Chapitre 1

Introduction

Sommaire

Informellement, un graphe est une structure permettant de représenter les liens entre divers éléments [1]. Il s'agit d'une structure de données d'une importance capitale en informatique, car on la retrouve dans des multitudes d'applications de natures extrêmement diverses. La Figure 1.1 montre quelques exemples naturels de structures modélisables par des graphes, dont on explique la construction ci-dessous :

— un réseau social (du type Facebook, LinkedIn, Last.fm, …) peut se représenter par un graphe, dont les éléments sont les utilisateurs du réseau et dont les liens relient deux connaissances. Dans certains cas (par exemple Twitter), la relation symétrique d'amitié (A est l'ami de B si et seulement si B est l'ami de A) est remplacée par des liens unidirectionnels, représentant le fait que certains utilisateurs peuvent en "suivre" d'autres. Cette relation est asymétrique, puisque le fait que A suive B n'oblige pas B à suivre A. Dans ce cas, les liens du réseaux sont représentés par des flèches, orientées de A vers B si A suit B, plutôt que par de simples segments ; le graphe est alors dit *orienté*.
— les moyens de transports relient des arrêts ou des stations par des lignes. De la même manière, les réseaux routiers relient des intersections par des routes. On peut selon les cas trouver des mélanges de liens symétriques ou non, selon la possibilité de prendre les transports ou les routes dans les deux sens. De même, pour estimer la distance ou le temps nécessaire pour aller d'un endroit à un autre, il est important de préciser sur chaque lien la distance entre les deux points reliés ou le temps mis pour la parcourir. On parle dans ce cas-là d'un graphe *pondéré*.
— la découpe en modules de grands projets informatiques produit un nombre conséquent de fichiers sources ; pour éviter de recompiler tous les fichiers du projet lorsqu'on modifie seulement certains fichiers, il est utile de construire un graphe de dépendance entre ces sources, dont les éléments sont les fichiers du projet et dans le-

1. Les notions d'"éléments" et de "liens" seront précisées plus loin.

quel on relie un fichier A à un fichier B si le fichier B dépend du fichier A. Ainsi, lorsqu'on devra recompiler A, on détectera qu'il faudra aussi recompiler B, ainsi que les fichiers dépendant de B, et ainsi de suite. Notons que les **indépendances** sont également intéressantes, car on peut compiler les ensembles d'éléments indépendants en parallèle.

— un système de fichiers peut être vu comme une arborescence, où chaque nœud correspond à un répertoire et où une flèche part du nœud A vers le nœud B si le répertoire correspondant à A contient le répertoire correspondant à B. Dans les systèmes d'exploitation dérivés de UNIX, on retrouve des systèmes de fichiers permettant de créer des liens symboliques, et il est donc possible de créer plusieurs chemins vers le même fichier sans dupliquer les données. On se retrouve alors avec un graphe dont la racine est le répertoire racine (C :, /, ... selon l'OS utilisé), dans lequel la présence de **cycles** peut être problématique et qu'il faudra donc détecter (voir plus loin).

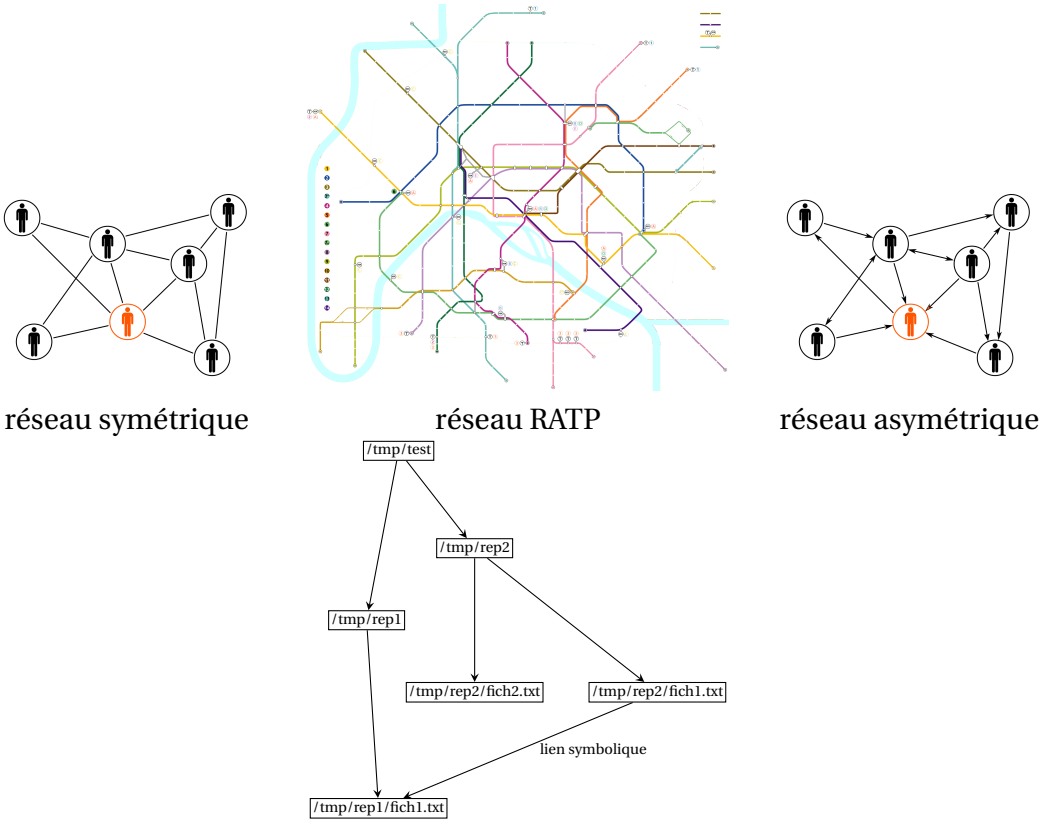


FIGURE 1.1 – Quelques exemples de graphes issus de notre quotidien.

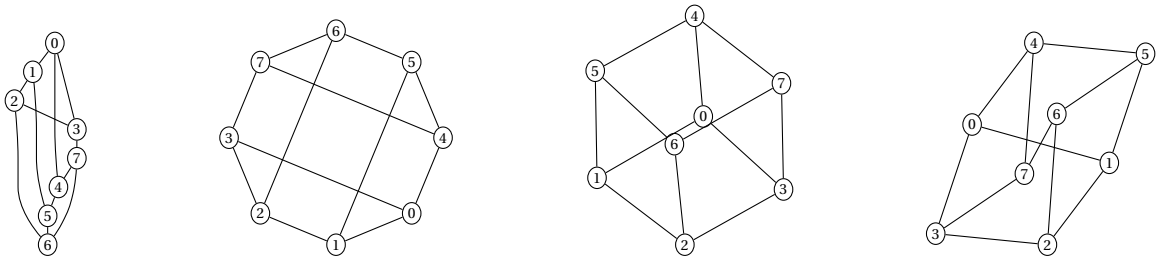
1.1 Concepts de base

Définition 4. Un *graphe* est un couple $G = (V, E)$, où
— V est un ensemble de *sommets* (ou *vertices* en anglais) ;
— $E \subseteq V \times V$ un ensemble d'*arêtes* (ou *edges* en anglais).
On utilisera souvent les notations $V(G)$ et $E(G)$ pour ces deux ensembles. On dira que G est *simple* s'il ne contient ni *arêtes parallèles* (plusieurs arêtes reliant les deux mêmes

sommets), ni *boucles* (une arête reliant un sommet à lui-même).

Les graphes se prêtent naturellement à une représentation graphique qui associe à chaque sommet un point du plan (ou de l'espace) et à chaque arête un trait ou une courbe reliant les deux points du plan correspondant.

Exemple 6. Voici plusieurs représentations du même graphe $G = (V, E)$, où $V = \{0, 1, 2, 3, 4, 5, 6, 7\}$ et $E = \{ \{0, 1\}, \{1, 2\}, \{2, 3\}, \{3, 0\}, \{4, 5\}, \{5, 6\}, \{6, 7\}, \{7, 4\}, \{0, 4\}, \{1, 5\}, \{2, 6\}, \{3, 7\} \}$:



Bien qu'ils soient très utiles, les dessins utilisés pour représenter les graphes ne sont que des **représentations** qui ne doivent pas nous induire en erreur : comme le montre l'exemple 6, peu importe le placement des sommets ou la manière dont sont dessinées les arêtes, seule la structure sous-jacente du graphe est importante.

Définition 5. On dit que deux graphes G et H sont *isomorphes*, ce que l'on note $G \simeq H$, si l'on peut numéroter les sommets de G et de H de manière à ce que les deux ensembles d'arêtes ainsi obtenus coïncident.

Informellement, cela signifie que G et H correspondent au même graphe ; décider si deux graphes sont isomorphes est un problème de complexité inconnue.

On peut associer autant de propriétés que l'on veut aux sommets, en leur donnant par exemple des noms, des couleurs, des poids, ou *orienter* les arêtes qu'on qualifie alors d'*arcs*. Pour l'instant, on se contentera du modèle le plus simple possible, où l'on identifie chaque sommet par un nombre naturel unique, où les graphes sont *non-orientés* et où chaque arête est un simple ensemble de deux sommets.

Définition 6. Une arête $\{u, v\}$ connectant deux sommets u et v d'un graphe est *incidente* à u et à v. Ses *extrémités* u et v sont *adjacentes*, et on dit également que u (ou v) est un *voisin* de v (ou u). Par extension, le *voisinage* d'un sommet u dans un graphe G est l'ensemble des sommets qui lui sont adjacents :

$$N_G(u) = \{v \in V(G) \mid \{u, v\} \in E(G)\},$$

et le cardinal de cet ensemble est le *degré* du sommet v dans G, noté $\deg_G(v)$. Enfin, si deux sommets ne sont pas adjacents, alors on dit qu'ils sont *indépendants*.

Dans ces notations comme dans les suivantes, il nous arrivera souvent de laisser tomber l'indice $_G$ s'il n'y a pas d'ambiguïté possible, et d'écrire par exemple $N(v)$ plutôt que $N_G(v)$.

On s'intéressera régulièrement à un "morceau" particulier du graphe, plutôt que de le considérer dans son entièreté.

Définition 7. Un *sous-graphe* d'un graphe $G = (V, E)$ est un graphe $H = (V', E')$ avec $V' \subseteq V$ et $E' \subseteq E$. On dit d'un sous-graphe qu'il est *induit* par un ensemble $U \subseteq V$ si toutes les arêtes de E reliant deux sommets de U dans G sont présentes dans H, et on désigne ce sous-graphe induit par la notation $G[U]$. De même, un sous-ensemble $F \subseteq E$ d'arêtes de G peut également induire un sous-graphe, que l'on obtient en sélectionnant toutes les arêtes de F et leurs extrémités.

Exemple 7. Reprenons le graphe G du cube donné dans l'exemple 6. L'ensemble $U = \{4, 5, 6, 7\}$ induit le sous-graphe $H = G[U]$ dont les arêtes sont $F = \{\{4, 5\}, \{5, 6\}, \{6, 7\}, \{7, 4\}\}$. Le sous-graphe $H' = G[F]$, induit cette fois par des arêtes plutôt que des sommets, coïncide donc avec H. Enfin, le sous-graphe de G défini par $H'' = (U, F \setminus \{\{4,5\}\}$ est induit par $F \setminus \{\{4,5\}\}$, mais pas par U puisque $\{4,5\} \notin E(H'')$.

Les sous-graphes particuliers suivants reviendront régulièrement dans les applications :

1. un *chemin* dans un graphe G est une séquence $P = (u_0, u_1, u_2, \ldots, u_{p-1})$ de sommets **distincts** où $\{u_i, u_{i+1}\} \in E(G)$ pour $0 \leq i \leq p-2$;
2. un *cycle* dans un graphe G est une séquence $C = (u_0, u_1, u_2, \ldots, u_p)$ de sommets où $\{u_i, u_{i+1 \mod p}\} \in E(G)$ pour $0 \leq i \leq p-1$. Si le cycle ne contient que des sommets et arêtes distinctes, on dira parfois qu'il est *élémentaire* ; mais bien souvent, cette précision sera omise car ce seront généralement les seuls cycles qui nous intéresseront.
3. enfin, on dit d'un graphe $G = (V, E)$ qu'il est *complet* si $E = V \times V$, c'est-à-dire que chaque sommet est adjacent à tous les autres.

Ces graphes, représentés à la Figure 1.2, sont tellement utilisés qu'ils sont souvent désignés par une notation particulière : P_n pour le chemin, C_n pour le cycle, et K_n pour le graphe complet, où n désigne le nombre de sommets. Précisons aussi que la *longueur* d'un chemin (ou d'un cycle) sera souvent importante : cette notion désigne simplement le nombre d'arêtes du chemin (ou du cycle).

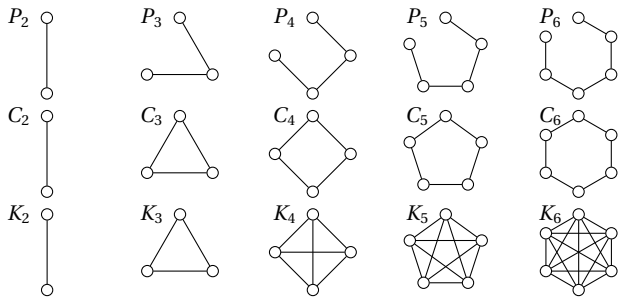


FIGURE 1.2 – Quelques graphes "célèbres" : le chemin, le cycle, et le graphe complet.

On peut facilement déduire le nombre d'arêtes d'un graphe à l'aide des degrés des sommets : si l'on parcourt les sommets du graphe en sélectionnant chaque arête incidente à chacun des sommets, on aura sélectionné chaque arête exactement deux fois, ce qui nous donne la relation suivante :

$$\sum_{v \in V} \deg(v) \quad = \quad 2|E|. \tag{1.1}$$

Par ailleurs, le graphe complet est celui qui contient le plus grand nombre possible d'arêtes : tout couple de sommets y étant relié par une arête, ce graphe en contient $\binom{|V|}{2}$. On peut donc affirmer que pour tout graphe $G = (V, E)$, on a $|E| \leq \binom{|V|}{2}$, et donc que $|E| = O(|V|^2)$, et également que $\log|E| = O(\log|V|)$.

1.2 Implémentation de graphes

Les algorithmes de ce cours s'exprimeront en pseudocode. On supposera tout du long qu'on aura accès à des classes représentant nos graphes, avec plus ou moins d'informations selon les cas, et qu'une certaine interface (c'est-à-dire un certain ensemble de méthodes) nous permettra d'interagir avec eux pour consulter des propriétés de ces graphes ou les modifier. Comme on le verra plus loin, la complexité de ces méthodes dépendra de la représentation choisie pour ces graphes.

Plus précisément, on supposera que les méthodes décrites dans le Tableau 1.1 sont disponibles dans la classe utilisée, qui implémente — pour l'instant — un graphe non orienté, non pondéré, dont les sommets ne sont pas modifiables, et admettant les boucles mais pas les arêtes parallèles. Il se peut, selon les implémentations que nous examinerons, que certaines méthodes soient légèrement différentes ou manquantes, et que d'autres soient ajoutées (comme nous le verrons plus loin avec les graphes orientés ou pondérés).

1.3 Représentations

Comme on le verra plus loin, le choix de la représentation d'un graphe en mémoire aura un impact sur la complexité des algorithmes qu'on se proposera d'implémenter. On examinera ici les deux représentations les plus simples et répandues, mais il en existe d'autres (par exemple les *matrices d'incidence*).

1.3.1 Matrice d'adjacence

Les matrices fournissent une représentation très naturelle des graphes.

Définition 8. La *matrice d'adjacence* $A(G)$ du graphe $G = (V, E)$ contient un 1 en ligne i et en colonne j si l'arête $\{i, j\}$ existe, et un 0 sinon :

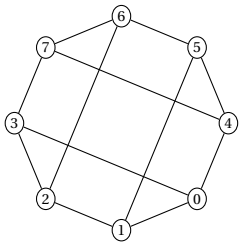
$$A_{ij}(G) = \begin{cases} 1 & \text{si } \{i, j\} \in E, \\ 0 & \text{sinon.} \end{cases}$$

Si le graphe est non-orienté, la matrice d'adjacence est symétrique, et on peut donc se contenter de n'en stocker que le triangle inférieur. Cela nous permet de ne stocker que $n(n+1)/2$ valeurs au lieu de n^2.

Exemple 8. Voici un graphe avec sa matrice d'adjacence, et une version "simplifiée" de cette matrice dont on ne garde que le triangle inférieur :

Nom	**Description**
`ajouter_arete(u, v)`	Ajoute une arête entre les sommets u et v, en créant au besoin ces sommets.
`ajouter_aretes(séquence)`	Ajoute toutes les arêtes de la séquence donnée au graphe.
`ajouter_sommet(sommet)`	Ajoute un sommet au graphe.
`ajouter_sommets(séquence)`	Ajoute tous les sommets de la séquence donnée au graphe.
`aretes()`	Renvoie l'ensemble des arêtes du graphe, représentées par des tuples.
`boucles()`	Renvoie les boucles du graphe, c'est-à-dire les arêtes reliant un sommet à lui-même.
`contient_arete(u, v)`	Renvoie VRAI si l'arête $\{u, v\}$ existe, FAUX sinon.
`contient_sommet(u)`	Renvoie VRAI si le sommet u existe, FAUX sinon.
`degre(sommet)`	Renvoie le nombre de voisins du sommet ; s'il n'existe pas, provoque une erreur.
`nombre_aretes()`	Renvoie le nombre d'arêtes du graphe.
`nombre_boucles()`	Renvoie le nombre d'arêtes de la forme $\{u, u\}$.
`nombre_sommets()`	Renvoie le nombre de sommets du graphe.
`retirer_arete(u, v)`	Retire l'arête $\{u, v\}$ si elle existe ; provoque une erreur sinon.
`retirer_aretes(séquence)`	Retire toutes les arêtes de la séquence donnée du graphe si elles existent ; provoque une erreur sinon.
`retirer_sommet(sommet)`	Efface le sommet du graphe, et retire toutes les arêtes qui lui sont incidentes.
`retirer_sommets(séquence)`	Efface les sommets de la séquence donnée du graphe, et retire toutes les arêtes incidentes à ces sommets.
`sommets()`	Renvoie l'ensemble des sommets du graphe.
`sous_graphe_induit(séquence)`	Renvoie le sous-graphe induit par la séquence de sommets donnée.
`voisins(sommet)`	Renvoie l'ensemble des voisins du sommet donné.

TABLEAU 1.1 – Les méthodes de la classe `Graphe`.



	0	1	2	3	4	5	6	7
0	0	1	0	1	1	0	0	0
1	1	0	1	0	0	1	0	0
2	0	1	0	1	0	0	1	0
3	1	0	1	0	0	0	0	1
4	1	0	0	0	0	1	0	1
5	0	1	0	0	1	0	1	0
6	0	0	1	0	0	1	0	1
7	0	0	0	1	1	0	1	0

	0	1	2	3	4	5	6	7
0	0							
1	1	0						
2	0	1	0					
3	1	0	1	0				
4	1	0	0	0	0			
5	0	1	0	0	1	0		
6	0	0	1	0	0	1	0	
7	0	0	0	1	1	0	1	0

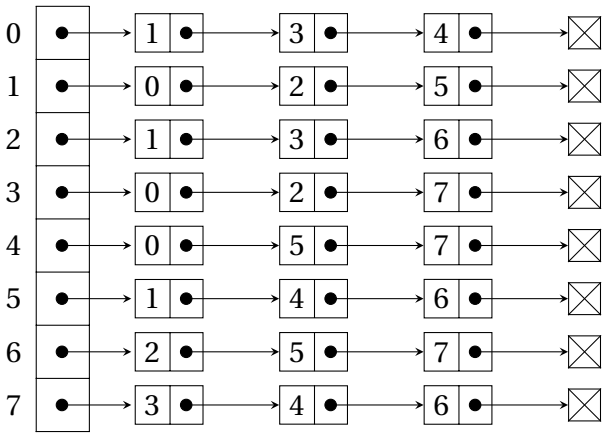
(fin exemple 8)

De même, si le graphe n'est pas pondéré, on peut se contenter d'utiliser un bit par arête pour indiquer si elle est présente ; la concaténation du nombre de lignes de la matrice et des lignes de la matrice simplifiée suffit pour reconstruire le graphe[2].

Les sommets servant également d'indices dans la matrice, les types de graphes que l'on peut stocker à l'aide de cette représentation sont limités. La matrice d'adjacence convient donc bien dans les cas où seule la structure du graphe nous intéresse, mais rien ne nous empêcherait d'associer des propriétés aux sommets stockés, par exemple dans une liste ou un dictionnaire propre au graphe.

1.3.2 Listes d'adjacence

La matrice d'adjacence est une représentation naturelle, mais elle peut gaspiller beaucoup d'espace en mémoire. Par exemple, si l'on veut représenter le chemin P_n, il nous faut $O(n^2)$ cases en mémoire pour le représenter, alors que ce chemin ne contient que n sommets et $n-1$ arêtes. Comme il est fréquent que les graphes rencontrés en pratique contiennent bien moins de n^2 arêtes, on a souvent recours à une représentation plus compacte sous la forme d'une *liste* d'adjacence. Il s'agit d'une liste de n sous-listes, dans laquelle la sous-liste en position i contient tous les voisins du sommet d'indice i. On remarque que chaque arête $\{u, v\}$ est encodée deux fois dans ce format : v est dans la liste des voisins de u, et u est dans la liste des voisins de v. Le graphe de l'exemple 6 s'encoderait donc comme suit :



On remarque que chaque arête $\{u, v\}$ est encodée deux fois dans ce format : v est dans l'ensemble des voisins de u, et u est dans l'ensemble des voisins de v. C'était également le cas pour la matrice d'adjacence.

2. C'est la base du format `graph6`, utilisé par nauty.

Méthode	**Matrice d'adjacence**	**Liste d'adjacence**
`ajouter_arete(u, v)`	$O(1)$ si u et v existent	$O(\deg(u)+\deg(v))$ si u et v existent
`ajouter_aretes(séquence)`	$O(\lvert\text{séquence}\rvert)$	$O(\Delta \times \lvert\text{séquence}\rvert)$
`ajouter_sommet()`	$O(n)$	$O(1)$
`aretes()`	$O(n^2)$	$O(m)$
`boucles()`	$O(n)$	$O(m)$
`contient_arete(u, v)`	$O(1)$	$O(\min(\deg(u),\deg(v)))$
`contient_sommet(u)`	$O(1)$	$O(1)$
`degre(sommet)`	$O(n)$	$O(1)$
`nombre_aretes()`	$O(n^2)$	$O(n)$
`nombre_boucles()`	$O(n)$	$O(m)$
`nombre_sommets()`	$O(1)$	$O(1)$
`retirer_arete(u, v)`	$O(1)$	$O(\deg(u)+\deg(v))$
`retirer_aretes(séquence)`	$O(\lvert\text{séquence}\rvert)$	$O(\Delta \times \lvert\text{séquence}\rvert)$
`retirer_sommet(sommet)`	$O(n^2)$	$O(m+n)$
`retirer_sommets(séquence)`	$O(n^2 \times \lvert\text{séquence}\rvert)$	$O((m+n) \times \lvert\text{séquence}\rvert)$
`sommets()`	$O(n)$	$O(n)$
`sous_graphe_induit(séquence)`	$O(\lvert\text{séquence}\rvert^2)$	$O(\Delta \times \lvert\text{séquence}\rvert^2)$
`voisins(sommet)`	$O(n)$	$O(\deg(sommet))$

TABLEAU 1.2 – Complexité **au pire cas** des diverses opérations selon le type de représentation choisie ($n = |V(G)|, m = |E(G)|$, Δ = le plus grand degré d'un sommet de la séquence fournie).

1.3.3 Stockage de propriétés supplémentaires

Certaines applications nous mènent naturellement à construire des graphes dont les sommets et les arêtes sont des objets plus compliqués que des entiers ou des liens binaires. Par exemple, les sommets pourraient représenter des chaînes de caractères, et les arêtes pourraient posséder des attributs. Il est possible de stocker ces propriétés directement sur les sommets et les arêtes, ou alors de séparer la structure combinatoire du graphe des attributs de ses sommets et de ses arêtes. Dans la suite de ce document, on optera pour cette dernière option : d'abord parce que l'implémentation résultante est plus propre, et ensuite parce que cela simplifie l'analyse de la complexité de nos algorithmes.

1.3.4 Considérations algorithmiques

Le choix de la représentation d'un graphe a des impacts importants sur la consommation en mémoire ainsi que sur le temps de calcul des opérations associées. La matrice d'adjacence possède une consommation en mémoire en $O(|V|^2)$, puisqu'on est toujours obligé de stocker tous les sommets même si aucune arête n'est présente. En revanche, les listes d'adjacence et les dictionnaires d'adjacence consomment un espace mémoire en $O(|V|+|E|)$, puisqu'ils ne stockent que ce qui est nécessaire ($|V|$ clés pour les sommets, et deux fois chaque arête — voir l'explication plus haut). Le Tableau 1.2 donne les complexités de chacune des méthodes du Tableau 1.1.

Exercice 1. Justifiez les complexités obtenues pour la matrice d'adjacence.

Si les complexités obtenues pour la représentation par une matrice d'adjacence sont relativement aisées à obtenir, la représentation sous forme de liste d'adjacence laisse plus de

place à la discussion. Par exemple :

— il serait possible d'implémenter `ajouter_arete(u, v)` en temps $O(1)$ en insérant les sommets u et v à l'extrémité de la liste qui permet de le faire en temps constant (le début pour les listes chaînées, la fin pour les listes Python). Mais alors, on ne vérifierait jamais la présence d'un voisin dans une liste de voisins, et on se retrouverait avec des arêtes multiples et une consommation en mémoire dangereusement élevée ; on suppose donc dans le Tableau 1.2 qu'on fait cette vérification au préalable, d'où la complexité annoncée.

— il peut s'avérer utile d'obtenir les voisins dans un certain ordre, en réalisant par exemple des insertions triées, ce qui nous permettrait ensuite d'utiliser la dichotomie pour implémenter `contient_arete(u, v)` en temps logarithmique.

— ...

Exercice 2. Justifiez les complexités obtenues pour les listes d'adjacence, et essayez de trouver des manières plus rapides de procéder pour certaines de ces méthodes. Expliquez dans quelles mesures vos améliorations auraient un impact positif ou négatif sur d'autres méthodes.

1.3.5 Choix de la représentation

Le Tableau 1.2 montre les complexités des opérations du Tableau 1.1, avec une petite simplification pour l'ajout de sommet : on suppose que le sommet rajouté sera toujours d'indice $|V|+1$, et on omet donc l'implémentation de la méthode `ajouter_sommets(séquence)`. Il en ressort que la liste d'adjacence est un meilleur choix en général, et c'est donc à cette structure qu'on accordera la priorité lors de l'analyse de nos algorithmes. Toutefois, on verra plus loin des algorithmes qui s'exécuteront plus rapidement sur une matrice d'adjacence, ou sur d'autres représentations que l'on détaillera plus tard. Comme souvent en algorithmique, les choix que l'on fera dépendront de l'usage que l'on fera de nos graphes, et il arrivera que l'on ait besoin de construire des variantes des structures présentées (par exemple, représenter les voisins d'un sommet à l'aide d'une liste plutôt que d'un ensemble, rajouter des attributs pour accélérer certaines opérations, ...). Nous examinerons d'autres structures d'encodage de graphes au fur et à mesure de nos besoins.

Ainsi, on ne se fixera pas une implémentation unique sur laquelle on s'attendra à ce que tous nos algorithmes s'exécutent de manière optimale. Quand on dira qu'un algorithme s'exécute par exemple en $O(|V|\log|V|)$, il faudra donc comprendre par là qu'il est possible d'implémenter une classe Graphe de façon à ce que l'algorithme présenté puisse être implémenté en $O(|V|\log|V|)$.

1.4 Visualisation de graphes

Les graphes se prêtant particulièrement bien à la visualisation, il est utile de connaître des outils qui nous permettront de les dessiner. On utilisera le format `dot`, dont la syntaxe est très simple et pour lequel de nombreux outils sont déjà disponibles.

Le format `dot` permet de représenter des graphes très facilement. On les stocke dans un fichier avec l'extension `dot` en suivant la syntaxe suivante :

```
graph G {
```

```
    # liste de sommets avec leurs propriétés
    0;
    1;
    2;
    # ...
    # liste d'arêtes avec leurs propriétés
    0 -- 1;
    1 -- 2;
    3 -- 7;
    # ...
}
```

Il n'est pas obligatoire de spécifier les sommets à l'avance : ils seront implicitement créés lors de la lecture d'une arête qui les relie. D'autres propriétés plus complexes peuvent être spécifiées, on y reviendra en cas de besoin. On peut aussi choisir d'écrire directement le graphe (ou certains de ses sous-graphes) sous la forme de séquences de sommets. Par exemple, un chemin pourra s'écrire `0 -- 1 -- 2 -- 3`, et un cycle `0 -- 1 -- 2 -- 3 -- 0`; le programme interprétant ce graphe calculera l'union des arêtes ou des sous-graphes spécifiés avant de dessiner le résultat.

Les *identifiants* sont des noms permettant de faire référence aux graphes, aux sous-graphes, ainsi qu'aux sommets. Les formats autorisés pour les identifiants sont :

1. des chaînes alphanumériques pouvant contenir des '_' mais ne pouvant pas commencer par un chiffre ;
2. des nombres entiers ;
3. des chaînes délimitées par des guillemets (et pouvant en contenir sous la forme `\"`) ;
4. ou des balises HTML.

On peut également donner des propriétés aux sommets, par exemple sous la forme de texte à afficher. Lors de la construction d'un graphe, il est donc plus propre d'écrire

```
graph G {
    0 [label="bonjour"];
    1 [label="au revoir"];
    0 -- 1;
}
```

plutôt que

```
graph G {
    "bonjour" -- "au revoir";
}
```

car la première option est plus facile à maintenir en cas de changements.

Une fois le graphe stocké dans ce format, on peut utiliser les outils en ligne de commande disponibles sur http://www.graphviz.org/ afin de le dessiner. Graphviz fournit les outils suivants pour générer des dessins à partir d'un fichier au format `dot`, qui seront tracés selon les règles propres à chaque programme si aucune autre instruction n'est donnée dans le fichier :

1. `dot` : dessine les graphes de manière hiérarchique, typiquement utilisé pour les graphes orientés (voir plus loin) ou les arbres.

2. `neato` : utilisé en général pour les graphes de taille raisonnable (moins de 100 sommets).
3. `fdp` : similaire à `neato`.
4. `sfdp` : plus adapté aux grands graphes.
5. `twopi` : dispose un sommet au centre, et les autres sur des cercles concentriques autour de ce centre.
6. `circo` : dispose les sommets du graphe sur un cercle.
7. `osage` : plus adapté aux graphes “en couches”.

Ces descriptions sommaires sont tirées du site de Graphviz; en pratique, on essaiera souvent plusieurs programmes pour voir lequel donne le résultat le plus réussi esthétiquement parlant. Remarquons que rien n'indique dans le fichier comment dessiner ou même positionner les sommets. On peut rajouter ces informations dans le fichier, ou laisser le soin aux programmes sélectionnés de s'en occuper. Dans leur utilisation la plus basique, ces programmes s'invoquent tous comme suit :

```
[nom_programme] -T[format] monfichier.dot [-o sortie.format]
```

Par exemple, le deuxième graphe dessiné dans l'exemple 6 a été obtenu comme suit :

```
circo -Tpng cube.dot -o cube.png
```

Lorsque l'on appelle un de ces programmes de dessin, on peut lui passer des paramètres concernant le graphe entier (avec le préfixe `-G`), concernant ses sommets (avec le préfixe `-N`), ou concernant ses arêtes (avec le préfixe `-E`). On développera ces options dans le restant du cours au fur et à mesure des besoins. Citons déjà les deux suivantes :

— `-Nlabel=''` : retire toutes les étiquettes des sommets;
— `-Nshape=...` : permet de changer la forme des sommets (voir http://www.graphviz.org/content/node-shapes)

Si cela a du sens, on peut également préciser les éléments graphiques dans le fichier `dot`, par exemple pour les fixer quel que soit le programme de dessin utilisé.

Enfin, signalons qu'il est également possible de visualiser les graphes au format `dot` en ligne à l'adresse suivante : http://www.webgraphviz.com/.