut considérer des listes de couples `(int * float)` où la première coordonnée est le voisin et la deuxième le poids du chemin.

— On peut considérer deux listes. La liste des adjacences et la liste des poids.

4.2 Recherche d'un chemin de plus faible poids

On considère un graphe connexe pondéré. On se donne deux sommets et on veut déterminer un/le chemin de poids minimal entre ces deux sommets.

Il y a une condition nécessaire évidente pour qu'un tel chemin existe.

On supposera toujours par la suite cette condition vérifiée.
Ce problème modélise par exemple la recherche du plus court chemin entre deux points par un logiciel de GPS.
Il existe deux algorithmes classiques : l'algorithme de Floyd-Warshall et l'algorithme de Dijkstra.

4.3 Algorithme de Floyd-Warshall

4.3.1 Le principe

Pour cet algorithme il est plus simple de représenter le graphe par une matrice d'adjacence. On notera donc $0, 1, 2 \ldots, (n-1)$ les sommets du graphe et $A = (a_{ij}) \in \mathscr{M}_n(\overline{\mathbf{R}})$ la matrice d'adjacence où pour tout couple (i, j), a_{ij} est le poids de l'arête allant du sommet i au sommet j en prenant la convention $a_{ij} = +\infty$ s'il n'y a pas d'arête entre i et j.

Ici les indices des éléments de la matrice vont de 0 à $n-1$ contrairement à l'usage en mathématique.

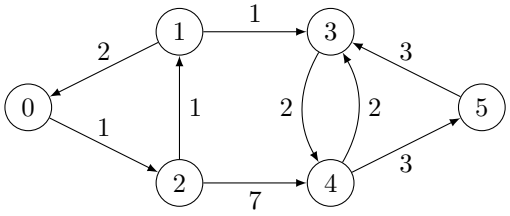
L'idée est alors de calculer de proche en proche les matrices $A^{(k)} = (a_{ij}^{(k)})$ où $a_{ij}^{(k)}$ est le poids minimal d'un chemin allant de i à j **et ne passant autrement que par des sommets** $l \leqslant k$.

Théorème 4.3.16

Avec les notations précédentes
- $A^{(-1)} = A$
- $\forall (i,j) \in [\![0\,;\, n-1]\!]^2, \forall k \in [\![-1\,;\, n-2]\!], a_{ij}^{(k+1)} =$

Démonstration :

4.3.2 Exemple



On considère le graphe pondéré suivant
On a donc

$$A = A^{(-1)} = \begin{pmatrix} \infty & \infty & 1 & \infty & \infty & \infty \\ 2 & \infty & \infty & 1 & \infty & \infty \\ \infty & 1 & \infty & \infty & 7 & \infty \\ \infty & \infty & \infty & \infty & 2 & \infty \\ \infty & \infty & \infty & 2 & \infty & 3 \\ \infty & \infty & \infty & 3 & \infty & \infty \end{pmatrix}.$$

On en déduit que

$$A^{(0)} = \begin{pmatrix} \infty & \infty & 1 & \infty & \infty & \infty \\ 2 & \infty & \mathbf{3} & 1 & \infty & \infty \\ \infty & 1 & \infty & \infty & 7 & \infty \\ \infty & \infty & \infty & \infty & 2 & \infty \\ \infty & \infty & \infty & 2 & \infty & 3 \\ \infty & \infty & \infty & 3 & \infty & \infty \end{pmatrix}.$$

Exercice : Calculer $A^{(1)}$ et $A^{(2)}$

En trouve finalement

$$A^{(5)} = \begin{pmatrix} 4 & 2 & 1 & 3 & 5 & 8 \\ 2 & 4 & 3 & 1 & 3 & 6 \\ 3 & 1 & 4 & 2 & 4 & 7 \\ \infty & \infty & \infty & 4 & 2 & 5 \\ \infty & \infty & \infty & 2 & 4 & 3 \\ \infty & \infty & \infty & 3 & 5 & 8 \end{pmatrix}.$$

On obtient donc que le chemin de poids le plus faible pour aller de 0 à 5 a un poids 8.

4.3.3 Implementation

On peut implémenter cet algorithme. On rappelle que l'on utilise le type `type int_inf = Entier of int | Inf` et que l'on a étendu l'addition et la recherche du minimum avec deux fonctions `add : int_inf -> int_inf -> int_inf` et `mini : int_inf -> int_inf -> int_inf`.

Remarque : Dans le programme précédent il n'est pas nécessaire de « sauvegarder » `a.(i).(j)`, lors du calcul, le fait qu'il n'y ait pas de cycle de poids négatif nous permet de remplacer $a_{i,k+1}^{(k)}$ et $a_{k+1,j}^{(k)}$ par $a_{k+1,j}^{(k+1)}$.

Proposition 4.3.17 (Complexité de l'algorithme de Floyd-Warshall)

Exercice : On veut utiliser l'algorithme de Floyd-Warshall pour écrire une fonction `chemin` telle que `chemin g i j` renvoie le poids d'un chemin de poids minimal allant de i à j ainsi que le chemin sous-forme d'une liste.

Pour cela, commencer par écrire une fonction `fwChemin : int_inf array array -> (int * int_inf) array array` telle que la fonction renvoie une matrice `a` où, pour tout i, j, `a.(i).(j)` contient le couple `(prec,poids)` où `poids` est le poids d'un plus court chemin de i à j (s'il en existe un) et `prec` est le sommet qui précède j dans ce chemin.

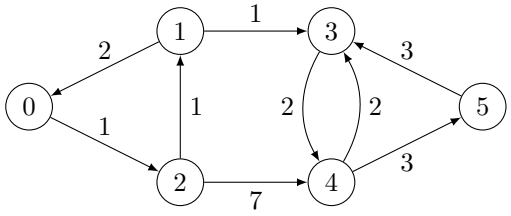
Ensuite écrire la fonction `chemin : int_inf array array -> int - int -> int list` qui renvoie un plus court chemin de i à j (on pourra supposer qu'un tel chemin existe).

4.4 Algorithme de Dijkstra

Nous allons étudier un autre algorithme pour déterminer le chemin de poids minimal dans un graphe. Nous étudierons à la fin les avantages et les inconvénients de chacun.

4.4.1 L'algorithme de Dijkstra

On suppose maintenant que tous les poids sont positifs. On considère cette fois le graphe donné par des listes d'adjacences. Précisément on suppose que les sommets du graphe sont les entiers compris entre 0 et $n-1$ et que l'on dispose d'un tableau de listes : `adjacente` où `adjacente.(i)` est la liste des couples (t, pt) où t est un voisin de i et pt le poids de l'arête (i, t). Par exemple, si on reprend le graphe précèdent



On obtient

adjacence	0	1	2	3	4	5
	$[(2,1)]$	$[(0,2);(3,1)]$	$[(1,1);(4,7)]$	$[(4,2)]$	$[(3,2);(5,3)]$	$[(3,3)]$

On se donne un sommet de départ (que nous noterons `i0`) et le but est de construire un tableau `poidsChemin` tel que `poidsChemin.(j)` soit le poids minimal d'un chemin de `i0` à `j`.

L'algorithme est le suivant :

Algorithme 2 : Algorithme de Dijkstra

Entrée Graphe $G = (S, A)$; sommet de départ `i0`
:

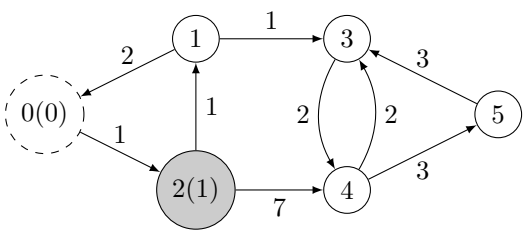
```
F ← S
poidsChemin.(j) ← ∞ pout tout sommet j
poidsChemin.(i0) ← 0
tant que F ≠ ∅ faire
    Choisir un sommet k dans F tel que PoidsChemin.(k) est minimal
    Enlever k de F
    pour l ∈ V_k faire
        PoidsChemin.(l) ← Min(PoidsChemin.(l) , PoidsChemin.(k) + poids(k,l) )
    fin
fin
```

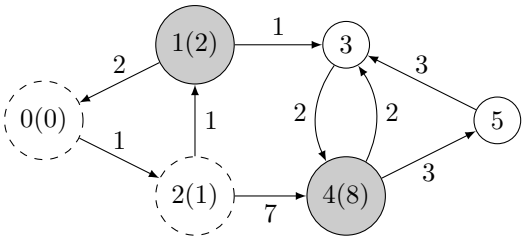
Théorème 4.4.18

A la fin de l'execution de l'algorithme ci-dessus, `poidsChemin.(j)` contient le poids minimal d'un chemin de `i0` à `j`.

Exemple : Commençons par un exemple

Appliquons l'algorithme de Dijkstra sur notre exemple. Les sommets visités deviendrons en pointillés alors que les sommets de la frontière vont être grisés. On indique entre parenthèses le poids du chemin.





Démonstration : On notera $E = V \setminus F$ l'ensemble des sommets qui ne sont plus dans F. C'est-à-dire les sommets qui ont été traités

On remarque , qu'à tout moment, `PoidsChemin.(j)` $\geqslant \delta(i0, j)$ où $\delta(i0, j)$ est le poids du chemin de plus faible poids entre $i0$ et j.

On utilise l'invariant de boucle suivant :

Au début de chaque itération de la boucle Tant que,

pour tout j qui est dans E, `poidsChemin.(j)` est le poids minimal d'un chemin de `i0` à `j`

— Initialisation : Au début, $E = \emptyset$ donc l'invariant est vrai.

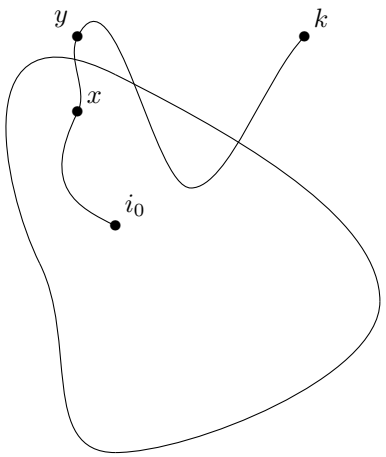
— Supposons par l'absurde qu'il existe un élément k tel que l'invariant de boucle était vrai avant d'ajouter k et que l'invariant ne soit plus vérifié après l'ajout de l'élément k. On peut de plus supposer que k est le « premier » sommet vérifiant cela. On note E l'ensemble des sommets traités avant d'ajouter k et $E' = E \cup \{k\}$.

— Pour tous les sommets j qui ont été ajoutés à E avant k, on a `PoidsChemin.(j)` $= \delta(i0, j)$.

— On en déduit qu'à la fin de la boucle, `PoidsChemin.(k)` $\neq \delta(i0, k)$ et donc $k \neq i0$. On considère un chemin p de plus faible poids de $i0$ à k. Ce chemin allant de $i0$ (qui est dans E) à k (qui n'est pas dans E), on peut considérer y le premier sommet du chemin qui n'est pas dans E ainsi que x le sommet qui précède dans le chemin. On a donc

$$p = i0 \xrightarrow{p_1} x \to y \xrightarrow{p_2} k.$$

où p_1 est un chemin intégralement dans E (éventuellement vide) et p_2 est un chemin qui peut passer (ou pas) dans E (lui aussi éventuellement vide).



Maintenant, **au moment où x a été ajouté à E**, on avait `PoidsChemin.(x)` $= \delta(i0, x)$ car cette égalité est vraie après l'ajout de x et que l'on ne modifie pas `PoidsChemin.(x)` à ce moment. On en déduit qu'après cet ajout

$$\begin{aligned} \texttt{PoidsChemin.(y)} &\leqslant \texttt{PoidsChemin.(x)} + \texttt{poids(x,y)} \\ &\leqslant \delta(i0, x) + \texttt{poids(x,y)} \\ &\leqslant \delta(i0, y) \end{aligned}$$

La dernière inégalité vient du fait que p est un chemin de poids minimal donc tout sous-chemin est encore de poids minimal.
On a donc

$$\texttt{PoidsChemin.(y)} \leqslant \delta(i0, y) \leqslant \delta(i0, k) \leqslant \texttt{PoidsChemin.(k)}.$$

Comme, par définition du choix de k, `PoidsChemin.(k)` $\leqslant$ `PoidsChemin.(y)`, on obtient que `PoidsChemin.(k)` $= \delta(i0, k)$ ce qui contredit notre hypothèse de départ.

L'invariant de boucle reste donc vrai. A la fin de la boucle, tous les sommets sont dans E et `poidsChemin.(j)` est le poids minimal d'un chemin de `i0` à `j`. □

Contre-exemple S'il y a des arêtes avec un poids négatif, l'algorithme de Dijkstra ne donnera pas nécessairement le chemin de plus faible poids.

4.4.2 Complexité et différence par rapport à Floyd-Warshall

— Contrairement au cas de l'algorithme de Floyd-Warshall, dans l'algorithme de Dijkstra on a besoin d'une origine. De plus il ne calcule que les chemins en partant de cette origine.

— L'algorithme de Dijkstra a une complexité meilleure

Il est clair que l'algorithme passe $|S|$ fois dans la boucle **tant que**, si on stocke les poids des chemins dans un tableau et que l'on fait une recherche linéaire, on a donc une complexité en temps totale $O(|S|^2 + |A|)$ puisque l'on considère chaque arête une fois dans la ligne 8 de l'algorithme.

Cependant on peut faire mieux en considérant une autre structure de données.

Théorème 4.4.19

Démonstration :

4.4.3 Implémentation

On va utiliser une file de priorité f (implémentée par un tas min) pour notre ensemble F. Elle contient des triplets `(x,p,a)` où `x` est un sommet, `p` le poids du chemin de `i` à `x` et `a` l'antécédent de `x` dans ce chemin.

Afin de reconstruire aussi le chemin, on va, parallèlement à la construction du tableau `poidsChemin` construire un tableau `antecedent` tel que `antecedent(j)` soit le sommet précédent `j` dans le chemin (de plus faible poids) allant de `i` à `j`.

On rappelle les constructeurs des files de priorité

- `creeFileVide : unit -> filePriorite` qui créé une file vide.
- `insertFile : (int * int * int) -> filePriorite -> unit` qui ajoute un élément (par effets de bord) à la file.
- `defile : filePriorite -> (int*int *int)` qui renvoie le sommet de la file (dont le poids est le plus faible) et le supprime de la file (par effets de bord)

On rappelle que l'on peut implémenter cela dans un tableau de type `(int*int*int) array` ayant suffisamment de cases (ou utiliser un tableau redimensionable comme cela a été fait en exercice lors du chapitre 1). Il faut alors utiliser un champ supplémentaire pour stocker le nombre de données réellement présentes dans la file. On utilise donc le type `type filePriorite = {mutable taille : int ; donnee : (int*int*int) array}`.

Exercice : Écrire les fonctions `creeFileVide`, `insertFile` et `defile`.

On peut alors écrire notre programme

```
let dijkstra g  i j =
let n = Array.length g in
let poidsChemin = Array.make n (-1) in
let antecedent = Array.make n (-1) in
poidsChemin.(i) <- -0;
let visit = Array.make n false in
let rec parcours f =
  let (x,p,a) = defile f in
  if not(visit.(x)) then
    (visit.(x) <- true;
     poidsChemin.(x) <- p;
     antecedent.(x) <- a;
     let rec traiteliste x p  l =
      match l with
         |[] -> ()
         | (v,pc):: q -> if not (visit.(v)) then insertFile (v,p+pc,x) f ; traiteliste x p q
     in
     traiteliste x p g.(x));
  if x <> j then parcours f in
let f = creeFileVide () in
insertFile  (i,0,-1) f;
parcours f;
(poidsChemin,antecedent);;
```

Remarques :

1. On voit que on commence avec une file qui ne contient que l'origine et on ajoute les sommets dedans. De plus on ajoute sans s'occuper de savoir si cela optimise la distance, de fait si cela ne l'optimise pas cela le placera dans la queue de la file.
2. Le tableau `visit` sert à ne pas visiter plusieurs fois un même sommet.

Exercice : Écrire la fonction pour reconstruire le chemin.