Chapitre 5

Graphes

5.1 Définitions

5.1.1 Graphes non-orientés

Définition 5.1.1
On appelle graphe *(simple) non-orienté tout couple* $G = (S, A)$ *où*

Les graphes seront représentés graphiquement, les sommets étant représentés par leurs noms, et les arêtes par des traits reliant les sommets.

EXEMPLE

Définition 5.1.2
Soit $G = (S, A)$ *un graphe. Soit* $a = \{x, y\}$ *une arête de* G.

- x *et* y *sont appelés*
- x *et* y *sont dits*

On appelle alors degré de x

Proposition 5.1.3
Soit $G = (S, A)$ un graphe à m arêtes. Alors

Démonstration. Par récurrence sur m :

- si $m = 0$, alors
  sommet est nul
- soit G un graphe à $m + 1$ arêtes. Soit $a = \{x, y\}$ une arête de G. Alors le graphe où l'on enlève a a exactement m arêtes, et on peut appliquer l'hypothèse de récurrence.

  En enlevant s,

  d'où le résultat voulu.

□

Définition 5.1.4
Soit $G = (S, A)$ un graphe. On appelle chemin de longueur k reliant x et y

Si $x = y$, et que tous les autres x_i sont distincts, on dit que le chemin est

Un chemin est dit élémentaire *si*

Pour tous sommets x et y, on appelle distance de x à y

Un graphe tel que tout couple de sommets à une distance finie est dit

Proposition 5.1.5
Un graphe connexe à n sommets a au moins arêtes.

Démonstration. Par récurrence sur n :

- si $n = 0$ ou $n = 1$, c'est évident
- sinon, soit G à $n + 1$ sommets. Deux cas se présentent :
  - si G possède un sommet de degré 1, alors

– sinon, tous les sommets sont de degrés au moins 2, et donc
  . Cette somme étant deux fois le nombre d'arêtes, il y a au moins arêtes.

□

Définition 5.1.6
Soit $G = (S, A)$ *un graphe. Un* sous-graphe *de* G *est*

Si $S' \subseteq S$*, on appelle* sous-graphe de G induit par S'

Définition 5.1.7
On appelle composante connexe *d'un graphe*

Définition 5.1.8
Un graphe pondéré *est un triplet* $G = (S, A, p)$ *où* (S, A) *est un graphe, et* p *une fonction de* A *dans* $\mathbb{R}$*. L'image d'une arête par* p *est appelée son* poids*.*

5.1.2 Graphes orientés

Définition 5.1.9
On appelle graphe (simple) orienté

On les représentera comme les graphes non orientés, mais en remplaçant les arêtes par des flèches.

EXEMPLE

NOTA

On travaillera dans la suite avec des graphes non orientés. Tous les résultats s'adapteront facilement.

Définition 5.1.10
Soit $G = (S, A)$ *un graphe orienté, et soient* $x, y \in S$*. On dit que* y *est voisin de* x *si* $(x, y) \in$

A. On appelle alors degré sortant de x *le nombre de voisins de x, et* degré entrant de x *le nombre de sommets dont x est voisin, notés respectivement $d^+(x)$ et $d^-(x)$.*

Un graphe orienté dont tous les sommets sont reliés par des chemins est dit fortement connexe.

EXERCICE

Montrer que pour un graphe orienté à m arêtes,

$$\sum_{x \in S} d^+(x) = \sum_{x \in S} d^-(x) = m.$$

5.2 Exemples de graphes

Définition 5.2.1
On appelle graphe complet à n sommets

EXEMPLE

Voici le graphe K_5 :

Définition 5.2.2
On appelle graphe chemin P_n

EXEMPLE

Voici P_5 :

Définition 5.2.3
On appelle graphe cycle C_n

EXEMPLE

Voici le graphe C_5 :

Définition 5.2.4
On appelle graphe biparti complet K_n^p

EXEMPLE

Voici K_3^3 :

Définition 5.2.5
On appelle arbre

EXEMPLE

Voici un arbre :

5.3 Représentation des graphes

On peut représenter en Caml les graphes de deux façons différentes :

- avec des listes d'adjacence, *i.e.* en associant à chaque sommet la liste de ses voisins.x
- avec des matrices d'adjacence, *i.e.* en notant dans une matrice si oui ou non il y a une arête.

Les opérations qu'on souhaite implémenter sont l'ajout ou la suppression d'un sommet où d'une arête, et le test d'existence d'une arête.

5.3.1 Liste de listes d'adjacence

Ici, on représente un graphe comme la liste des couples (Sommet, Liste des voisins du sommet) :

```
  type 'a sommet = {Nom : 'a; Voisins : 'a list}
  type 'a graphe = 'a sommet list
```

On peut alors implémenter nos fonctions :

```
let rec ajoute_arete_orientee g a b =
  match g with
  |[] -> failwith "ajoute_arete"
  |s::g' when s.nom = a && List.mem b s.voisins -> g
  |s::g' when s.nom = a -> {nom=a; voisins = b::(s.voisins)}::g'
  |s::g' -> s::(ajoute_arete_orientee g' a b)

let rec ajoute_arete g a b =
  ajoute_arete_orientee (ajoute_arete_orientee g a b) b a

let rec delete l a =
  match l with
  |s::l' when s =a -> l'
  |s::l' -> s::(delete l' a)
  |[] -> []

let rec supprime_arete_orientee g a b =
  match g with
  |[] -> failwith "supprime_arete"
  |s::g' when s.nom=a && List.mem b s.voisins ->
    {nom=a; voisins = delete (s.voisins) b}::g'
```

```
  |s::g' -> s::(supprime_arete g' a b)

let rec supprime_arete g a b =
  supprime_arete_orientee (supprime_arete_orientee g a b) b a
```

```
let rec ajoute_sommet g a =
  match g with
  |[] -> [{nom=a; voisins=[]}]
  |s::g' when s.nom = a -> g
  |s::g' -> s::(ajoute_sommet g' a)

let rec supprime_sommet g a =
  match g with
  |[] -> []
  |s::g' when s.nom = a -> supprime_sommet g' a
  |s::g' ->
    {nom=s.nom; voisins = delete s.voisins a}::(supprime_sommet g' a)
```

```
let rec check_arete g a b =
  match g with
  |[] -> false
  |s::_ when s.nom = a ->
    List.mem b (s.voisins)
  |s::_ when s.nom = b ->
    List.mem a (s.voisins)
  |s::g' -> check_arete g' a b
```

5.3.2 Tableau de liste d'adjacence

Dans cette partie, on considère que les sommets des graphes sont $0, 1, \ldots, n-1$.

On aurait aussi pu choisir de faire un tableau de listes d'adjacences. Dans ce cas, l'accès à la liste des voisins d'un nœud est beaucoup plus rapide, mais il est en revanche impossible de rajouter un nœud.

EXERCICE

Réécrire les fonctions précédentes en utilisant des tableaux de liste d'adjacence.

5.3.3 Matrice d'adjacence

Si on veut pouvoir accéder en temps constant aux sommets et aux arêtes, on peut utiliser une matrice d'adjacence. Si les sommets sont $1, \ldots, n$, on définit la matrice $M \in \mathcal{M}_n(\{0,1\})$ par

$$m_{ij} =$$

Ici, il est toujours impossible de supprimer ou ajouter des sommets, mais l'ajout ou la suppression d'arêtes se fait en temps constant. En revanche, l'espace mémoire occupé par une matrice d'adjacence est beaucoup plus important que pqr les listes d'adjacence.

EXERCICE

Donner la matrice d'adjacence du graphe du début du cours.

5.3.4 Désorientation

Les représentations précédentes s'appliquent aux graphes orientés ou non. On a cependant une injection canonique des graphes orientés dans les graphes non orientés.

EXERCICE

Écrire une fonction de désorientation pour les listes d'adjacence et pour les matrices d'adjacence.

5.3.5 Passage de l'une à l'autre

On peut évidemment passer d'une représentation à l'autre :

```
let graphe_to_matrice g =
  let rec aux g mat =
    match g with
    |[] -> mat
    |s::g' -> match s.voisins with
              |[] -> aux g' mat
              |n::a ->
                mat.(s.nom).(n) <- 1;
                aux ({nom=s.nom; voisins = a}::g') mat
  in
  let n = List.length g in
  let mat = Array.make_matrix n n 0 in
  aux g mat
```

```
let liste_voisins_mat mat n =
  let p=Array.length mat in
  let l= ref [] in
  for i=0 to (p-1) do
    if mat.(n).(i) = 1
    then l := i::!l
  done;
  !l

let matrice_to_graphe mat =
  let l = ref [] in
  let n = Array.length mat in
  for i=0 to (n-1) do
    l := {nom=i; voisins = liste_voisins_mat mat i}::!l
  done;
  !l
```

5.4 Parcours d'un graphe

Pour utiliser un graphe, on a souvent besoin de traiter tous les sommets : il faut parcourir le graphe. La question qui se pose est de choisir l'ordre de visite de chacun des nœuds.

Pour ces parcours, on utilisera la représentation des graphes par liste d'adjacence.

5.4.1 Parcours en largeur

Dans ce type de parcours, on commence par regarder tous les voisins d'un sommet avant de passer aux sommets suivants. C'est un parcours qui peut servir à calculer la distance entre deux sommets.

L'algorithme utilise deux listes : la liste des sommets déjà visités, et la liste des sommets à voir. À chaque sommet visité, on ajoute tous ses voisins à la liste des sommets à voir.

On utilise donc une structure de file d'attente (FIFO).

```
let rec get_voisins g s =
  match g with
  |[] -> failwith "get_voisins"
  |s'::g'-> if s'.nom = s
            then s'.voisins
            else get_voisins g' s

let rec bfs_aux g vus avoir =
```

```
  match avoir with
  |[] -> List.rev vus
  |s :: q ->
    if List.mem s vus
    then bfs_aux g vus q
    else bfs_aux g (s::vus) (q @ get_voisins g s)

let bfs g s =
  bfs_aux g [] [s];;
```

Proposition 5.4.1
La fonction `bfs` *est de complexité* , *où n est le nombre de sommets et m le nombre d'arêtes.*

5.4.2 Parcours en profondeur

Dans ce type de parcours, on commence par suivre un chemin avant de passer à un autre voisin. On utilisera donc une pile (LIFO).

```
let rec dfs_aux g vus avoir =
  match avoir with
  |[] -> List.rev vus
  |s :: q ->
    if List.mem s vus
    then bfs_aux g vus q
    else bfs_aux g (s::vus) (get_voisins g s @ q)

let dfs g s =
  dfs_aux g [] [s]
```

5.4.3 Connexité d'un graphe

Le parcours d'un graphe peut en particulier servir à tester sa connexité : en parcourant le graphe à partir d'un sommet quelconque, il n'y a plus qu'à vérifier qu'on a bien visité tous les sommets.

```
let connexe g =
  match g with
  |[] -> true
  |a::_ ->
```

```
    let taille = List.length g in
    let taille_parcours = List.length (dfs g a.nom) in
    taille = taille_parcours
```

Pour trouver les compostantes connexes, on utilise un parcours en profondeur, puis une fois arrivé au bout, on recommence à partir d'un sommet qui n'a pas encore été visité.

```
let rec listesommets graphe =
  match graphe with
  |[] -> []
  |s::g' -> s.nom::(sommets g')

exception Fini

let rec libre sommets visites =
  match sommets with
  |[] -> raise Fini
  |s :: reste -> if List.mem s visites
                 then libre reste visites
                 else s

let comp_connexes graphe =
  let sommets = listesommets graphe in
  let rec aux sommets comp visites =
    try
      let s = libre sommets visites in
      let parcours = dfs graphe s in
      aux sommets (parcours::comp) (visites @ parcours)
    with
      Fini -> comp
  in aux sommets [] []
```

5.5 Plus court chemin

Dans cette partie, on va essayer de trouver le chemin le plus court entre deux sommets.

Dans le cas où le graphe n'est pas pondéré, c'est facile : il suffit de faire un parcours en largeur du graphe à partir du premier sommet, jusqu'à rencontrer le second.

Nous allons donc travailler avec des graphes pondérés. Il y a trois types de recherche possibles :

-
-
-

On ne connaît pas, en général, d'algorithme permettant de résoudre directement le premier problème. Le second est résolu par l'algorithme de Dijkstra, et le troisième par l'algorithme de Floyd-Warshall.

Ces algorithmes sont basés sur le lemme suivant :

Lemme 5.5.1
S'il existe un plus court chemin entre a et b qui passe par c, alors

5.5.1 Algorithme de Floyd-Warshall

On utilise la représentation des graphes par matrice d'adjacence : le coefficient i, j de la matrice sera $+\infty$ s'il n'y a pas d'arête entre i et j, et sera le poids de l'arête sinon.

Les sommets du graphe sont donc supposés être $1, \dots, n$.

On suppose que les poids des arêtes sont tous positifs : dans le cas contraire, il suffit d'ajouter l'opposé du plus petit poids pour translater le problème.

Soit M la matice qu'on considère.

On définit alors une suite de matrices par

$$M^{(0)} =$$

$$\left(M^{(k+1)}\right)_{i,j} =$$

Proposition 5.5.2
Pour tous $i, j \in [\![1, n]\!]$ et tout $k \leqslant n$, si $M^{(k)}_{i,j} \neq \infty$, alors

Démonstration. Par récurrence sur k.

- Pour $k = 0$, c'est évident.
- Supposons la propriété vraie pour $k < n$. Soit c un chemin dont les sommets intermédiaires sont dans $[\![1, k+1]\!]$. Deux cas se présentent :

- si $k+1$ n'est pas parcouru alors
- si $k+1$ est parcouru, c se décompose nécessairement en

  le premier et le dernier chemin ne passant pas par $k+1$. Le premier chemin est donc de longueur supérieure à et le second à ; en ajoutant le poids du cycle central, on a donc un poids supérieur à

Dans les deux cas, le poids de c est supérieur à $M_{i,j}^{(k+1)}$.

On peut facilement vérifier qu'un chemin de poids $M_{i,j}^{(k+1)}$ existe bien, et on a donc montré le résultat.

□

On en déduit alors le résultat :

Proposition 5.5.3
Pour tous i et j, le plus court chemin de i à j est de poids

Proposition 5.5.4
La complexité de l'algorithme de Floyd-Warshall est un , où n est le nombre de sommets du graphe.

Démonstration. Chaque matrice demande le calcul de n^2 coefficients, chacun étant calculé avec deux opérations.

On calcule n matrices, d'où la complexité demandée. □

EXERCICE

Définir un type `poids` qui est soit `Inf`, soit un entier, puis programmer l'algorithme de Floyd-Warshall.

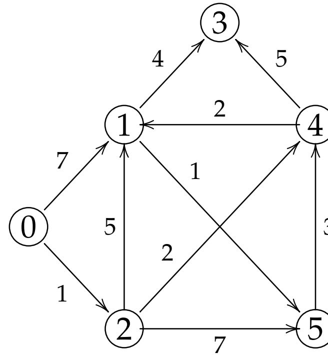
5.5.2 Algorithme de Dijkstra

Cet algorithme permet de trouver les plus courts chemins en partant d'un sommet fixé s. L'idée est la suiante :traités de distance minimale, et on atualise le tableau des distances :

```
fonction Dijkstra(graphe,s):
  Visite = {s}
```

```
  Pour tous v dans Sommets:
    d(v) <- poids(s,v)
  Tant que Visite <> Sommets:
    Choisir u non visite de d(v) minimal
    Visite = Visite + {u}
    Pour tout v non visite:
    d(v) <- min(d(v), d(u)+poids(u,v))
```

Regardons l'évolution de l'algorithme sur l'exemple suivant, en partant du sommet 0 :



-
-
-
-

On peut résumer l'évolution dans un tableau :

Visite	0	1	2	3	4	5

L'implémentation de cet algorithme sera vue en TP.

Proposition 5.5.5
La complexité au pire de l'algorithme de Dijkstra est en

Une bonne implémentation permet d'avoir une complexité en
, où n et m sont respectivement le nombre de sommets et d'arêtes du graphe.

5.6 Exercices

Exercice 1. Montrer que tout graphe non orienté connexe peut s'obtenir en ajoutant un nombre fini d'arêtes à un arbre.

Exercice 2. On dit qu'un graphe non orienté est *planaire* si on peut le représenter dans le plan sans que les arêtes ne se croisent. On appelle alors *face* de la représentation de G toute région maximale qui ne contient pas d'arête.

a) Le degré d'une face est le nombre d'arêtes la délimitant. Montrer que si G est un graphe planaire à m arêtes, alors

$$\sum_{F \text{ face}} \deg(F) \geqslant 2a.$$

b) Soit G planaire à s sommets, a arêtes et f faces. Montrer la formule d'Euler :

$$s - a + f = 2.$$

Indication : on pourra montrer qu'elle est vraie pour des arbres, et qu'elle reste vraie en ajoutant une arête qui ne coupe pas les autres.

c) Montrer que pour tout graphe planaire connexe à s sommets et a arêtes, alors

$$a \leqslant 3s - 6.$$

Montrer que si le graphe est sans triangle (*i.e.* sans cycle de longueur 3), alors

$$a \leqslant 2n - 4.$$

d) Montrer que K_5 et K_3^3 ne sont pas planaires.

Exercice 3. On appelle isomorphisme entre deux graphes non orientés $G = (S, A)$ et $G' = (S', A')$ toute bijection $f : A \to A'$ telle que pour tous $u, v \in S$, $\{u, v\} \in A \Leftrightarrow \{f(u), f(v)\} \in A'$.

On appelle automorphisme de graphe toute isomorphisme d'un graphe vers lui-même.

a) Montrer que l'ensemble des automorphismes d'un graphe muni de la composition est un groupe.

b) Déterminer l'ensemble des automorphismes d'un graphe G si :

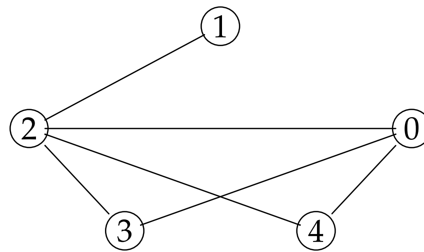
   i. $G = K_n$
   ii. $G = C_n$
   iii. $G = P_n$
   iv. $G = K_p^q$.

Exercice 4. Soit M la matrice d'adjacence d'un graphe orienté G a n sommets.

a) Soit $p \in \mathbb{N}$. Que représente M^p ?

b) Montrer que G est un arbre si et seulement si $M^n = 0$.

Exercice 5. E3A 2018 :



a) Écrire la matrice d'adjacence du graphe ci-dessus.

b) Écrire en Caml une fonction `chemin:int vect vect -> int list -> bool` qui prend en entrée la matrice d'adjacence d'un graphe et un chemin (une liste de sommets du graphe) et qui vérifie si ce chemin est possible dans le graphe. Par exemple, sur le graphe ci-dessus, avec le chemin `[2;1;0;4]` la fonction `chemin` doit renvoyer `false` car les sommets 1 et 0 ne sont pas connectés. Avec le chemin `[1;2;3]` la fonction `chemin` doit renvoyer `true` car les sommets 1 et 2 sont connectés, ainsi que les sommets 2 et 3.

Exercice 6. Proposer un algorithme déterminant l'existence de cycle de poids strictement négatif dans un graphe pondéré.