Informatique

Cours

Graphes

Ipesup

Objectifs

À l'issue de cette leçon, l'étudiant doit être capable de :
- définir formellement les différentes notions liées aux graphes (graphe orienté, non orienté, pondéré, biparti) ;
- manipuler les concepts fondamentaux tels que chemin, cycle, degré, connexité et arbre ;
- représenter un graphe en machine à l'aide d'une matrice d'adjacence ou de listes d'adjacence ;
- comparer ces représentations en termes de complexité en temps et en mémoire.

D'une manière informelle, un graphe est un ensemble d'objets, appelés sommets, dont certains sont reliés deux à deux par des arcs. Les graphes permettent de modéliser *beaucoup* de situations. En voici quelques exemples.

un réseau routier Les sommets représentent des villes (ou des points sur une carte) et les arcs des routes entre elles, éventuellement étiquetées par des informations comme la longueur ou la vitesse maximale autorisée.

un réseau informatique Les sommets sont des machines et les arcs des connexions entre elles, pouvant préciser un port ou la nature de la connexion.

un réseau social Les sommets sont des individus et les arcs des relations entre eux, comme « être une connaissance de ». Un exemple célèbre est le graphe des coautorships : deux mathématiciens sont reliés s'ils ont écrit un article ensemble. On y définit le *nombre d'Erdős* d'un mathématicien X comme la distance entre Paul Erdős et X.

un labyrinthe Les sommets sont des salles et les arcs des portes reliant ces salles.

une carte Les sommets représentent des pays, régions ou départements, et les arcs traduisent une contiguïté : deux pays sont reliés s'ils sont voisins.

le Web Les sommets sont les pages de la Toile et les arcs les liens hypertextes entre elles.

un jeu Les sommets sont les configurations possibles et les arcs les coups valides. Le graphe peut être infini si le nombre de configurations est infini, par exemple lorsque la mise ou les gains ne sont pas bornés.

Avec tous ces exemples, on anticipe l'intérêt d'algorithmes pouvant répondre à des questions comme « existe-t-il un chemin dans le graphe reliant tel sommet à tel autre sommet ? », ou « quelle est la plus longue distance entre deux sommets ? » ou encore « le graphe est-il connexe, *i.e.*, tous les sommets du graphe sont-ils reliés ? ».

1 Définitions

Dans cette section, on pose les définitions de la notion de graphe. On commence par les graphes orientés, puis les graphes non orientés et enfin les graphes pondérés.

1.1 Graphes orientés

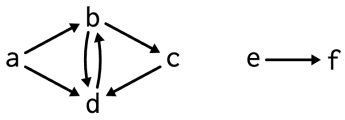
Définition – graphe orienté

Un *graphe orienté* est défini par un ensemble V de *sommets* et un ensemble $E \subseteq V \times V$ de couples de sommets appelés *arcs*.

Un arc $(x, y) \in E$ est traditionnellement dessiné comme une flèche entre les sommets x et y.

Exemple

Voici un exemple de graphe orienté avec six sommets et sept arcs :



On a $V = \{a, b, c, d, e, f\}$ et

$$E = \{(a, b), (a, d), (b, c), (b, d), (c, d), (d, b), (e, f)\}.$$

Il est important de comprendre que le dessin importe peu. Seule la donnée des ensembles V et E définit le graphe.

Entre deux sommets, il existe au plus un arc. Dit autrement, E est un ensemble, pas un multiensemble. Il serait tout à fait possible d'autoriser de tels *multi-arcs* et on parlerait alors de *multi-graphe*. Mais cette notion plus générale n'est pas considérée ici.

Définition - adjacence dans un graphe

Si $(x, y) \in E$, on dit que y est un *successeur* de x et que x est un *prédécesseur* de y. On note $x \to y$ la présence de cet arc. Un arc de la forme (x, x) est appelé une *boucle*. Pour un sommet $x \in V$, le nombre d'arcs de la forme (x, y) est appelé le *degré sortant* du sommet x et noté $d_+(x)$. De même, le nombre d'arcs de la forme (y, x) est appelé le *degré entrant* du sommet x et noté $d_-(x)$.

Sur l'exemple ci-dessus, on a $d_-(a) = 0$ et $d_+(a) = 2$. Dans la suite, on utilisera aussi le terme de *voisins* pour désigner les successeurs d'un sommet.

Définition – chemin dans un graphe

Un *chemin* du sommet u au sommet v dans un graphe (V, E) est une séquence $x_0, \ldots, x_n$ de sommets de V tels que $x_0 = u$, $x_n = v$ et $(x_i, x_{i+1}) \in E$ pour $0 \leq i < n$:

$$u = x_0 \to x_1 \to \cdots \to x_{n-1} \to x_n = v$$

La longueur d'un tel chemin est n, c'est-à-dire le nombre d'arcs qui le constitue. Un *chemin simple* est un chemin sans répétition d'arc. Un *cycle* est un chemin simple de u à u de longueur $n > 0$. Un graphe orienté qui ne contient pas de cycle est appelé un DAG pour *Directed Acyclic Graph*.

On note $x_0 \to^* x_n$ la présence d'un chemin entre les sommets x_0 et x_n. Il y a toujours un chemin de longueur 0 entre un sommet u et lui-même.

Définition – forte connexité

Un graphe orienté $G = (V, E)$ est *fortement connexe* si, pour toute paire de sommets x et y de V, il existe un chemin de x à y. Une *composante fortement connexe* de G est un sous-ensemble de sommets deux à deux reliés par des chemins, maximal pour l'inclusion.

Dans l'exemple de graphe donné plus haut, le graphe n'est pas fortement connexe car il n'y a pas de chemin de b à a. En revanche, l'ensemble $\{b, c, d\}$ est une composante fortement connexe.

Pour exprimer la complexité des algorithmes sur les graphes, on utilise abusivement V pour désigner le *nombre de sommets* et E pour désigner le *nombre d'arcs* du graphe dont il est question. En particulier, on a l'inégalité

$$E \leq V^2$$

qui implique notamment que l'on pourra remplacer $\mathcal{O}(\log E)$ par $\mathcal{O}(\log V)$ dans les calculs de complexité que nous ferons. Si le graphe ne contient pas de boucle, alors on a l'inégalité plus fine $E \leq V(V-1)$.

1.2 Graphes non orientés

Lorsque la relation d'adjacence E est symétrique, c'est-à-dire que l'on a un arc entre x et y si et seulement si on a un arc entre y et x, on parle alors de *graphe non orienté*. De façon équivalente, on peut en donner une définition directe en terme de *paires* de sommets plutôt que de couples de sommets.

Définition – graphe non orienté

Un *graphe non orienté* est défini par un ensemble V de *sommets* et un ensemble E de *paires* de sommets appelés *arcs*. On note $x - y$ la présence d'un arc entre les sommets x et y, c'est-à-dire $\{x, y\} \in E$.

Dans le contexte des graphes non orientés, on parle parfois de *nœud* plutôt que de sommet et d'*arêtes* plutôt que d'arcs.

Beaucoup de définitions sur les graphes orientés restent valables, ou se simplifient, sur les graphes non orientés. Ainsi, la notion de chemin reste exactement la même.

Notons cependant qu'il n'y a pas de cycle de longueur 2, car $x - y - x$ n'est pas considéré comme un cycle (l'arc $x-y$ est répété), là où $x \to y \to x$ est un cycle dans un graphe orienté.

La notion de degré est simplifiée, dans la mesure où on ne distingue plus le degré entrant et le degré sortant. Le degré est simplement le nombre de voisins.
On note que le nombre d'arcs est maintenant majoré par $V(V+1)/2$ si on admet les boucles et par $V(V-1)/2$ sinon.

Définition – connexité

Un graphe non orienté $G = (V, E)$ est *connexe* si, pour toute paire de sommets x et y de V, il existe un chemin de x à y. Une *composante connexe* de G est un sous-ensemble de sommets deux à deux reliés par des chemins, maximal pour l'inclusion.

Définition – arbre

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

Il est intéressant de faire ici une comparaison avec la structure d'arbre. Ici, dans le contexte des graphes, on ne distingue pas de sommet particulier qui serait la racine. Si on le fait, on parle alors d'*arbre enraciné*. On n'ordonne pas non plus les arbres qui forment une forêt.

Propriété

Tout arbre qui possède V sommets est composé d'exactement $V - 1$ arcs.

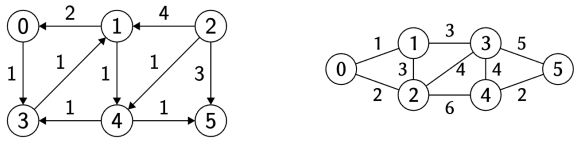
Démonstration. Par récurrence forte sur V.
C'est clair pour $V = 1$.
Soit un graphe connexe acyclique de $V \geq 2$ sommets. Soit v l'un de ses sommets. Il y a au moins un arc issu de v, car G est connexe. Les $k \geq 1$ arcs issus de v relient v à autant de graphes $G_1, \ldots, G_k$ qui sont eux-mêmes connexes et acycliques. Par hypothèse de récurrence, chaque graphe G_i possède V_i sommets et $V_i - 1$ arcs, avec $V = 1 + V_1 + \cdots + V_k$. Par ailleurs, les graphes G_i ne sont pas reliés entre eux, sans quoi il y aurait un cycle dans G. Le nombre d'arcs de G est donc $k + (V_1 - 1) + \cdots + (V_k - 1) = V - 1$. □

1.3 Graphes pondérés

Il est fréquent d'attacher une information aux arcs d'un graphe. On parle alors de *graphe pondéré*. Chaque arc se voit attacher une *étiquette*, qui peut être d'une nature quelconque : un coût, une distance, un caractère, etc.

Exemple

Voici deux exemples de graphes pondérés, l'un orienté et l'autre non orienté, où les étiquettes sont des entiers :



Selon la nature des étiquettes, on parle parfois de *distance* ou encore de *poids* et, en faisant la somme, de la *longueur* d'un chemin ou du *poids total* d'un ensemble d'arcs.

1.4 Graphes bipartis

Dans certains graphes, les sommets se répartissent naturellement en deux ensembles disjoints, avec des arcs uniquement entre les deux. Ainsi, un graphe de publications scientifiques peut distinguer deux sortes de sommets, les chercheurs et les articles, et un arc relie un chercheur avec un article dont il est coauteur. Un autre exemple est le graphe du jeu d'échecs, où les sommets sont les configurations possibles d'une partie en cours. On distingue alors les configurations où c'est aux blancs de jouer et celles où c'est aux noirs de jouer.

Définition - graphe biparti

On dit qu'un graphe $G = (V, E)$ est *biparti* si ses sommets peuvent être partitionnés en deux ensembles disjoints X et Y tels que chaque arc de E a une extrémité dans X et l'autre dans Y, soit, formellement,

$$\text{pour tout } v \in V, \quad (v \in X \text{ et } v \notin Y) \text{ ou } (v \in Y \text{ et } v \notin X),$$

$$\text{pour tout } (u, v) \in E, \quad (u \in X \text{ et } v \in Y) \text{ ou } (u \in Y \text{ et } v \in X).$$

Une telle définition s'applique aussi bien à un graphe non orienté, comme notre exemple du graphe des publications scientifiques, qu'à un graphe orienté, comme notre exemple du graphe du jeu d'échecs. Plus loin dans ce chapitre, on décrit un algorithme de couplage maximum sur un graphe biparti non orienté.

2 Structures de données

On en vient maintenant à la question de représenter un graphe en machine. On se limite ici à des sommets qui sont des entiers, et plus précisément les entiers $0, 1, \ldots, n-1$ pour un graphe possédant n sommets. Nous verrons plus loin que ce n'est pas là une grosse contrainte en pratique.

Le programme suivante contient l'interface d'une structure de graphe orienté, en OCaml et en C, sous la forme d'un type `digraph` (en anglais, un graphe orienté se dit *directed graph*) et de quelques opérations.

Programme – Interface d'une structure de graphe orienté

Les sommets sont les entiers $0, 1, \ldots, n-1$.

En OCaml :

```
type digraph
val create: int -> digraph
val size: digraph -> int (* nombre de sommets *)
val add_edge: digraph -> int -> int -> unit
val has_edge: digraph -> int -> int -> bool
val succ: digraph -> int -> int list
val edges: graph -> (int * int) list
```

En C :

```
typedef struct Digraph digraph;
digraph *digraph_create(int size);
int digraph_size(digraph *g); // nombre de sommets
void digraph_add_edge(digraph *g, int u, int v);
bool digraph_has_edge(digraph *g, int u, int v);
list *digraph_succ(digraph *g, int u);
void digraph_delete(digraph *g);
```

Bien entendu, on pourrait imaginer bien d'autres opérations encore, par exemple pour obtenir le degré entrant, sortant, le nombre total d'arcs, pour supprimer un arc, etc. Cependant, cette interface minimale nous permet déjà de construire un graphe, d'une part, et de le parcourir, au sens de parcourir tous ses sommets (grâce à `size`) et tous les voisins d'un sommet donné (grâce à `succ`). Comme nous le verrons, c'est là tout ce dont nous avons besoin pour écrire efficacement des algorithmes sur les graphes. Pour notre confort, nous ajoutons également dans l'interface OCaml une fonction `edges` qui renvoie l'ensemble des arcs du graphe. Mais il serait possible de la reconstruire à partir des fonctions `size` et `succ`.

Attention

Certains algorithmes sur les graphes, comme chercher un plus court chemin entre deux sommets, s'appliquent sans changement à des graphes infinis. La fonction `succ` est alors la seule description du graphe dont ces algorithmes ont besoin.

2.1 Matrice d'adjacence

Le plus naturel pour représenter un graphe est sans doute une matrice M de booléens, de taille $V \times V$, où l'élément $M_{i,j}$ indique la présence d'un arc entre les sommets i et j. La figure 1 illustre cette représentation.

	0	1	2	3
0	false	true	true	false
1	false	false	true	false
2	true	false	false	false
3	true	false	true	false

FIGURE 1 – Graphe représenté par une matrice d'adjacence.

En OCaml, ce sera donc une valeur de type `bool array array`. En C, ce sera par exemple un tableau statique déclaré comme `bool m[n][n]` pour une certaine valeur n. Mais on peut également allouer un tableau bidimensionnel sur le tas. Sur une telle matrice de booléens, il est immédiat de coder les différentes opérations de l'interface [exercice].

D'une façon évidente, les opérations `has_edge` et `add_edge` sont en $\mathcal{O}(1)$, car on consulte ou on affecte un élément de la matrice. Pour la fonction `succ`, qui renvoie la liste des voisins d'un sommet i, il faut en revanche parcourir toute la ligne correspondante de la matrice, pour un coût total $\mathcal{O}(V)$. En particulier, le sommet i pourrait avoir très peu de voisins, voire aucun voisin, et le calcul de `succ i` aura toujours un coût $\mathcal{O}(V)$. Cette constatation nous conduit à une autre idée.

2.2 Listes d'adjacence

Pour améliorer l'efficacité de la fonction `succ`, on peut avantageusement représenter le graphe par un tableau qui donne, pour chaque sommet i, directement la liste de ses voisins. La figure suivante illustre cette idée.

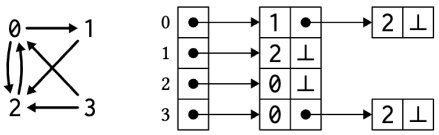


FIGURE 2 – Graphe représenté par des listes d'adjacence.

Avec cette représentation, la fonction succ devient immédiate, en temps constant. En revanche, tester la présence d'un arc ou ajouter un nouvel arc ne se fait plus en temps constant, car il faut parcourir la liste des voisins, avec un coût qui dépend maintenant du degré.

Le programme suivante contient une implémentation OCaml des graphes orientés par listes d'adjacence.

Programme – graphes orientés par listes d'adjacence

```
type digraph = int list array

let create (n: int) : digraph =
  Array.make n []

let size (g: digraph) : int =
  Array.length g

let add_edge (g: digraph) (u: int) (v: int) : unit =
  if not (has_edge g u v) then
    g.(u) <- v :: g.(u)

let has_edge (g: digraph) (u: int) (v: int) : bool =
  List.mem v g.(u)

let succ (g: digraph) (u: int) : int list =
  g.(u)
```

```
let edges (g: digraph) : (int * int) list =
  let l = ref [] in
  for i = 0 to size g - 1 do
    List.iter (fun j -> l := (i, j) :: !l) g.(i)
  done;
  !l
```

Il est intéressant de comparer par ailleurs l'empreinte mémoire de nos deux représentations. Si on prend le cas d'OCaml, où un booléen occupe un mot mémoire, alors le décompte du nombre total de mots mémoire utilisés par chacune des deux solutions est le suivant :

représentation	occupation mémoire
matrice d'adjacence	$1 + V + V^2$ mots
listes d'adjacence	$1 + V + 3E$ mots

Si le graphe est peu dense, avec E petit devant V^2, alors les listes d'adjacence sont plus économes. Mais si en revanche le graphe est dense, avec E de l'ordre de V^2, alors la matrice d'adjacence sera asymptotiquement plus économe d'un facteur trois.

En C, où un booléen occupe un octet, la comparaison est favorable aux matrices d'adjacence encore plus longtemps. Mais il convient de se rappeler que la matrice d'adjacence n'offre pas `succ` en temps constant. Comme souvent, on peut être amenés à faire un compromis entre temps et espace, dans un sens ou dans l'autre.

2.3 Graphes non orientés

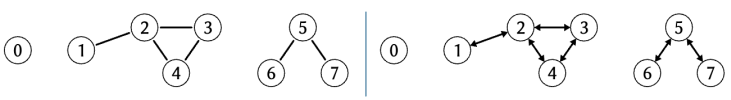
La meilleure façon de représenter un graphe non orienté consiste à réutiliser directement la représentation d'un graphe orienté,

```
type graph = digraph
```

tout en maintenant l'invariant qu'à tout arc $u \to v$ correspond également un arc $v \to u$. On peut le garantir facilement en se donnant une fonction qui ajoute les deux arcs dans le graphe.

```
let add_edge g u v =
  Digraph.add_edge g u v;
  Digraph.add_edge g v u
```

Ainsi, le graphe non orienté à gauche est représenté comme le graphe orienté à droite :



Avec cette représentation, bon nombre d'opérations restent exactement les mêmes, comme tester la présence d'un arc ou obtenir les voisins d'un sommet. Et comme nous le verrons, certains algorithmes sont également inchangés.

Il faut cependant tenir compte du caractère non orienté pour certaines opérations sur les graphes. Compter le nombre d'arcs, par exemple, nécessitera de diviser par deux le total obtenu. De même, pour imprimer les arcs du graphe, on évitera de les imprimer deux fois. Une solution simple consiste à n'imprimer l'arc $u \to v$ que lorsque $u \geq v$.

2.4 Graphes pondérés

Pour représenter un graphe pondéré, qu'il soit orienté ou non, on commence par choisir un type pour les poids. C'est une bonne idée de choisir le type `float` des nombres flottants pour cela. D'une part, cela apporte une grande flexibilité dans l'utilisation : distances euclidiennes calculées avec des racines carrées, temps de trajet exprimés avec des décimales, etc. D'autre part, cela évite de confondre dans le code, par accident, les sommets qui sont des entiers et les poids qui sont des flottants.

Si on adopte une représentation par listes d'adjacence, un graphe pondéré est donc un tableau donnant, pour chaque sommet, la liste de ses voisins avec pour chacun le poids de l'arc correspondant.

```
type wdigraph = (int * float) list array
```

Lorsqu'on accède aux successeurs d'un sommet donné, on récupère donc une liste donnant les arcs sortants avec leur poids.

```
val succ: wdigraph -> int -> (int * float) list
```

On peut également se donner une fonction pour renvoyer le poids de l'arc $u \to v$ lorsqu'il existe.

```
let weight g u v =
  List.assoc v g.(u)
```

À la charge alors de l'appelant de s'assurer qu'il y a bien un arc $u \to v$ dans le graphe. Une autre solution consisterait à renvoyer une valeur particulière lorsque l'arc n'existe pas, comme par exemple la valeur `infinity` de type `float`.