Chapitre 11

Graphes non-pondérés

Notions	Commentaires
Graphe orienté, graphe non orienté. Sommet (ou noeud) ; arc, arête. Boucle. Degré (entrant et sortant). Chemin d'un sommet à un autre. Cycle. Connexité, forte connexité. Graphe orienté acyclique. Arbre en tant que graphe connexe acyclique. Forêt. Graphe biparti.	Notation : graphe $G = (S, A)$, degrés $d_+(s)$ et $d_-(s)$ dans le cas orienté. On n'évoque pas les multi-arcs. On représente un graphe orienté par une matrice d'adjacence ou par des listes d'adjacence.

Extrait de la section 3.4 du programme officiel de MP2I : « Structures de données relationnelles ».

Notions	Commentaires
Notion de parcours (sans contrainte). Notion de parcours en largeur, en profondeur. Notion d'arborescence d'un parcours.	On peut évoquer la recherche de cycle, la bicolorabilité d'un graphe, la recherche de plus courts chemins dans un graphe à distance unitaire.
Accessibilité. Tri topologique d'un graphe orienté acyclique à partir de parcours en profondeur. Recherche des composantes connexes d'un graphe non orienté.	On fait le lien entre accessibilité dans un graphe orienté acyclique et ordre.

Extrait de la section 4.5 du programme officiel de MP2I : « Algorithmique des graphes ».

Sommaire

0 Graphes non-orientés

0.0 Introduction

Voici un graphe :

FIGURE 11.1 – Un graphe G_{ex}

Il est composé de **sommets** (0, 1, 2,...) qui peuvent être liés par des **arêtes**. Un enchainement d'arête forme un **chemin** ou un **cycle**.

Remarque.

- C'est une structure très naturelle pour représenter des réseaux ! Par exemple des cartes routières, des connexions entre des serveurs, etc.
- Les graphes ont des applications *très* vastes : une arête entre deux sommets correspond à un lien, une *relation* entre ces deux sommets. On dit que les graphes sont une **structure de données relationnelles**.
- Le problème originel de la théorie des graphes est un problème de réseaux. Il s'agit du problème du pont de Königsberg ; qui demande à déterminer s'il existe un cycle qui passe une et une seule fois par chaque pont [1] dans ce schéma du centre-ville de Königsberg (circa 1735) :

(a) Schéma du centre-ville (b) Représentation en graphe

FIGURE 11.2 – Le problème des ponts de Konigsberg. Chaque rive est un sommet, et chaque pont une arête.

- Un autre problème se modélisant bien comme des graphes est le problème d'attribution de créneau : on considère des activités qui demandent à réserver une même salle, chacune sur un créneau fixé. Deux activités ne peuvent pas réserver la même salle en même temps. On peut représenter cela comme un graphe : chaque activité est un sommet, et deux activités incompatibles sont reliées. On veut donc trouver un ensemble de sommet sans « arête interne » le plus grand possible : on dit que l'on recherche un indépendant maximal.

1. On appelle cela un chemin eulérien. C'est un problème plutôt simple à résoudre.

(a) Des créneaux d'activité (b) Représentation en graphe

FIGURE 11.3 – Indépendant maximal sur un graphe d'intervalle

0.1 Définitions

0.1.0 Vocabulaire

Convention 1.
Soit E un ensemble :

- On note $\mathscr{P}_2(S)$ l'ensemble des parties d'ordre 2 de S, c'est à dire des $\{x, y\} \subseteq S$ (avec $x \neq y$).
- On note $|E|$ le cardinal de E.

Définition 2 (Graphe non-orienté).
Un **graphe non-orienté** est un couple $G = (S, A)$ où :

- S est un ensemble fini d'éléments appelés des **sommets** (ou **noeuds**).
- $A \subseteq \mathscr{P}_2(S)$ est un ensemble d'**arêtes**.

Remarque.

- En anglais, un *sommet* est un *vertex* et une arête une *edge*. Aussi on note parfois les graphes $G = (V, E)$.
- Les arêtes sont symétriques : $\{u, v\} = \{u, v\}$.
- Cette définition interdit les **boucles**, c'est à dire les arêtes d'un sommet vers lui-même. Elle interdit aussi les **multi-arêtes**, c'est à dire une arête qui relie plus de 2 sommets.
- Graphiquement, on représente les sommets avec des ronds et les arêtes avec des traits.
- Les valeurs de S sont une façons d'identifier les sommets, mais aussi une façon de les étiquetter.
- On peut aussi vouloir étiquetter les arêtes : c'est par exemple ce que l'on fera dans les graphes pondérés. Un formalisme agréable pour cela est d'ajoindre au graphe une fonction qui à une arête associe son étiquette.

Convention 3.
On note uv l'arête $\{u, v\}$.

Définition 4 (Vrac de vocabulaire).
Dans $G = (S, A)$ un graphe :

- Pour $a = uv$ une arête de A, on dit que u et v sont les **extrémités** de l'arête. Réciproquement, on dit que l'arête est **incidente** à ses extrémités.
- Pour u et v deux sommets de S, on dit que u et v sont **adjacents** si $uv \in A$.
- Le nombre de sommets $|S|$ est appelé l'**ordre** du graphe.
- Pour u un sommet de S, le nombre $deg(u)$ d'arêtes incidentes à u est appelé le **degré** de u.

Définition 5 (Sous-graphe et graphe induit).
Soit $G = (S, A)$ un graphe.
Pour $S' \subseteq S$ et $A' \subseteq A$, on dit que $G' = (S', A')$ est un **sous-graphe** de G.
Pour $S' \subseteq S$, en posant $A_{|S'} = A \cap \mathscr{P}_2(S')$, on dit que $G_{|S'} = (S', A_{|S'})$ est le **sous-graphe induit** par S' de G.

Exemple.

(a) Le graphe K_6

(b) Un sous-graphe de K_6

(c) Le sous-graphe K_6 induit par $\{0; 2; 4; 5\}$

FIGURE 11.4 – Sous-graphes et sous-graphe induit

Remarque. Une autre transformation usuelle est l'obtention d'un mineur : un mineur d'un graphe est obtenu par répétition d'opérations parmi les 3 suivantes :

- Supprimer un sommet.
- Supprimer une arête.
- Contracter une arête, c'est à dire fusionner ses deux extrémités en un seul sommet et supprimer la boucle crée.

Elle est notamment utile pour caractériser les graphes planaires : un graphe peut-être représenté dans le plan sans que 2 de ses arêtes ne s'intersectent si et seulement si il ne contient ni K_5 ni $K_{3,3}$ comme mineurs (ces deux graphes sont définis plus loin).

0.1.1 Chemins et connexité

Définition 6 (Chemins, cycles).
Dans $G = (S, A)$ un graphe :

- Un **chemin** de longueur p est une suite finie $v_0, v_1, ...v_p$ de $p + 1$ sommets tels que pour tout $i \in [\![0; p[\![$, $v_i v_{i+1} \in A$.
  Si de plus toutes ces arêtes $v_i v_{i+1}$ sont deux à deux distinctes, on dit que le chemin est **simple**.
  Si de plus tous les v_i sont distincts, on dit que le chemin est **élémentaire**.
- Un **cycle** est un chemin simple $v_0, ..., v_p$ tel que $v_0 = v_p$.
  Si de plus $v_0, ..., v_{p-1}$ est élémentaire, on dit que le cycle est **élémentaire**.

Exemple.

FIGURE 11.5 – Un graphe contenant des chemins et des cycles

Remarque.

- La longueur d'un chemin ou d'un cycle est son nombre d'arêtes. Ainsi, la hauteur d'un arbre correspond à cette définition.
- Un cycle est de longueur au moins 3. En effet, un cycle de longueur 2 serait de la forme x, y, x mais ce chemin n'est pas simple.
- Un chemin élémentaire est simple puisque si les sommets sont distincts les arêtes le sont aussi.
- Les terminologies peuvent changer. Notamment, il arrive que « chemin simple » désigne « chemin élémentaire »... Toujours penser à vérifier les définitions utilisées !

Lemme 7 (Facteur et concaténation de chemins).
Soit $G = (S, A)$ un graphe non-orienté et $s_0, ..., s_p$ un chemin. Alors :

- Pour tous $0 \leq i \leq j \leq p$, $s_i, ..., s_p$ est un chemin. Si le chemin initial était simple (resp. élémentaire), celui-ci est également simple (resp. élémentaire).
- Si $s_p, ..., s_q$ est un chemin, alors $s_0, ..., s_p, ..., s_q$ est un chemin.

Démonstration. Brièvement :

- Si un $\forall$ est vrai sur un ensemble, alors il est vrai sur tout sous-ensemble. À appliquer à $s_k s_{k+1} \in A$; ainsi qu'à simple et élémentaire.
- Si un $\forall$ est vrai sur deux ensembles, alors il est vrai sur leur union. De même.

□

Définition 8 (Distance).
Soit G un graphe non-orienté et u et v deux sommets du graphe. On appelle **distance** de u à v, notée $d(u, v)$, le minimum des longueurs des chemins de u à v.

Exemple. Cf exemple précédent.

Remarque. En particulier :

- $d(u, v) = 0$ si et seulement si $u = v$
- $d(u, v) = 1$ si et seulement si $uv \in A$
- $d(u, v) = +\infty$ si et seulement si v n'est pas accessible depuis u

Lemme 9 (Élémentarité d'un chemin minimal).
Soit $G = (S, A)$ un graphe orienté et u et v deux sommets du graphe. Alors tout chemin de u à v de longueur $d(u, v)$ est élémentaire.

Démonstration. Si $d(u, v) = +\infty$, il n'existe aucun tel chemin, donc le résultat est immédiat. On suppose donc $d(u, v) < +\infty$.

Considérons un chemin de u à v de longueur $d = d(u, v)$ non-élémentaire, c'est à dire de la forme $\underbrace{s_0}_{=u}, ..., s_{i-1}, s_i, s_{i+1}, ..., s_{j-1}, s_j, s_{j+1}, ..., \underbrace{s_d}_{=v}$ avec $s_i = s_j$ (et $i < j$).

Figure 11.6 – Un chemin non-élémentaire

En particulier, $s_0, ..., s_i$ est un chemin de u à s_i, et comme $s_i = s_j$, $s_i, s_{j+1}, ..., s_d$ est un chemin de s_i à v. Donc $s_0, ..., s_i, s_{j+1}, ..., s_d$ est un chemin de u à v de longueur $d(u, v) - (j - i) < d(u, v)$. Absurde. □

Remarque.

- Il est très important de savoir formaliser un chemin comme une suite de sommets. Une bonne preuve de graphe, c'est un dessin convainquant accompagné d'une preuve qui formalise rigoureusement ce dessin !
- L'absurde n'est pas nécessaire : on prouve que dans le chemin $s_i = s_j$ implique $i = j$ et l'élémentarité s'en déduit. Il me permet uniquement de gagner quelques mots.

Théorème 10 (Lemme d'extraction de chemin).
Soit $G = (S, A)$ un graphe non-orienté et u et v deux sommets du graphe. S'équivalent :

(0) Il existe un chemin de u à v.

(1) Il existe un chemin simple de u à v.

(2) Il existe un chemin élémentaire de u à v.

Démonstration. (2) $\implies$ (1) et (1) $\implies$ (0) sont immédiats par définition.

Montrons que (0) $\implies$ (2) : s'il existe un chemin de u à v, alors $d(u, v) < +\infty$. Considérons donc un chemin de longueur minimale de u à v : d'après le lemme précédent, il est élémentaire. □

Remarque.

- Le nom de ce lemme est inventé par moi. Je ne lui connais pas de nom officiel ; mais comme ce résultat très souvent le nommer est confortable.
- En pratique, on peut extraire un chemin simple d'un chemin, et un chemin élémentaire d'un chemin simple (d'où le nom que je donne au lemme). Notez toutefois que cette preuve, basée sur le lemme 9, montre (par récurrence) comment extraire un chemin élémentaire d'un chemin, mais pas comment extraire un chemin simple non-élémentaire.

  C'est rarement un besoin que l'on rencontre, donc je ne m'attarde pas dessus. L'idée est la même : si l'on passe deux fois par une même arête, on peut enlever tout ce qu'il y a entre les deux passages (ainsi que le second passage). Continuer récursivement. Servir chaud.

- L'extraction de cycle est plus embêtante à formaliser : on aurait envie de dire « s'il existe un chemin non-vide de u à u alors il existe un cycle »... mais c'est faux :

FIGURE 11.7 – Cas problématique de l'extraction de cycle

Ce cas correspond au cas où le chemin simple que l'on extrait du chemin est... vide, donc pas un cycle. Il faut interdire ce cas pour que l'extraction de cycle fonctionne.

Théorème 11 (Lemme d'extraction de cycle).
Soit G un graphe non-orienté. S'équivalent :

(0) Il existe un chemin d'un sommet u à lui-même dans lequel il existe une arête empruntée une seule fois.

(1) G contient un cycle.

(2) G contient un cycle élémentaire.

Démonstration. Laissée en TD. Vous en prouverez aussi une variante en MPI. □

Définition 12 (Acyclicité).
Un graphe est dit **acyclique** s'il ne contient pas de cycle. On parle aussi de **forêt**.

Définition 13 (Composantes connexes).
Dans un graphe G, la relation d'accessibilité, c'est à dire la relation binaire « v est accessible depuis u », est une relation d'équivalence. Ses classes d'équivalence sont appelées les **composantes connexes**.
En particulier, les composantes connexes partitionnent les sommets du graphe, et toute arête relie deux sommets de la même composante.
Si u est un sommet, on note C_u la composante connexe de u.

Démonstration. Pourvons brièvement que la relation d'accessibilité est bien d'équivalence :

- Tout sommet est accessible depuis lui-même par un chemin de longueur 0. D'où la réflexivité.
- Si v est accessible depuis u, alors il existe $s_0, ..., s_p$ un chemin de longueur p de $u = s_0$ à $v = s_p$. Mais comme pour tout arête xy, $xy \in A \iff yx \in A$, $s_p, ..., s_0$ est aussi un chemin de v à u. D'où la symétrie.
- Si v est accessible depuis u et w depuis v, alors il existe $s_0, ..., s_p$ un chemin de $u = s_0$ à $v = s_p$ et $t_0, ..., t_q$ de $v = t_0$ à $w = t_q$. Mais comme $s_p = t_0$, $s_0, ..., s_p, t_1, ..., t_q$ est un chemin de u à w. D'où la transitivité.

FIGURE 11.8 – Transitivité de l'accessibilité

□

Exemple. Le graphe initial de ce cours, G_{ex}, est partitionné en deux composantes connexes.

Définition 14 (Graphe connexe).
Un graphe est dit **connexe** s'il possède une seule composante connexe.

0.2 Graphes non-orientés remarquables

Certains graphes ou familles de graphes sont des exemples classiques. On nomme :

- P_n le chemin élémentaire de longueur n.

(a) P_1 (b) P_3 (c) P_6

FIGURE 11.9 – Chemin élémentaire P_n

- C_n le cycle élementaire de longueur n :

(a) C_3 (b) C_5 (c) C_8

FIGURE 11.10 – Cycle C_8

- K_n la **clique** (aussi appelé **graphe complet**) à n sommets, c'est à dire le graphe d'ordre n où toutes les arêtes xy possibles existent :

(a) K_3 (b) K_4 (c) K_5

FIGURE 11.11 – Clique K_n

- $K_{p,q}$ la **clique bipartie** (aussi appelé **graphe complet**) à p et q sommets, c'est à dire le graphe d'ordre où les sommets sont partitionnés en deux parties de p et q avec toutes les arêtes possibles entre les parties et aucune au sein d'un groupe :

(a) $K_{1,3}$ (b) $K_{3,3}$ (c) $K_{3,4}$

FIGURE 11.12 – Clique $K_{p,q}$

- Le **graphe de Petersen** est un contre-exemple à de *très* nombreuses conjectures sur les graphes. Il est donc très utile à avoir en tête :

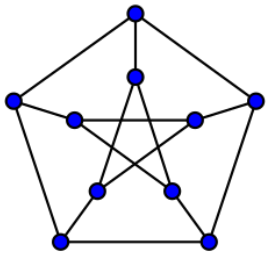


FIGURE 11.13 – Graphe de Petersen (src : Wikipédia)

0.3 Propriétés combinatoires

Dans toute cette sous-section, n désigne le nombre de sommets du graphe étudié.

0.3.0 Liens entre $|S|$ et $|A|$

Proposition 15 (Nombre maximal d'arêtes).
Dans $G = (S, A)$ un graphe non-orienté :

$$|A| \leq \frac{n(n-1)}{2}$$

Démonstration. $|\mathscr{P}_2(S)| = \binom{n}{2} = \frac{n(n-1)}{2}$ □

Proposition 16 (Nombre d'arêtes et degrés).
Dans $G = (S, A)$ un graphe non-orienté :

$$|A| = \frac{\sum_{s \in S} deg(s)}{2}$$

Démonstration. Informellement : quand on somme les degrés, on compte chaque arête exactement deux fois (une fois par sommet incident).

Formellement : posons $\mathbb{1}_{a,s}$ l'indicatrice de « s est incident à a ». Alors :

$$\begin{aligned}
\sum_{s\in S} deg(s) &= \sum_{s\in S}\sum_{a\in A} \mathbb{1}_{a,s} \\
&= \sum_{a\in A}\sum_{s\in S} \mathbb{1}_{a,s} && \text{car les sommes sont finies} \\
&= \sum_{a\in A} 2 && \text{car chaque arête a exactement 2 sommets incidents} \\
&= 2|A|
\end{aligned}$$

□

0.3.1 Bornes de connexité et d'acyclicité

Proposition 17 (Connexité et nombre d'arêtes).
Un graphe non-orienté connexe d'ordre n possède au moins $n-1$ arêtes.

Démonstration. Par récurrence sur n.

- Initialisation : C'est immédiat pour le graphe vide ainsi que pour les graphes à 1 sommet.
- Hérédité : Supposons la propriété vraie jusqu'au rang $n > 1$ exclu, montrons-la vraie au rang n. Soit $G = (S, A)$ un graphe non-orienté connexe avec $|S| = n$.
  Comme G est connexe, pour tout $s \in S$, $deg(s) > 0$ (un sommet de degré 0 est isolé donc non connecté au reste du graphe). Distinguons maintenant deux cas :
  - Si il existe $s_0 \in S$ de degré 1 : on va se ramener à $G_{|S\setminus\{s_0\}}$. En effet, si celui-ci est connexe, il a $n-1$ sommets et $|A|-1$ arêtes donc lui appliquer l'HR donne $|A|-1 \geq (n-1)-1$ qui permet de conclure.

FIGURE 11.14 – $G_{|S\setminus\{s_0\}}$

    Montrons donc que $G_{|S\setminus\{s_0\}}$ est connexe (pour lui appliquer l'HR). Soient u et v deux sommets de G distincts de s_0. Comme G est connexe (et par lemme d'extraction), considérons un chemin élémentaire de u à v et montrons que ce chemin n'utilise pas s_0 :
    - Il n'en est pas une extrémité car u et v sont distincts de s_0.
    - Il n'en est pas un sommet intermédiaire : notons t l'unique voisin de s_0. Si s_0 est un sommet intermédiaire du chemin élémentaire de u à v, alors l'arête menant à $s0$ est ts_0 et l'arête suivante est s_0t donc le chemin n'est pas élémentaire. Contradiction.

    Aussi, tout chemin de u à v dans G est un chemin de u à v dans $G_{|S\setminus\{s_0\}}$. Donc la connexité de G implique celle de $G_{|S\setminus\{s_0\}}$. On conclut comme indiqué au début de ce point.
  - Sinon, pour tout $s \in S$ on a $deg(s) \geq 2$. Donc :

$$n = \frac{1}{2}\sum_{s\in S} \leq \frac{1}{2}\sum_{s\in S} deg(s) = |A|$$

□

Lemme 18.
Soit $G = (S, A)$ un graphe non-orienté non-vide dont tous les sommets sont de degré au moins 2. Alors il contient un cycle.

Démonstration. Idée de la preuve : on part d'un sommet, et on avance dans le graphe (il n'y a pas de cul de sac par hypothèse). Au bout d'un moment, on repasse sur un sommet déjà visité et on en déduit un cycle.

FIGURE 11.15 – La suite (s_i) construite et le cycle atteint

On définit la suite (s_i) comme suit : s_0 est un sommet de S fixé, s_1 un voisin de s_0, et pour $i > 1$, s_{i+1} est un voisin de s_i distinct de s_{i-1} (possible car $deg(s_i) \geq 2$). Comme S est fini, il existe des sommets qui se répètent dans cette suite. Notons j l'indice de la première répétition, c'est à dire le premier indice de la suite tel qu'il existe $i < j$ tel que $s_i = s_j$. Alors $s_i, ..., s_j$ est un chemin de s_i à s_i par construction.

Montrons que ce chemin est simple[2] [3]. S'il ne l'est pas, une arête $s_k s_{k+1}$ est empruntée deux fois, avec $k + 1 \leq j$. Mais alors on passe deux fois par s_k... alors que s_j correspond par définition à la première répétition. Contradiction.

FIGURE 11.16 – La contradiction

On a donc trouvé un chemin simple de longueur > 1 (car $i < j$) de s_i à s_i, c'est à dire un cycle. □

Proposition 19 (Borne supérieure d'acyclicité).
Un graphe non-orienté acyclique non-vide d'ordre n possède au plus $n - 1$ arêtes.

Démonstration. Procédons par récurrence sur n :

- Initialisation : Pour $n = 1$, c'est immédiat.
- Hérédité : Supposons la propriété vraie jusqu'au rang $n > 1$ exclu, montrons-la vraie au rang n. Soit G un graphe non-orienté acyclique d'ordre n.

  D'après le lemme précédent, G possède un sommet s_0 de degré au plus 1. Considérons à nouveau $G_{|S\setminus\{s_0\}}$: tout cycle du graphe induit est un cycle de G, donc $G_{|S\setminus\{s_0\}}$ est acyclique. Donc par HR ce graphe induit a au plus $n-2$ arêtes. Or, $deg(s_0) \leq 1$ donc G a au plus une arête de plus que $G_{|S\setminus\{s_0\}}$, d'où $|A| \leq n - 1$.

2. On ne peut pas juste en extraire un chemin simple : l'extraction pourrait extraire le chemin vide qui est simple mais n'est pas un cycle...
3. Je n'ai pas formalisée l'extraction de cycle, donc on doit faire sans !

FIGURE 11.17 – $G_{|S\setminus\{s_0\}}$

□

0.3.2 Liens avec les arbres

Les arbres au sens du cours sur les arbres ne sont pas stricto sensu des graphes non-orienté (ils ont une notion « de haut en bas »). Toutefois, il est très naturel de vouloir transformer considérer un arbre comme un graphe.

Définition 20 (Arbre non-enraciné).
On appelle **arbre non-enraciné** un graphe (non-orienté) acyclique connexe.

Remarque.

- Parfois, on appelle *arbre* un graphe (non-orienté) acyclique connexe et *arbre enraciné* un arbre au sens du cours sur les arbres. Encore une fois, vérifiez les définitions avec lesquelles vous travaillez !
- Un arbre non-enraciné est un arbre dont on a oublié qui est parent et enfant sur chaque arête, ainsi que l'ordre gauche-droite éventuel sur les enfants.
- Réciproquement, on peut enraciner un arbre non-enraciné en « attrapant un sommet et soulevant » (c'est à dire en faisant un parcours depuis ce sommet !).

Dans la suite de cours, tant qu'il n'y a pas de risque de confusion, j'appelerais « arbre » un graphe acyclique connexe.

Théorème 21 (Caractérisations d'un arbre non-enraciné).
Soit $G = (S, A)$ un graphe non-orienté d'ordre n. S'équivalent :

(i) G est acyclique et connexe (c'est à dire est un arbre)

(ii) G est acyclique et possède $n - 1$ arêtes.

(iii) G est connexe et possède $n - 1$ arêtes.

(iv) G est minimalement connexe, c'est à dire qu'on ne peut lui enlever une arête sans rompre la connexité.

(v) G est maximalement acyclique, c'est à dire qu'on ne peut lui rajouter une arête sans rompre l'acyclicité.

(vi) Pour tous u et v sommets, il existe un unique chemin élémentaire de u à v.

Démonstration.

- $(i) \implies (ii)$ et (iii) est immédiat avec les propriétés précédentes.
- Montrons $(ii) \implies (i)$: soit $G = (S, A)$ un graphe acyclique à $n - 1$ arêtes. Notons r son nombre de composantes connexes, et $n_0, ..., n_{r-1}$ les nombres de sommets de ces composantes. Chacune d'entre elles est connexe et acyclique, donc a $n_i - 1$ arêtes. D'où :

$$\begin{aligned} n-1 &= |A| \\ &= \sum_{i=0}^{r-1}(n_i - 1) && \text{les arêtes sont partitionnées entre les CC qui sont des arbres} \\ &= \left(\sum_{i=0}^{r-1}\right) - r \\ &= n - r && \text{les sommets sont partitionnés entre les CC} \end{aligned}$$

D'où $r = 1$: le graphe est connexe.

- Montrons $(iii) \implies (i)$. Soit $G = (S, A)$ un graphe connexe à $n-1$ arêtes. Supposons par l'absurde qu'il possède un cycle $s_0, ..., s_{p-1}, \underbrace{s_p}_{=s_0}$. Alors supprimer l'arête s_0s_1 ne déconnecte pas le graphe : on peut la remplacer par le chemin $s_0, s_{p-1}, s_{p-2}, ..., s_1$; qui ne contient pas s_0s_1 puisqu'un cycle est simple.

Figure 11.18 – Un cycle « propose » deux chemins

Donc le sous-graphe obtenu en supprimant s_0s_1 est connexe à $n-2$ arêtes : absurde.

Nous avons montré l'équivalence des trois premiers points. Les points (iv) et (v) ont également presque déjà été faits :

- $(iii) \implies (iv)$: c'est exactement la preuve fait pour $(iii) \implies (i)$.
- $(iv) \implies (v)$: un graphe minimalement connexe est acyclique (sinon on peut enlever une arête du cycle, cf $(iii) \implies (i)$). Il est donc connexe est acyclique, donc a $n-1$ arêtes : donc ajouter une arête lui donnerait $> n-1$ arêtes ce qui est impossible pour un graphe acyclique.
- $(v) \implies (i)$: un graphe G maximalement acyclique est connexe car sinon on pourrait connecter deux composantes par une arête sans créer de cycle.

Terminons avec l'équivalence entre les 5 premiers points et (vi) :

- $(i) \implies (vi)$: comme un arbre est connexe, il existe un chemin entre toute paire de sommets. Supposons par l'absurde qu'il existe deux sommets u et v avec deux chemins élémentaires distincts : ces deux chemins sont distincts par au moins un sommet donc au moins une arête. En particulier, on en déduit un chemin de u à u dans lequel une arête est empruntée une seule fois, donc un cycle [4] : absurde puisqu'un arbre est acyclique.
- $(vi) \implies (v)$: un tel graphe est connexe. Mais il est de plus minimalement connexe puisqu'ôter une arête uv déconnecte u de v (cette arête est un chemin élémentaire de u à v donc le seul).

□

Remarque. Ces six points sont tous utiles !

- Le point (v) mène à l'algorithme de Kruskall (MPI) et le point (iv) à l'algorithme reverse-delete (égalament de Kruskall, MPI).
- Le point (vi) assure que l'on peut enraciner un arbre depuis n'importe quel sommet.
- Les points (ii) et (iii) sont très pratiques pour analyser la complexité d'un algorithme sur des arbres.

4. Finalement, l'extraction de cycle me sert à rendre cette preuve plus agréable...

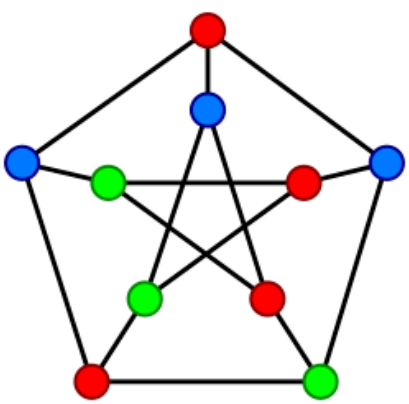
0.4 (Mi-HP) Coloration de graphes

0.4.0 Degré chromatique

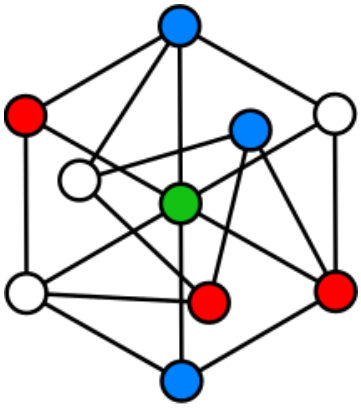
Définition 22 (k-coloration de graphe).
Soit $G = (S, A)$ un graphe. Une k-coloration de G est une application $c : S \mapsto [\![0; k[\![$ telle que pour tout $uv \in A$, $c(u) \neq c(v)$.
Un graphe qui admet une k-coloration est dit k-coloriable.

Remarque.

- Autrement dit, une k coloration donne une couleur à chaque sommet (avec k couleurs autorisées au maximum) de sorte à ce que deux sommets adjacents ne soient pas de la même couleur.
- On est pas obligé d'utiliser les k couleurs pour k colorier, on peut en utiliser moins.
- Les différents composantes connexes sont indépendantes pour le coloriage (colorier un sommet d'une composante n'aura aucun impact sur les couleurs possibles des sommets d'une autre). On suppose donc souvent qu'un graphe est connexe quand on le colorie.



(a) 3-coloration du graphe de Petersen



(b) 4-coloration d'un graphe (graphe de Golomb)

FIGURE 11.19 – Coloration de graphe

Définition 23 (Degré chromatique d'un graphe).
Le degré chromatique d'un graphe G, noté $\chi(G)$, est le plus petit k tel qu'il existe une k-coloration du graphe.

Exemple. Pour les deux graphes précédents, le coloriage donné correspondait au nombre chromatique.

Proposition 24.
Le nombre chromatique d'un graphe est supérieur ou égal à celui de tous ses sous-graphes.

Démonstration. Un k-coloriage d'un graphe peut-être appliqué à chacun des sous-graphes (en ignorant les éventuels sommets absents des sous-graphes) et reste valide puisque les arêtes des sous-graphes sont des arêtes du graphe. □

Proposition 25 (Degré chromatique des graphes usuels).
Pour les graphes usuels, en ignorant les graphes à 1 sommet ou moins :

Clique : $\chi(K_n) = n$

Cycle élem. : $\chi(C_n) = \begin{cases} 2 & \text{si } n \text{ est pair} \\ 3 & \text{sinon} \end{cases}$

Chemin élém. : $\chi(P_n) = 2$

Étoile : $\chi(E_n) = 2$

Arbres : $\chi = 2$

Démonstration.
- Dans une clique, tous les sommets sont adjacents donc doivent avoir des couleurs deux à deux distinctes.
- $\chi(C_n) \geq 2$ car il y a deux sommets adjacents. Dans le cas où le cycle est de longueur paire, le coloriage qui alterne est valide donc le $\chi = 2$. Dans le cas où la longueur est impair, ce coloriage n'est pas valide mais c'est le seul 2-coloriage possible. On exhibe à la place un 3-coloriage et conclut.
- Cas particulier des arbres.
- Idem.
- On enracine l'arbre, et on donne une couleur aux étages pairs et une autre aux étages impairs.

□

Définition 26 (Degré maximal d'un graphe).
On note $\Delta(G)$ le degré maximal d'un graphe G.

Proposition 27.
Tout graphe G est $(\Delta(G)+1)$-coloriable.

Démonstration. On colorie les sommets les uns après les autres, dans un autre arbitraire. Quand on colorie un sommet, il y a au plus Δ couleurs déjà utilisées pour ses voisins : il en reste donc une disponible pour le sommet lui-même. □

Définition 28 (Clique number).
Pour G un graphe, on dit $\omega(G)$ l'ordre de la plus grande clique qui est un sous-graphe de G.

Proposition 29 (Bornes sur χ).
Pour un graphe G :

$$\omega(G) \leq \chi(G) \leq \Delta(G) + 1$$

Démonstration. On a déjà prouvé les deux inégalités. □

0.4.1 (HP) Cas des graphes planaires

Définition 30 (Graphe planaire).
Un graphe est dit planaire s'il peut être représenté dans le plan sans que deux de ses arêtes ne se croisent.

Exemple.
- Le graphe de Petersen est planaire.

- K_4 est planaire (il suffit de faire passer une des « diagonales » du carré par l'extérieur et l'autre par l'intérieur).
- K_5 n'est pas planaire.
- $K_{3,3}$ non plus (c'est l'énigme des trois maisons).

Avant de parler coloriage, mentionnons un théorème qui caractérise les graphes planaires à l'aide de la notion de *mineur* (que nous avons mentionnée avec les sous-graphes et les graphes induits) :

Théorème 31 ((très HP) Théorème de Kuratowski-Wagner).
Un graphe est planaire si et seulement si il ne contient ni $K_{3,3}$ ni K_5 comme mineurs.

Démonstration. Ahahahahaha. Non. □

Théorème 32 (Théorème des 4 couleurs).
Tout graphe planaire est 4-coloriable.

Démonstration. Non, mais voici l'idée :

- Dans tout graphe planaire il existe un sommet de degré au plus 5. On en déduit un algorithme pour 6-colorier un graphe planaire (enlever ce sommet, 6-colorier le graphe induit obtenu, remettre le sommet et lui donner une couleur que ses 5 voisins n'ont pas).
- À l'aide de la méthode des chaines de Kempe[5], modifier l'algorithme précédent pour que 5 couleurs suffisent.
- Écrire un programme qui trouve les situations critiques « minimales » où la méthode des chaines de Kempe aboutit à 5 couleurs et non 4 : tous les autres cas sont 4-coloriables. Ensuite, 4-colorier chacune des situations minimales gênantes et conclure (on trouve de 600 à 1500 de situations critiques, selon l'efficacité du premier programme) : cette étape-ci est également faite par ordinateur.

□

Remarque.

- Le théorème des 4 couleurs a une importance historique en informatique, car il s'agit de l'une des premières utilisations d'un ordinateur pour mener à terme une preuve. Cela a mené à des débats sur le recevabilité de la preuve.
- La preuve a depuis été formalisée en Rocq, un assistant de preuve (c'est à dire un programme qui vérifie si une preuve est correcte). Autrement dit, on a fournit les codes, une preuve des codes, et Rocq a validé que cette (longue) preuve de (longs) programmes est correcte.
- On présente généralement ce théorème en termes de cartes : on peut colorier des régions connexes d'une carte avec 4 couleurs sans que deux régions adjacentes n'aient la même couleur (l'adjacence ne peut pas être réduite à un point).

Remarque. Le théorème des 4 couleurs est une implication, pas une équivalence : $K_{3,3}$ est 2-coloriable mais n'est pas planaire.

0.4.2 2-coloration de graphe

Déterminer si un graphe est 1-coloriable est simple : il faut et suffit qu'il soit entièrement déconnecté. Tester si un graphe est 2-coloriable est également simple : il n'y a pas de choix à faire !

Proposition 33.
Soit G un graphe est s_0 un sommet du graphe. Alors G est deux coloriable si est seulement si $v \mapsto d(u,v) \ mod \ 2$ est un 2-coloriage.

5. Pour plus d'informations, allez voir l'article Wikipédia sur le théorème des 5 couleurs.

Démonstration. La réciproque est immédiate. Montrons l'implication directe.

Considérons un 2-coloriage de G. Quitte à échanger les couleurs, supposons que s_0 soit colorié avec la couleur 0. Ses voisins ont donc la couleur 1. Les sommets à distance 2 sont voisins des sommets à distance 1 (qui sont de couleur 1) : ils sont donc de couleur 0. Et ainsi de suite : on montre par récurrence qu'un 2-coloriage de G est celui annoncé par la propriété. □

Corollaire 34.
Tester si un graphe est 2-coloriable se fait en temps linéaire en la taille du graphe.

Démonstration. Corollaire du parcours en largeur (qui calcule les distances). □

En fait, les graphes 2-coloriables correspondent à une famille de graphes particulières :

Proposition 35.
Un graphe est deux coloriable si et seulement il est biparti.

Démonstration. Cf fin du cours. □

En particulier, on retrouve le fait que les arbres sont 2-coloriables.

0.4.3 (MPI) 3-coloration de graphe : NP-difficile

1 et 2 coloriables étaient très simples... mais savoir si un graphe est 3 coloriable est (très) compliqué ! On ne sait pas le faire en temps polynomial, et on pense que cela est impossible !!

En MPI, vous prouverez cela en prouvant que :

Théorème 36.
Le problème de savoir si un graphe est 3-coloriable est NP-Complet.

Démonstration. MPI. Réduction depuis, probablement, 3-SAT. □

Remarque. En fait, ce que ce théorème dit, c'est que le problème de 3-coloriage a deux grandes qualités :

- On peut « encoder » dedans beaucoup d'autres problèmes. Par exemple, il existe une façon de transformer des instances de SubsetSum[6] en des graphes tels que 3-colorier ces graphes corresponde à une solution de l'instance.
- Et malgré cette grande force d'encodage, il est simple de vérifier si un coloriage proposé est une solution valide ou non.

Remarque. La notion de NP-complétude est au programme de MPI, pas du tout de MP2I. Ce petit morceau de cours a juste pour but de vous donner un avant goût !

6. Étant donnés S ensemble d'entiers positifs et t, trouver s'il existe une partie de S qui se somme à t.

1 Graphes orientés

1.0 Définitions

1.0.0 Vocabulaire

Dans un graphe orienté, les arêtes ne sont plus à double sens. C'est assez classique pour un problème de réseau ou de dépendances.

Définition 37 (Graphe orienté).
Un **graphe orienté** est un couple $G = (S, A)$ où :

- S est un ensemble fini d'éléments appelés des **sommets** (ou **noeuds**).
- $A \subseteq \{(x, y) \in S \times S \, tq \, x \neq y\}$ est un ensemble d'**arcs**.

Remarque.

- Cette fois-ci, le sens des arêtes compte ! En effet, l'arc (la paire) (x, y) est distincte de l'arc (la paire) (y, x).
- Les boucles sont toujours interdites : la définition interdit explicitement des (x, x).
- Graphes induits et sous-graphes sont définis comme précédemment.

Proposition 38.
Un graphe orienté d'ordre n a au plus $n(n-1)$ arcs.

Démonstration. Pour définir un arc, il faut et suffit de choisir le sommet de départ de l'arc (n choix) puis celui d'arrivée (qui doit être distinct, donc $n-1$ choix). □

Définition 39 (Degré orienté).
Dans $G = (S, A)$ un graphe orienté, pour $s \in S$ un sommet, on appelle :

- **degré entrant** de s, noté $deg_-(s)$, le nombre d'arcs qui vont *vers* s ; autrement dit le nombre d'arcs *us* du graphe.
- **degré sortant** de s, noté $deg_+(s)$, le nombre d'arcs qui *partent* de s ; autrement dit le nombre d'arcs sv du graphe.
- **degré total** de s, noté $deg(s)$, est $deg(s) = deg_-(s) + deg_+(s)$.

Remarque. Les $+$ et $-$ des degrés orientés sont à comprendre en termes de « contribution au reste du graphe » : le degré sortant se note avec un plus car il « donne » au reste du graphe en y menant, alors que le degré entrant « prend » au reste du graphe.

Proposition 40.
Dans $G = (S, A)$ un graphe orienté :

$$\sum_{s \in S} deg_+(s) = \sum_{s \in S} deg_-(s) = |A|$$

Démonstration. Cela revient à dénombrer les arêtes en comptant leur source ou bien leur destination. □

1.0.1 Chemins orientés

Définition 41 (Chemin orienté, circuit).
On définit un **chemin orienté** comme précédemment : $v_0, ..., v_p$ est un chemin de longueur p si et seulement si pour tous $i \in [\![0; p[\![$, $v_i v_{i+1} \in A$.
Un chemin orienté simple ou élémentaire est défini comme précédemment. Un **circuit** est un cycle orienté et est défini comme précédemment (idem pour les circuit élémentaires).

Remarque.

- On peut avoir des circuits de longueur 2, puisque l'arc aller et l'arc retour sont distincts :

FIGURE 11.20 – Un circuit de longueur 2

- La relation d'accessibilité n'est pas symétrique : il peut y avoir un chemin de u à v mais pas de v à u :

FIGURE 11.21 – Accessibilité à sens unique

Exemple.

- Le graphe orienté ci-dessous n'a pas de circuit :

FIGURE 11.22 – PAS un cycle orienté

- On dit parfois qu'un graphe sans circuit est **acyclique**[7] ; mais cela peut mener à confusion (cf exemple précédent). Aussi je vais éviter d'utiliser ce terme dans le cas orienté.
- Le graphe des dépendances d'un mode d'emploi (ou d'une compilation ;)) doit être sans circuit ; puisqu'un circuit correspondrait à une étape e_0 qui dépend d'une étape e_1 qui... qui dépend de e_p qui dépend de e_0, donc e_0 se requiert elle-même.
- Plus généralement, le graphe des successeurs immédiats dans une relation d'ordre est sans circuit.

Définition 42 (Distance orienté).
On peut définir la distance $d(u, v)$ comme précédemment.

7. C'est ce que l'on a fait dans les rappels sur les relations d'ordre !

Remarque. Notez que cette fois-ci, elle n'est plus symétrique !

Théorème 43 (Extraction de chemins orientés).
Soit $G = (S, A)$ un graphe orienté et u et v deux sommets du graphe. S'équivalent :

(0) Il existe un chemin orienté de u à v.

(1) Il existe un chemin orienté simple de u à v.

(2) Il existe un chemin orienté élémentaire de u à v.

Démonstration. Comme en non-orienté : un chemin orienté de longueur minimale est élémentaire. □

Théorème 44 (Extraction de cycle orienté).
Soit $G = (S, A)$ un graphe orienté. S'équivalent :

(0) Il existe un chemin orienté d'un sommet u vers lui-même.

(1) Il existe un circuit.

(2) Il existe un circuit élémentaire.

Démonstration. Laissée en exercice. □

Remarque.

- Ici, il n'y a plus besoin de l'hypothèse « chemin de u vers lui-même *qui n'emprunte pas deux fois la même arête* » : cette hypothèse servait à interdit « l'aller-retour » (qui n'est pas un cycle non-orienté), mais cette situation est un cycle orienté.
- En fait, dans l'extraction de circuit, la seule situation à éviter est de « faire un tour de plus ».

1.1 Études des composantes fortement connexes

Définition 45 (Bi-accessibilité).
Dans un graphe orienté G, on définit la relation $\mathscr{R}'$ de bi-accessibilité par $u\mathscr{R}'v$ lorsqu'il existe un chemin orienté de u à v et un de v à u.
C'est une relation d'équivalence dont les classes sont appelées les **composantes fortement connexes** (abrégé CFC).
Si u est un sommet, on note C_u sa composante fortement connexe.

Démonstration. Prouvons brièvement que c'est bien une relation d'équivalence :

- Le chemin orienté vide mène de u à u.
- La symétrie est par définition.
- La transitivité se fait comme dans le cas non-orienté.

□

Exemple.

FIGURE 11.23 – Un graphe orienté et ses composantes fortement connexes

Proposition 46.
Les composantes fortement connexes d'un graphe orienté sans circuit sont des singletons.

Démonstration. Par contraposée : si $u \neq v$ sont dans la même CFC, il existe un chemin orienté (non-vide) de u à v et un de v à u. En les concaténant, on obtient un chemin orienté non-vide de u à u dont on peut extraire un circuit. □

Définition 47 (Graphe des CFC).
Soit $G = (S, A)$ un graphe orienté. On note $\mathscr{C}$ l'ensemble de ses CFC.
On pose $G^{CFC} = (\mathscr{C}, \mathscr{E})$ où $C_iC_j \in \mathscr{E}$ lorsqu'il existe dans G un arc d'un sommet de C_i vers un sommet de C_j. Ce graphe est appelé le **graphes des composantes fortement connexes** de G.

Exemple. Pour le graphje orienté de l'exemple précédent :

FIGURE 11.24 – Graphe des CFC du graphe de la figure 11.23

Théorème 48 (Acyclicité du graphe des CFC).
Pour tout G graphe orienté, G^{CFC} est sans circuit.

Démonstration. Procédons par l'absurde. Soit $G = (S, A)$ un graphe orienté et G^{CFC} son graphe des CFC. On suppose qu'il contient un circuit, que l'on nomme $C_0, C_1, ..., C_{r-1}, \underbrace{C_r}_{C_0}$. Autrement dit, dans G, il existe un arc v_0u_1 d'un sommet de C_0 à un sommet de C_1, v_1u_2 de C_1 à C_2, etc, jusqu'à $v_{r-1}u_0$:

FIGURE 11.25 – Impossibilité d'un cycle dans G^{CFC}

Mais comme chacun des C_i est fortement connexe, pour chaque i il existe dans G un chemin orienté de u_i à v_i. En concaténant tous ces chemins, on obtient un (long) chemin dans G :

$$u_0 \to ... \to v_0 \to u_1 \to ... \to v_1 \to u_2 \to ... ... \to v_{r-1} \to u_0$$

En particulier, ce cycle donne un chemin de u_1 à v_0. Or, $v_0u_1 \in A$: donc les deux sommets devraient être dans la même CFC, absurde. □

Remarque. Une façon de comprendre ce théorème est que l'on peut ordonner les composantes fortement connexes. Ainsi :

- Deux sommets d'un graphe orienté « sont au même niveau » s'ils sont dans la même CFC.
- Les sommets dans des composantes distinctes sont comparables s'il existe un chemin de la composante de l'un vers la composante de l'autre dans le graphe des CFC

Exemple. Voici un graphe de flot de controle :

FIGURE 11.26 – Un graphe de flot de controle

L'analyse de ce graphe des CFC consiste à savoir quelle(s) opération(s) peuvent avoir lieu avant quelles autres (et est donc utile pour compiler). Trouver les CFC permet également de trouver quelles sont les boucles, c'est à dire les emplacements du code succeptibles de ne pas terminer !

Exercice. Retrouver et spécifier le code d'origine.

2 Implémentation

Dans toute cette partie, on suppose que l'ensemble des sommets est $[\![0;n[\![$: autrement dit, les sommets sont des indices.

Remarque. Quand on voudra échapper à cette hypothèse, on utilisera un dictionnaire (que l'on verra bientôt) !

2.0 Matrice d'adjacence

Définition 49.
La **matrice d'adjacence** d'un graphe (orienté ou non) $G = ([\![0;n[\![, A)$ est la matrice $M = (m_{i,j})_{0\leq i,j<n}$ définie par :

$$m_{i,j} = \begin{cases} 1 & \text{si } ij \in A \\ 0 & \text{sinon} \end{cases}$$

Exemple.

(a) Un graphe (orienté) (b) Matrice d'adjacence associée

Figure 11.27 – Matrice d'adjacence d'un graphe

Remarque.

- La matrice d'adjacence d'un graphe non-orienté est symétrique.
- L'absence de boucles (arêtes *uu*) dans nos graphes impose que la diagonale est nulle.
- Le degré sortant d_+ d'un sommet est la somme de sa ligne.
- Le degré entrant d_- d'un sommet est la somme de sa colonne.
- On pourrait [8] [9] utiliser des booléens à la place des entiers.

8. On devrait.
9. Mais les 0/1 sont assez standards... et plus rapide à écrire.

- Si l'on veut étiquetter les arcs, on peut mettre `None` dans la matrice pour un arc absent et `Some etq` pour un arc présent.

Proposition 50 (Complexités avec une matrice d'adjacence).
Avec une matrice d'adjacence, on a les complexités suivantes :

- Tester si une arête est présente : $O(1)$
- Trouver tous les voisins d'un sommet : $\Theta(|S|)$
- Ajouter un noeud : $O(|S|^2)$
- Ajouter une arête : $\Theta(1)$
- Enlever un noeud : $O(|S|^2)$
- Enlever une arête : $\Theta(1)$

Démonstration.

- Pour savoir si $ij \in A$, il suffit de lire `M[i][j]`
- Pour trouver tous les voisins de i, il faut parcourir toute la ligne `M[i]` pour y trouver tous les 1 (et ce même si le degré du sommet est très faible !).
- Pour ajouter un nouveau noeud, il faut recréer toute la matrice avec le nouveau noeud en plus.
- Il suffit de modifier la bonne case de la matrice.
- Les deux derniers points sont comme l'ajout de sommet/arête.

□

Remarque.

- Cette représentation est particulièrement adaptée lorsque le graphe est dense, c'est à dire lorsqu'il possède « beaucoup » d'arêtes.
- Si on utilise des tableaux dynamiques, ajouter/enlever le sommet d'indice n peut se faire en $\Theta(|S|)$ amorti : il suffit d'ajouter une nouvelle case à la fin de chaque ligne, et une nouvelle ligne en bas de la matrice.

Proposition 51.
Une matrice d'adjacence coute $\Theta(|S|^2)$ en mémoire.

Démonstration. Djur à faire sans implémentation précise... surtout qu'avec une implémentation volontairement catastrophique des tableaux vous pouvez me donner tort... mais dans l'idée, on stocke chaque coefficient une seule fois et c'est tout. □

2.1 Listes d'adjacence

Définition 52 (Listes d'adjacence).
Soit $G = (\llbracket 0; n \llbracket, A)$ un graphe et $s \in S$. On appelle **liste d'adjacence** de s la liste de ses voisins. La **représentation par listes d'adjacence** de G consiste à calculer et stocker toutes les listes d'adjacence. On utilise généralement pour cela un tableau (ou un tableau associatif) qui à un sommet associe sa liste d'adjacence.

Exemple.

FIGURE 11.28 – Liste d'adjacence du graphe de la figure 11.27

Remarque.

- Sauf mention contraire, on ne suppose pas les listes d'adjacences ordonnées : les voisins peuvent être dans n'importe quel ordre.
- Je n'ai volontairement pas précisé comment sont implémentées les listes : on veut une façon de faire qui permet d'ajouter ou d'enlever le premier élément efficacemment ; peu importe l'implémentation précise (liste chainée, tableau borné, tableau dynamique).

Proposition 53 (Complexités avec des listes d'adjacence).
Avec des listes d'adjacence, on a les complexités suivantes :

- Tester si une arête uv est présente : $O(deg_+(u))$
- Trouver tous les voisins d'un noeud u : $\Theta(1)$
- Ajouter un noeud : $O(|S|)$
- Ajouter une arête uv : $\Theta(1)$
- Enlever un noeud : $O(|S| + |A|)$
- Enlever une arête uv : $O(deg_+(u))$

Démonstration.

- Il faut parcourir toute la liste d'adjacence de u pour y chercher v. Cette liste est de longueur $deg_+(u)$.
- Les voisins sont exactement la liste d'adjacence, qui est déjà calculée !
- Pour ajouter un nouveau noeud, il faut ajouter une case au tableau des listes. [10]
- Pour ajouter une arête, il suffit de l'ajouter à la liste d'adjacence.
- Les deux derniers points sont comme l'ajout de sommet/arête.
- Pour enlever un noeud, il faut enlever sa case du tableau des listes en $O(|S|)$; mais aussi parcourir toutes les autres listes pour y enlever le noeud s'il y est présent. Cela revient à parcourir toutes les arêtes en $O(|A|)$.
- Pour enlever une arête, il faut parcourir la liste d'adjacence

□

Remarque.

- Cette représentation est adaptée quand le graphe est creux, c'est à dire quand la matrice d'adjacence a « beaucoup » de zéros.
- C'est la représentation la plus utilisée en MP2I/MPI, car c'est la plus adaptée au parcours du graphe.
- Des implémentations spécifiques pourraient obtenir des complexités différentes. Par exemple, en imposant que les listes d'adjacence sont des tableaux (dynamiques) triées, la recherche d'une arête se fait en temps $\log(deg_+)$... mais l'insertion devient linéaire en deg_+.

10. Là encore, avec des tableaux dynamiques cela se fait en temps constant.

Proposition 54.
Les listes d'adjacence coutent $\Theta(|S| + |A|)$ en mémoire.

Démonstration. Là encore, dur à faire sans implémentation précise... Mais dans l'idée, on stocke $|S|$ listes. La liste d'un sommet s contient $deg_+(s)$ valeurs, donc elles contiennent au total $\sum_{s \in S} deg_+(s) = |A|$. □

Remarque. En comparant avec le coût d'une matrice d'adjacence, on comprend pourquoi cette représentation est adaptée aux graphes creux.

2.2 Comparatif

Opération \ Structure	Matrice d'adjacence	Listes d'adjacence
Tester si $uv \in A$	1	$deg_+(u)$
Parcourcir les voisins de u	$\|S\|$	$deg_+(u)$
Ajouter un sommet	$\|S\|^2$	$\|S\|$
Ajouter une arête	1	1
Enlever un sommet	$\|S\|^2$	$\|S\| + \|A\|$
Enlever une arête uv	1	$deg_+(u)$

(a) Complexités temporelles des opérations usuelles

	Matrice d'adjacence	Listes d'adjacence
Complexité spatiale	$\|S\|^2$	$\|S\| + \|A\|$

(b) Complexités spatiale

FIGURE 11.29 – Ordre de grandeur des complexités pour des matrices d'adjacence et des listes d'adjacence

3 Parcours de graphes

3.0 Parcours générique

3.0.0 Parcours depuis un sommet

Le parcours de graphe sert à se déplacer dans un graphe en appliquant un certain traitement lorsque l'on visite un sommet. On ne peut cependant pas simplement appliquer tel quel l'algorithme de parcours des arbres :

FIGURE 11.30 – Boucle infinie d'un DFS à cause d'un cycle

Il faut « réparer » le parcours en imposant que l'on ne peut pas repasser deux fois par le même sommet. Afin de garder le code du parcours ci-dessous, on se donne un « sac » qui permet de stocker les arêtes empruntées (elle joue le rôle de la pile/file que l'on a utilisée dans le parcours d'arbre).

Algorithme 16 : Parcours de graphe

Entrées : G un graphe et s_0 un sommet du graphe

```
1 sac ←sac vide
2 Insérer s_0 dans sac
3 tant que sac est non-vide faire
4 |   u ←extraire un sommet du sac
5 |   si u n'est pas déjà marqué alors
6 |   |   Marquer u
7 |   |   Visiter u (çad lui appliquer le traitement que l'on veut)
8 |   |   pour chaque v voisin de u faire
9 |   |   |   Insérer v dans sac
```

Remarque.

- En pratique, le marquage (qui permet donc de savoir si un sommet a déjà été visité ou non) se fait avec un tableau qui à un sommet associe s'il est marqué (`true`) ou non (`false`).
- Le sac peut contenir des doublons. Cela sera même très important pour les parcours en profondeur.
- On peut optimiser en ne réinsérant pas dans le sac des sommets déjà marqués. Je ne le fais pas encore dans ce pseudo-code afin de simplifier l'analyse de complexité, mais une fois celle-ci terminée je le ferai systématiquement.

Exemple.

(a) Un graphe

(b) Un parcours du graphe, sans réinsérer des sommets déjà marqués

Proposition 55.
Un parcours de G depuis s_0 visite exactement les sommets accessibles depuis s_0.

Démonstration.

- Tout sommet visité est accessible depuis s_0. En effet, tout sommet visité est d'abord extrait du sac : on montre l'invariant que tous les sommets du sac sont accessibles depuis s_0 :
  - Avant la première itération de la boucle `while`, le sac contient uniquement s_0.
  - Si l'invariant est vrai au début d'une itération : on extrait u du sac, qui est donc accessible depuis s_0. Ensuite si l'on rentre dans le Si, on ajoute dans le sac les voisins de u : ils sont donc accessibles depuis u donc par transitivité depuis s_0. D'où la conservation.
- Tout sommet u accessible est visité. Pour montrer cela, remarquons que tout sommet ajouté dans le sac fini par être visité et montrons par récurrence sur $d(s_0, u)$ que u est inséré dans le sac :
  - Si $d(s_0, u) = 0$, alors $u = s_0$ et ce sommet est bien inséré dans le sac (avant la boucle).
  - Si $d(s_0, u) > 0$ et que la propriété est vraie pour tout sommet plus proche : considérons un plus court chemin de s_0 à u et nommons p_u le sommet juste avant u dans ce chemin. Comme un sous-chemin d'un plus court chemin est un plus court chemin, $d(s_0, p_u) = d(s_0, u) - 1$. Donc par HR, p_u a été ajouté dans le sac. Mais d'après la condition de fin du `while`, p_u finit par sortir du sac. La première fois qu'il en sort, on entre dans le Si et ajoute au sac tous ses voisins, dont u.

□

3.0.1 Parcours complet

Il arrive souvent que l'on veuille visiter *tous* les sommets. On appelle cela faire un **parcours complet** du graphe. C'est en fait très simple :

Algorithme 16 : Parcours complet

Entrées : G un graphe

```
1 pour chaque sommet u de G faire
2 |  si u n'a pas déjà été marqué lors d'un parcours alors
3 |  |  Lancer un parcours de G depuis u
```

Remarque. En pratique, cela demande de faire en sorte que le tableau qui mémorise si un sommet est marqué ou non soit partagé entre les différents parcours. Cela se met typiquement en oeuvre en en faisant une entrée de la fonction de parcours.

Proposition 56.
Lors d'un parcours complet du graphe, chaque sommet est visité exactement une fois.

Démonstration. Chaque sommet est visité au plus une fois grâce au marquage partagé, et le parcours complet lance un parcours depuis chaque sommet non-encore marqué (donc chaque sommet est visité au moins une fois). □

3.0.2 Complexité

Théorème 57 (Complexité d'un parcours).
En notant c_{visite} la complexité de la visite d'un sommet, c_{insere} le coût de l'insertion dans le sac et $c_{extrait}$ le coût de l'extraction du sac, la complexité d'un parcours d'un graphe G est :

- $\Theta(|S|c_{visite} + |A|(c_{insere} + c_{extrait}))$ pour un graphe implémenté par liste d'adjacence.
- $\Theta(|S|c_{visite} + |S|(|S| + c_{insere} + c_{extrait}))$ pour un graphe implémenté par matrice d'adjacence.

Démonstration. Comptons les opérations qui ont lieu. Tout d'abord, la boucle `Pour` du parcours de graphe complet, sans compter les parcours qu'elle lance, est en $\Theta(|S|)$. Étudions les opérations qui ont lieu lors des parcours :

- Il y a les opérations hors de la boucle. Elles sont en $O(1)$, comme on lance un parcours depuis au plus chaque sommet cela fait $O(|S|)$ au total.
- Il y a chaque début d'itération : tester si le sac est vide en $O(1)$, extraite un sommet du sac en $\Theta(c_{extrait})$ et tester si le sommet est marqué en $O(1)$: le tout est donc en $\Theta(c_{extrait})$. Mais comme une extraction n'a lieu qu'après une insertion, on va compter ces coûts-ci lorsque l'on compte les insertions.
- Et le corps du Si de la boucle TantQue. Chaque exécution de ce corps effectue :
  - un marque en $O(1)$
  - une visite en $\Theta(c_{visite})$
  - Puis le parcours de tous les voisins de u pour les insérer. Sur une liste d'adjacence, cela se fait en $\Theta(deg_+(u)c_{insere})$, sur une matrice en $\Theta(|S| + deg_+(u)c_{insere})$; la différence venant du fait que dans une matrice d'adjacence on doit parcourir tous les sommets pour savoir s'ils sont voisins ou non.

Terminons les calculs dans le cas des listes d'adjacence. Comme chaque sommet est visité exactement une fois, la complexité des boucles `while` des parcours successifs est un Θ de :

$$\begin{aligned}\sum_{u\in S} c_{visite} + deg_+(u)(c_{extrait} + c_{insere}) &= |S|c_{visite} + (c_{extrait} + c_{insere})\sum_{u\in S} deg_+(u) \\ &= |S|c_{visite} + (c_{extrait} + c_{insere})|A|\end{aligned}$$

La complexité totale étant la complexité de toutes les boucles while ci-dessous, plus le $O(|S|)$ total des lignes 1-2 du parcours (avant la boucle) plus le $O(|S|)$ total de la boucle for du parcours complet, on obtient bien le résultat attendu. □

Remarque. Dans cette preuve, j'ai supposée que créer un sac vide et tester si un sac était vide sont en $O(1)$... c'est très raisonnable.

Corollaire 58.
Quand le coût de la visite, de l'insertion et de l'extraction sont constant, on obtient les complexités suivantes pour un parcours de G depuis s_0 :

- $O(|S| + |A|)$ pour des listes d'adjacence
- $O(|S|^2)$ pour une matrice d'adjacence.

Démonstration. C'est un corollaire du théorème précédent, avec un $O()$ et non un Θ car on ne fait ici pas un parcours complet. □

Cela dit, la preuve du théorème était un peu longue. On la présente souvent de manière plus courte comme suit :

Démonstration.

- Avec des listes d'adjacence : Chaque sommet est visité au plus une fois. Lors de la visite d'un sommet, on le visite puis on parcourt sa liste d'adjacence pour insérer ses voisins. Chaque extraction correspond à une insertion. Donc la complexité totale est :

$$O\Big(\sum_{s\in S} \underbrace{1}_{\text{visite}} + \underbrace{deg_+(s)}_{\text{extraction et insertion des voisins}}\Big) = O(|S| + |A|)$$

- Avec une matrice d'adjacence : Chaque sommet est visité au plus une fois. Lors de la visite d'un sommet, on le visite puis on parcourt sa ligne de la matrice pour insérer ses voisins. Chaque extraction correspond à une insertion. Donc la complexité totale est :

$$O\Big(\sum_{s\in S} \underbrace{1}_{\text{visite}} + \underbrace{|S|}_{\text{parcours ligne}} + \underbrace{deg_+(s)}_{\text{extraction et insertion des voisins}}\Big) = O(|S|^2)$$

□

Remarque.

- J'accepte tout à fait la preuve ci-dessus en DS (elle provient presque d'un rapport de jury après tout !). On peut très simplement la modifier pour gérer c_{visite}, c_{insere} et $c_{extrait}$.
- Les parcours de graphe, et l'analyse de leur complexité, sont le B-A-BA de l'algorithmique des graphes qui constitue un *gros* morceau de MP2I/MPI ! Il est donc *très* important de savoir écrire un parcours et de savoir réaliser la preuve abrégée ci dessus !!

3.0.3 Arbre de parcours

Définition 59.
L'**arbre de parcours** associé à un parcours d'un graphe G depuis s_0 est l'arbre constitué des arcs $p_u u$ où p_u est le sommet qui a inséré l'exemplaire de u dont l'extraction a causé la visite de u. Autrement dit, on insère des *arcs* $p_u u$ dans le sac. Quand on extrait un arc $p_u u$ du sac, si u n'a pas encore été marqué on note que p_u est son parent.

Remarque. Cette définition est rendue subtile car un sommet peut apparaitre en doublon dans le sac ; et qu'il y aurait alors ambiguïté sur qui est le parent.

Exemple.

FIGURE 11.32 – Ambiguïté possible sur la parenté si l'on ne fait pas attention

Proposition 60.
Pour représenter un arbre de parcours, le plus agréable est généralement un tableau de parenté : pour chaque sommet, on mémorise son parent dans le parcours.

Remarque. Cela fonctionne car on peut identifier uniquement un sommet (par exemple, par son étiquette.)

Exemple.

FIGURE 11.33 – Arbre de parcours pour l'exemple de parcours générique précédent et tableau de parenté associé

Cela donne le pseudo-code suivant :

Algorithme 17 : Calcul de l'arbre du parcours

Entrées : G un graphe et s_0 un sommet du graphe
Sorties : *parent* un tableau tel que $parent[u]$ est le parent de u lors du parcours (vide si u n'est pas visité ou est s_0)

```
1  sac ←sac vide
2  parent ←tableau dont toutes les cases sont vides
3  Insérer (s_0, s_0) dans sac
4  tant que sac est non-vide faire
5      (p_u, u) ←extraire un sommet du sac
6      si u n'est pas déjà marqué alors
7          Marquer u
8          parent[u] ←p_u
9          pour chaque v voisin de u non-encore marqué faire
10             Insérer v dans sac
11 parent[s_0] ←vide
12 renvoyer parent
```

Remarque. Notez que :

- On insère désormais des *arcs* dans le sac et non plus des sommets !!
- Il y a une légère difficulté : quel est l'arc qui insère s_0 le sommet initial ? Aucun... Donc on fait comme si s_0 était son propre parent ; et on corrige cela à la fin (ligne 11) en marquant que s_0 n'a en fait pas de parent.
- Les éléments qui n'ont pas de parent dans l'arbre sont s_0 et les sommets non-accessibles depuis s_0.
- En OCaml, un type `option` est très adapté pour stocker les parentés.

Proposition 61.
Un chemin depuis s_0 dans l'arbre de parcours de G depuis s_0 est un chemin depuis s_0 dans G.
Autrement dit, l'arbre de parcours est un sous-graphe de G.

Démonstration. C'est presque exactement la preuve du fait que le parcours depuis s_0 visite les sommets accessibles depuis s_0. □

Proposition 62.
Soit G un graphe et s_0 un sommet de G. On suppose que tous les sommets sont accessibles depuis s_0.
Si l'on dé-enracine (et désoriente) l'arbre du parcours de G depuis s_0, il est bien connexe acyclique : c'est bien un arbre.

Démonstration. Il est connexe car tout le monde est relié à s_0 le sommet de départ du parcours (encore une fois presque exactement la preuve du fait que le parcours visite les sommets accessibles).

De plus, chaque sommet a exactement un prédecesseur sauf s_0 qui n'en a pas : avec n le nombre de sommets de l'arbre, il y a donc $n-1$ liens de parentés ($n-1$ arêtes) et il s'agit donc d'un arbre d'après le théorème de caractérisation des arbres. □

Remarque. Les sommets non-accessibles depuis s_0 ne sont pas dans l'arbre, donc il est toujours acyclique. Ils sont juste embêtants pour la connexité... Mais dans l'idée, on a même pas envie de dire qu'ils sont dans l'arbre de parcours : c'est donc assez naturel de les exclure comme le fait cette propriété.

Définition 63.
Les parentés calculées par un parcours complet forme une **forêt de parcours**, c'est à dire qu'il s'agit de plusieurs arbres disjoints.

Démonstration. Dans un parcours complet, chaque parcours complet visite exactement les sommets non encore marqués accessibles depuis le point de départ. Donc les arbres de parcours sont disjoints ; aussi leur union crée une forêt. □

Exemple.

Figure 11.34 – Une forêt de parcours possible pour un graphe

3.1 Parcours en largeur

3.1.0 Définition et propriété

Définition 64.
Le **parcours en largeur** d'un graphe consiste à parcours les sommets par distance croissante au sommet initial.
Il est obtenu **en utilisant une file dans le parcours**.

Exemple.

FIGURE 11.35 – Parcours en largeur d'un graphe (sans réinséré les sommets déjà marqués)

Remarque. Il n'y a pas de règle sur l'ordre d'enfilage des voisins d'un sommet. Cela n'a pas d'importance [11].

Théorème 65 (Arbre d'un parcours en largeur).
Dans un graphe G (non-pondéré), l'arbre du parcours en largeur depuis s_0 est un arbre de plus courts chemins depuis s_0.

Démonstration. PREUVE FAUSSE À REPRENDRE. Admis en attendant.

□

3.1.1 Une optimisation : le marquage anticipé

Comme une file est FIFO, l'extraction d'un sommet u de la file correspond à sa *première* insertion. Aussi, au lieu de marquer un sommet lorsqu'il sort de la file, on peut le marquer lors de son *entrée* dans

11. Si vous avez besoin d'un ordre rpécis, vous ne faites pas un parcours en largeur ; vous faites un parcours en largeur particulier.

la file sans changer l'ordre de parcours. On parle de **marquage anticipé**. Cela permet de réduire la taille maximale de la file en y éliminant les doublons.

3.1.2 Implémentation en OCaml

On se donne les types suivants pour manipuler un graphe par liste d'adjacence, et un type d'arbres représentés par tableau de parenté :

parcours.ml

```
type sommet = int
type graphe = sommet list array  (* listes d'adjacence *)

type arbre = sommet option array (* tableau de parenté *)
```

On propose ci-dessous une implémentation en OCaml du parcours en largeur, qui utilise le module `Queue` pour manipuler des files. Il s'agit de files impératives : les opérations que l'on fait dessus *modifient* la file.

parcours.ml

```
let bfs (g : graphe) (s0 : sommet) : arbre =
  let n = Array.length g in
  let parent = Array.make n None in
  let marque = Array.make n false in
  let file = Queue.create () in
  (* La file contient des arcs [(pu,u)] avec
     [pu] le sommet qui enfile [u] dans la file *)
  Queue.add (s0, s0) file;
  marque.(s0) <- true; (* marquage anticipé !*)

  while not (Queue.is_empty file) do
    let (pu, u) = Queue.take file in
    parent.(u) <- Some pu;

    let insere v =
      if not marque.(v) then begin
        Queue.add (u,v) file;
        marque.(v) <- true (* marquage anticipé ! *)
      end
    in
    List.iter insere g.(u)
  done;

  parent.(s0) <- None;
  parent
```

3.2 Parcours en profondeur

3.2.0 Définition et propriétés

Définition 66.
Le **parcours** en profondeur d'un graphe consiste à parcourir les sommets en « avançant » le plus possible dans le graphe, et en revenant sur ses pas (jusqu'à pouvoir avancer à nouveau) lorsque tous les embranchements possibles ont déjà été visités.
Il s'écrit très naturellement **récursivement**, **ou bien en parcours avec une pile**.
Lors d'un parcours en profondeur récursif, on appelle **ouverture** d'un sommet le moment où ce sommet est marqué/visité, et **fermeture** le moment où les appels récursifs sur ses voisins termine. On numérote souvent ces sommets à partir de 0 : la première ouverture a lieu au temps 0, l'ouverture/fermeture qui suit au temps 1, celle qui suit encore au temps 2, etc.

Voici l'écriture récursive du parcours en profondeur (remarquez qu'il faut que le marquage soit partagé entre les appels) :

Algorithme 18 : DFS

Entrées : G un graphe, u un sommet

```
1 si u n'est pas déjà marqué alors
      // Ouverture
2     Marquer u
3     Visiter u
4     pour chaque voisin v de u non-encore marqué faire
5         DFS(G, v)
      // Fermeture
```

Exemple.

Figure 11.36 – Parcours en profondeur récursif d'un graphe

Remarque.

- Le parcours récursif empile des appels sur la... pile d'appels. Donc il utilise aussi une pile !
- Les notions d'ouverture et de fermeture sont compliquées à définir en itératif : c'est une raison de préférer le parcours récursif.
- L'ouverture d'un sommet a lieu avant sa fermeture.

L'arbre d'appels d'un parcours en profondeur a *d'excellentes* propriétés, qui font de ce parcours un *must-have*. Avant de les étudier, formalisons la construction de l'arbre :

Algorithme 19 : Construction de l'arbre de parcours en profondeur

Entrées : G un graphe, p_u un sommet et u un sommet
Pré-conditions : p_u est le sommet qui a lancé l'appel actuel

```
1 si u n'est pas déjà marqué alors
      // Ouverture
2     Marquer u
3     Noter que p_u est le parent de u dans l'arbre
4     pour chaque voisin v de u non-encore marqué faire
5         DFS(G, u, v)
      // Fermeture
```

Exemple.

FIGURE 11.37 – Un arbre de parcours en profondeur avec les dates d'ouverture/fermeture

Lemme 67.
On considère un parcours en profondeur complet d'un graphe G. On note $d(\cdot)$ et $f(\cdot)$ les temps d'ouverture et fermeture (qui ont une numérotation commune à partir de 0).
Alors v est descendant de u dans la forêt de parcours si et seulement si $d(u) < d(v) < f(v) < f(u)$.

Démonstration. C'est dur à formaliser ; et quand on le fait c'est embêtant pour pas grand chose [12]. Je vais simplement donner l'idée :

- Sens direct : v est descendant de u signifie exactement que u a lancé un appel qui a lancé un appel qui... qui a lancé l'appel qui a visité v. Ainsi, $d(u) < d(v)$. Mais de plus, avant que u ne se ferme ses appels récursifs (et les appels de ses appels, etc) doivent se fermer : donc $f(v) < f(u)$. D'où le sens direct de l'implication.
- Sens réciproque : Si $d(u) < d(v) < f(v) < f(u)$, cet encadrement signifie exactement que l'appel de la visite de v a été au-dessus de l'appel de la visite de u dans la pile d'appel (avec éventuellement d'autres appels entre les deux). Or, d'après le fonctionnement de la pile d'appel tous les appels au-dessus de l'appel de u sont des appels récurifs de u, ou des appels de leurs appels, etc. En particulier, l'ouverture de v est un appel d'un appel d'un appel ... de l'appel de u : v descend donc de u dans l'arbre d'appels.

□

12. Et je *déteste* voir ce lemme demandé en concours ; car je ne sais pas ce qui est attendu.

Lemme 68.
On considère un parcours en profondeur complet d'un graphe G. On note $d(\cdot)$ et $f(\cdot)$ les temps d'ouverture et fermeture (qui ont une numérotation commune à partir de 0).
Alors il ne peut pas existe de u et v tels que $d(u) < d(v) < f(u) < f(v)$.

Démonstration. À peu près comme la preuve précédente : $d(u) < d(v) < f(u)$ signifie que l'appel sur u a fait un appel qui ... qui a fait l'appel sur v. En particulier, l'appel sur v doit terminer avant de rendre la main à l'appel sur u, donc $f(v) < f(u)$. □

Théorème 69 (Théorème des parenthèses).
On considère un parcours en profondeur complet d'un graphe G. On note $d(\cdot)$ et $f(\cdot)$ les temps d'ouverture et fermeture (qui ont une numérotation commune à partir de 0).
Pour deux sommets u et v, une seule des trois conditions suivantes est vérifiée :

- Les intervalles $[d(u); f(u)]$ et $[d(v); f(v)]$ sont disjoints, et ni u ni v ne sont descendant l'un de l'autre dans la forêt du parcours complet.
- l'intervalle $[d(u); f(u)]$ est entièrement inclus dans l'intervalle $[d(v); f(v)]$ et u est un descendant de v dans la forêt de parcours.
- l'intervalle $[d(v); f(v)]$ est entièrement inclus dans l'intervalle $[d(u); f(u)]$ et v est un descendant de u dans la forêt de parcours.

Démonstration. D'après le second lemme précédent, les intervalles sont soit inclus soit disjoints. Le premier lemme conclut. □

Remarque.

- Ce que ce théorème dit, c'est que les ouvertures/fermetures sont « bien parenthésées », et que la forêt de parcours indique les imbrications de ces parenthèses.
- En MPI, vous approfondirez ce théorème en classifiant les arcs du graphe G de en 4 catégories par rapport à la forêt de parcours G_Π [13]. Ce sera le théorème de classification des arcs.

3.2.1 Pas de marquage anticipé !!

Considérons le parcours en profondeur itératif avec une pile. Si l'on effectue un marquage anticipé (c'est à dire que l'on interdit les doublons dans la pile), on peut obtenir l'arbre de parcours suivant :

Figure 11.38 – Le marquage anticipé ne donne pas un parcours en profondeur

Cet arbre n'est pas un arbre de parcours en profondeur... En fait :

Proposition 70.
Quand on écrit un parcours en profondeur itératif :

- avec marquage anticipé, on obtient pas un DFS.
- **sans marquage anticipé**, on obtient bien un DFS.

13. Les arcs de liaison (les arcs G qui sont dans $G\Pi$), les arcs avant (les arcs de G « raccourcissent » un chemin de G_Π), les arcs arrière (les arcs de G qui « remontent » un chemin de G_Π), et les arcs transverses (les autres, çad ceux qui relient deux sommets qui ne sont pas ancêtres l'un de l'autre dans G_Π).

Démonstration. Le premier point est Figure 11.38, le second est admis. □

Remarque.

- Le parcours en profondeur s'écrit très très bien en récursif, où cette subtilité n'apparait pas.
- On appelle parfois « parcours pile » le parcours itératif avec une pile et marquage anticipé.

3.2.2 Implémentation en OCaml

Voici le code de l'implémentation récursive, pour construire l'arbre de parcours.

parcours.ml

```
67  let dfs (g : graphe) (s0 : sommet) : arbre =
68    let n = Array.length g in
69    let marque = Array.make n false in
70    let parent = Array.make n None in
71
72    let rec dfs_rec pu u =
73      if not marque.(u) then begin
74        marque.(u) <- true;
75        parent.(u) <- Some pu;
76        List.iter (fun v -> if not marque.(v) then dfs_rec u v) g.(u)
77      end
78    in
79
80    dfs_rec s0 s0;
81    parent.(s0) <- None;
82    parent
```

Remarque. Dans le code ci-dessus, les tests de non-marquage ligne 72 et 75 sont redondants... mais c'est spécifique à l'écriture récursive. Il vaut mieux mettre trop de tests de non-marquage que pas assez.

Et voici une autre version, qui cette fois calcule les dates d'ouverture et de fermeture lors d'un parcours complet. On y utilise la fonction `incr` qui augmente de 1 le contenu d'une référence.

parcours.ml

```
89  let dates (g : graphe) : int array * int array =
90    let n = Array.length g in
91    let ouverture = Array.make n (-1) in
92    let fermeture = Array.make n (-1) in
93    let time = ref 0 in
94    let marque = Array.make n false in
95
96    let rec dfs_rec u =
97      if not marque.(u) then begin
98        marque.(u) <- true;
99        ouverture.(u) <- !time; incr time;
100
101       List.iter (fun v -> if not marque.(v) then dfs_rec v) g.(u);
102
103       fermeture.(u) <- !time; incr time
104     end
105   in
106
107   for s0 = 0 to n-1 do
108     if not marque.(s0) then dfs_rec s0
109   done;
110   ouverture, fermeture
```

3.3 Application : calcul des composantes connexes

On peut utiliser des parcours pour calculer les composantes connexes d'un graphe non-orienté. En effet, on a dit qu'un parcours visite exactement les sommets accessibles depuis son point de départ. Donc dans un graphe non-orienté, un parcours visite exactement une composante connexe.

On va donc faire un parcours complet : il numérote tous les sommets du programme parcours par 0 (et ils forment exactement une composante connexe), tous les sommets du second parcours par 1, etc. Ainsi, on construit les composantes connexes $C_0, C_1, \ldots$ en numérotant tous les sommets de la composante C_i par i.

Cela donne le code suivant en OCaml, toujours avec les mêmes types que précédemment :

parcours.ml

```
let composantes (g : graphe) : int array =
  let n = Array.length g in
  let comp = Array.make n (-1) in
  let num_comp = ref 0 in
  let marque = Array.make n false in

  let bfs s0 =
    let file = Queue.create () in
    Queue.add s0 file;
    marque.(s0) <- true; (* marquage anticipé ! *)
    while not (Queue.is_empty file) do
    let u = Queue.take file in
      marque.(u) <- true;
      comp.(u) <- !num_comp;
      let insere v =
        if not marque.(v) then begin
          Queue.add v file;
          marque.(v) <- true (* marquage anticipé ! *)
        end
      in
      List.iter insere g.(u)
    done
  in

  for s0 = 0 to n-1 do
    if not marque.(s0) then begin
      bfs s0;
      incr num_comp
    end
  done;
  comp
```

En particulier :

Proposition 71.
Un parcours complet d'un graphe non-orienté permet de calculer les composantes connexes en temps linéaire en la taille du graphe.

Remarque. Pour un graphe orienté, on veut calculer les composantes fortement connexes. C'est un problème plus compliqué, que vous résoudrez en MPI à l'aide de l'algorithme de Kosaraju [14].

14. Il s'agit d'exploiter à leur plein potentiel le théorème des parenthèses et le théorème de classification des arcs ; c'est un très bel algorithme.

3.4 Application : calcul d'un ordre topologique

3.4.0 Définition

Définition 72 (Ordre topologique).
Soit $G = (S, A)$ un graphe orienté. Un **ordre topologique** sur S est un ordre $\preccurlyeq$ tel que $uv \in A$ implique $u \prec v$.

Exemple.

FIGURE 11.39 – Un graphe orienté et des sommets numérotés selon un ordre topologique

Remarque.

- Un ordre topologique sert à trouver un ordre dans lequel réaliser les tâches d'un graphe de dépendances.
- Il n'y a pas unicité d'un ordre topologique. Par exemple, le graphe ci-dessous admet deux ordres topologiques :

FIGURE 11.40 – Multiples ordres topologiques

Il s'avère même que trouver le nombre d'ordres topologiques est un problème difficile [15].

Proposition 73.
Soit G un graphe orienté qui contient un circuit. Alors G n'a pas d'ordre topologique.

Démonstration. Soit $u_0, ..., \underbrace{u_p}_{=u_0}$ un tel circuit. Alors un ordre topologique donnerait $u_0 \prec ... \prec u_p = u_0$ donc $u_0 \prec u_0$. Contradiction. □

3.4.1 Construction d'un ordre topologique

Définition 74 (DAG).
Un **DAG** (**D**irect **A**cyclic **G**raph) est un graphe orienté sans cycle.

Théorème 75 (Calcul d'un ordre topologique).
Soit $G = (S, A)$ un DAG.
Ordonner les sommets de S par date de fin décroissante dans un DFS complet donne un ordre topologique.

Démonstration. Soit $G = (S, A)$ un DAG et $d()$ et $f()$ les dates d'ouverture et de fermeture lors d'un parcours en profondeur complet.

Soit $uv \in A$. Distinguons deux cas :

- Si le parcours visite u avant v : alors comme $uv \in A$, la visite de u lance un appel sur v. Ainsi, v n'était pas visité à l'ouverture de u et est garanti de l'être à la fermeture de u : donc $d(u) < d(v) < f(u)$. D'après le théorème des parenthèses, on a en fait $d(u) < d(v) < f(v) < f(u)$ et on a donc bien $u \prec v$.
- Sinon il visite v avant u : alors montrons que $d(v) < f(v) < d(u) < f(u)$.
  D'après le théorème des parenthèses, il n'y a que deux possibilités avec $d(v) < d(u)$:
  - Celle que l'on veut prouver.
  - Et l'imbrication des intervalles, c'est à dire $d(v) < d(u) < f(u) < f(v)$. Or, dans ce cas, d'après le théorème des parenthèses il y a un chemin dans l'arbre de parcours de v à u. Comme l'arbre de parcours est un sous-graphe, il y aurait un chemin dans G de v à u. Mais puisque $uv \in A$ cela donne un circuit : absurde.
    Donc la seule possibilité était la précédente, qui donne bien $u \prec v$.

□

En particulier, on a prouvé que :

Corollaire 76.
Un graphe orienté G admet un ordre tyopologique si et seulement si c'est un DAG.
Si c'est le cas, un ordre peut-être calculé en temps linéaire en la taille du graphe par un parcours en profondeur.

3.4.2 Détection de cycles

Lemme 77 (Théorème du chemin blanc).
Dans G un graphe orienté (pas forcément un DAG), un sommet v est descendant d'un sommet u dans la forêt de parcours si et seulement si lors de la visite de u (le moment $d(u)$) il existe un chemin de u à v composé uniquement de sommets qui non-encore ouverts.

Démonstration. $\Longrightarrow$) Immédiat d'après le théorème des parenthèses.

$\Longleftarrow$) Supposons que lors de la visite de u (donc au temps $d(u)$), il existe un tel chemin vers un sommet v mais que v ne devienne pas descendant de u dans l'arbre.

Sans perte de généralité, supposons que tous les autres sommets du chemin deviennent descendants de u (sinon, remplacer v par le premier sommet du chemin qui ne devient pas descendant). Notons w le sommet juste avant v dans le chemin :

FIGURE 11.41 – Preuve du chemin blanc

D'après le théorème des parenthèses, on a donc $d(u) < d(w) < f(w) < f(u)$.

Reste à se demander quand est-ce que v doit être ouvert/fermé par rapport à w. Pour que v ne soit pas descendant de w, il ne doit pas avoir été ouvert par w (théorème des parenthèses) ; mais puisqu'il est voisin de w il doit avoir été ouvert avant.

Donc :

$$d(u) < d(v) < d(w) < f(w) < f(u)$$

On conclut en appliquant encore le théorème des parenthèses qui donne $f(v) < f(u)$ et donc v descendant de u : absurde.

□

Proposition 78.
Soit $G = (S, A)$ un graphe orienté. Alors G admet un circuit si et seulement si lors d'un DFS complet de G il existe un sommet x qui a un voisin y ouvert non-encore fermé.

Démonstration.

$\Longrightarrow$) C'est à peu près comme la preuve du théorème de calcul d'ordre topologique : si lors de la visite de x, son voisin y est déjà ouvert et pas encore fermé, on a $d(y) < d(x) < f(y)$. D'après le théorème des parenthèses, cela implique $d(y) < d(x) < f(x) < f(y)$ et donc il y a un chemin de y à x. Mais y est voisin de x c'est à dire $vu \in A$: d'où un cycle.

$\Longleftarrow$) Par contraposée. Considérons $s_0, ..., \underbrace{s_p}_{=s_0}$ un circuit dans G. Quitte à renuméroter, supposons que s_0 soit le premier sommet visité par le parcours.

Par théorème du chemin blanc, s_{p-1} est descendant de s_0 et donc (théorème des parenthèses) : $d(s_0) < d(s_{p-1}) < f(s_{p-1}) < f(s_0)$. Autrement dit, lors du parcours de s_{p-1}, celui voit son voisin s_0 qui est ouvert non encore fermé.

□

On en déduit un critère pour détecter si un graphe orienté est un DAG. Ainsi, on peut faire un code qui :

- Renvoie une erreur si le graphe donné en entrée contient un circuit.
- Renvoie un ordre topologique sinon.

3.4.3 Implémentation en OCaml

On utilise toujours les mêmes types pour implémenter. Afin d'alléger le code, remarquons qu'ici le tableau des dates d'ouverture ne sert pas : pour tester si un sommet `v` a déjà été ouvert, il suffit de tester si `marque.(v)` est vrai. En particulier, on va ne compter *que* les dates de fermeture, de 0 à $n-1$.

De plus, l'étape finale consiste grosso-modo à « échanger » les indices et les valeurs du tableau fermeture. Plus précisément, le sommet fermé au temps $n-1$ est le premier sommet de l'ordre donc va à l'indice 0 du tableau des sommets trié, celui fermé au temps $n-2$ va à l'indice 1, etc : le sommet fermé au temps f va à l'indice $n-1-f$.

Figure 11.42 – Construction du tableau trié des sommegts à l'aide du tableau des temps de fermeture

parcours.ml

```
exception Cycle

let ordre_topo (g : graphe) : sommet array =
  let n = Array.length g in
  let fermeture = Array.make n (-1) in
  let time = ref 0 in
  let marque = Array.make n false in

  let rec dfs_rec u =
    if not marque.(u) then begin
      marque.(u) <- true;

      let check_voisin v =
        if marque.(v) && fermeture.(v) = -1
        then raise Cycle
        else if not marque.(v) then dfs_rec v
      in
      List.iter check_voisin g.(u);

      fermeture.(u) <- !time; incr time
    end
  in

  (* Calculer toutes les fermetures *)
  for u = 0 to n-1 do
    if not marque.(u) then dfs_rec u
  done;

  (* Construire l'ordre à partir des fermetures *)
  let ordre = Array.make n (-1) in
  for u = 0 to n-1 do ordre.(n-1 - fermeture.(u)) <- u done;
  ordre
```

4 Graphes bipartis

4.0 Définition

Définition 79 (Graphe biparti).
Un graphe $G = (S, A)$ (orienté ou non) est dit **biparti** s'il existe une bipartition $S = T \sqcup U$ tel que toute arête/arc de A ait une extrémité dans T et l'autre dans U.

Exemple.

(a) Un graphe biparti

(b) Un graphe biparti

FIGURE 11.43 – Des graphes bipartis

Exemple. On a défini au début du cours $K_{p,q}$ la clique bipartie.

Proposition 80.
Les arbres (graphe non-orienté acyclique connexe) sont biparits.

Démonstration. C'est... surprenament embêtant avec la définition « graphe » des arbres (alors que ce serait très simple dans des arbres orientés du cours sur les arbres). Vous le prouverez en MPI. □

Proposition 81.
Un graphe non-orienté G est biparti si et seulement si il est deux coloriable.

Démonstration. Par double implication.

$\Longrightarrow$) Soit $S = T \sqcup U$ une bipartition des sommets qui vérifie la définition de graphe biparti. Colorions avec la couleur 0 les sommets de T et 1 ceux de U. Alors toute arête a une extrémité 0 et une extrémité 1, donc le coloriage est valide.

$\Longleftarrow$) Soit $c : S \to \{0; 1\}$ une 2-coloration du graphe. Posons $T = c^{-1}(\{0\})$ et $U = c^{-1}(\{0\})$. Alors T et U sont bien une bipartition de S, et comme c'est un coloriage valide il n'y a pas d'arête entre deux sommets de T ou entre deux de U.

□

4.1 Couplage (dans un graphe biparti ou non)

Définition 82.
Soit $G = (S, A)$ un graphe non-orienté (pas forcément biparti). Un **couplage** est un ensemble $B \subseteq A$ d'arêtes dont les extrémités sont deux à deux distinctes.
Un couplage B est dit **parfait** si chaque sommet est incident à une arête du graphe.

Exemple. Pour choisir les duos de chambres à l'internat, on considère le graphe suivant :

- Chaque élève est un sommet
- Chaque arête est une possibilité de mettre deux élèves dans la même chambre. On peut imposer des contraintes en enlevant des arêtes, comme « pas de chambre mixte », ou « horaires de matin compatibles », etc.

Choisir les duos de chambre revient alors à choisir un couplage parmi ce graphe ! Notez que le graphe n'est pas biparti.

Exemple. Le couplage ci-dessous n'est pas parfait :

FIGURE 11.44 – Un couplage non-parfait dans un graphe (biparti)

Proposition 83.
Dans un graphe biparti, on peut trouver efficacemment un couplage de cardinal maximal.

Démonstration. MPI ! Mais si vous êtes impatient·es, il y a une résolution très bien expliquée dans cette leçon du Collège de France : `https://www.college-de-france.fr/fr/agenda/lecon-inaugurale/algorithmes/algorithmes` (à partir de 11min35). □

Remarque. Si je reprends mon exemple de l'internat, un couplage maximal est le remplissage du plus de chambres possibles (en respectant les contraintes)... c'est une objectif raisonnable.

Remarque. Une variante célèbre de ces problèmes de couplage dans des graphes bipartis consiste à donner des *préférences* à chaque noeud. C'est typiquement ce qui a lieu sur Parcoursup ! Les candidat·es ont des préférences vis à vis des formations, et les formations vis-à-vis des candidat·es. On cherche alors un couplage où deux candidat·es ne peuvent pas mutuellement améliorer leur satisfaction en échangeant leur formation ; on appelle cela un *couplage stable*. Pour en savoir plus : `https://fr.wikipedia.org/wiki/Probl%C3%A8me_des_mariages_stables`