Graphe et Langage

Mathieu Sablik

Table des matières

Chapitre I

Différentes notions de graphes

I.1 Différents problèmes à modéliser

On peut considérer que l'article fondateur de la théorie des graphe fut publié par le mathématicien suisse Leonhard Euler en 1741. Il traitait du problème des sept ponts de Königsberg : est il possible de réaliser une promenade dans la ville de Königsberg partant d'un point donné et revenant à ce point en passant une et une seule fois par chacun des sept ponts de la ville ?

Cette théorie va connaitre un essor au cours du $XIX^{\text{ème}}$ par l'intermédiaire du problème suivant : quel est le nombre minimal de couleurs nécessaires pour colorier une carte géographique de telle sorte que deux régions limitrophe n'ont pas la même couleur ? Le théorème des quatre couleurs affirme que seulement quatre sont nécessaires. Le résultat fut conjecturé en 1852 par Francis Guthrie, intéressé par la coloration de la carte des régions d'Angleterre, mais ne fût démontré qu'en 1976 par deux Américains Kenneth Appel et Wolfgang Haken. Ce fut la première fois que l'utilisation d'un ordinateur a permis de conclure leur démonstration en étudiant les 1478 cas particulier auxquels ils ont ramené le problème.

Au $XX^{\text{ème}}$ siècle, la théorie des graphes va connaître un essor croissant avec le développement des réseaux dont il faut optimiser l'utilisation. On peut citer quelques exemples de manière non exhaustive :

- réseaux de transports routier, d'eau, d'électricité : les sommets représentent les carrefours et les arêtes les rues ;
- réseaux informatiques : les sommets représentent les ordinateurs et les arêtes les connexions physiques ;
- réseaux sociaux : les sommets représentent les membres du groupe, deux personnes sont reliées par une arête si elles se connaissent (Facebook : graphe non orienté, twiter : graphe orienté, combien de poignées de main on est du président ?...) ;
- graphe du web : les sommets représentent les pages web et chaque arc correspond a un hyperliens d'une page vers une autre ;
- réseau de transports de données (téléphonie, wifi, réseaux informatique...) ;
- représentation d'un algorithme, du déroulement d'un jeu ;
- réseaux de régulation génétique ;

— organisation logistique : les sommets représentent des évènements, deux évènements sont reliées par une arête s'ils ne peuvent pas avoir lieu en même temps ;
— ordonnancement de projet : les sommets représentent les différentes tâches composant un projet, deux tâches sont reliés par une flèche si la deuxième ne peut pas commencer avant que la première soit terminée ;
— et beaucoup d'autres encore…

L'étude des graphes se réalise sous deux point de vues complémentaire. L'étude de propriétés structurelles de graphes ou de familles de graphes et l'étude algorithmique de certaines propriétés.

I.2 Différentes notions de graphes

I.2.1 Graphe orienté ou non

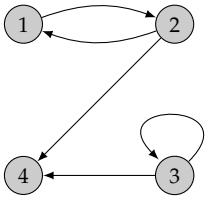
Dans les exemples que l'on a vus, un graphe est un ensemble fini de sommets reliés par des arêtes. Ces arêtes peuvent être orientées ou non, de plus une valeur peut être associée à chaque arête ou aux sommets.

Définition I.1. Un *graphe orienté* $G = (S, A)$ est la donnée :
— d'un ensemble S dont les éléments sont des sommets ;
— d'un ensemble $A \subset S \times S$ dont les éléments sont les arcs.

Un arc $a = (s, s')$ est aussi noté $s \to s'$, s est l'*origine* de a et s' l'*extrémité*. On dit aussi que s' est le *successeur* de s et s le *prédécesseur* de s'.

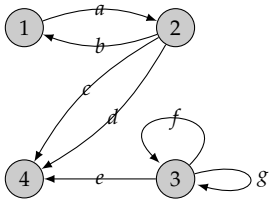
On peut souhaiter qu'il y ait plusieurs arcs entre deux mêmes sommets. On parle alors de graphe orienté *multi-arcs*. Formellement, $G = (S, A, \mathbf{i}, \mathbf{f})$ c'est la donnée :
— d'un ensemble S dont les éléments sont des sommets ;
— d'un ensemble A dont les éléments sont les arcs ;
— de deux fonctions $\mathbf{i} : A \to S$ et $\mathbf{f} : A \to S$ qui à chaque arcs $a \in A$ associe son prédécesseur $\mathbf{i}(a)$ et son successeur $\mathbf{f}(a)$.

Exemple I.1. Exemple de graphe orienté :



$G = (S, A)$ où
— $S = \{1, 2, 3, 4\}$,
— $A = \{(1,2), (2,1), (2,4), (3,4), (3,3)\}$.

Exemple de graphe orienté multi-arcs :



$G = (S, A, \mathbf{i}, \mathbf{f})$ où
— $S = \{1, 2, 3, 4\}$,
— $A = \{a, b, c, d, e, f, g, h\}$,
— $\mathbf{i}$: $\begin{array}{lcl} a & \mapsto & 1 \\ b & \mapsto & 2 \\ c & \mapsto & 2 \\ d & \mapsto & 2 \\ e & \mapsto & 3 \\ f & \mapsto & 3 \\ g & \mapsto & 3 \end{array}$ et $\mathbf{f}$: $\begin{array}{lcl} a & \mapsto & 2 \\ b & \mapsto & 1 \\ c & \mapsto & 4 \\ d & \mapsto & 4 \\ e & \mapsto & 4 \\ f & \mapsto & 3 \\ g & \mapsto & 3 \end{array}$.

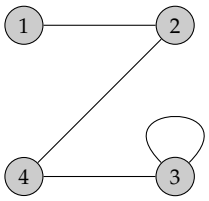
Définition I.2. Un *graphe non orienté* $G = (S, A)$ est la donnée :

— d'un ensemble S dont les éléments sont les sommets du graphe,
— d'un ensemble A dont les éléments, les arêtes du graphe, sont des parties à un ou deux éléments de S.

Le ou les sommets d'une arête sont appelés extrémités de l'arête. Les arêtes n'ayant qu'une seule extrémité sont des boucles.

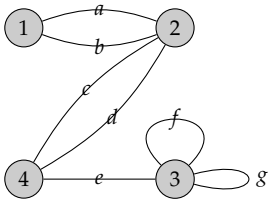
On peut de la même façon un graphe non-orienté *multi-arêtes*. Formellement, $G = (S, A, \alpha)$ est la donnée :
— d'un ensemble S dont les éléments sont des sommets ;
— d'un ensemble A dont les éléments sont les arêtes ;
— d'une fonction α de A dans les parties à un ou deux éléments de S.

Exemple I.2. Exemple de graphe non-orienté :



$G = (S, A)$ où
— $S = \{1, 2, 3, 4\}$,
— $A = \{\{1, 2\}, \{2, 4\}, \{3, 4\}, \{3\}\}$.

Exemple de graphe non orienté multi-arêtes :



$G = (S, A, \alpha)$ où
— $S = \{1, 2, 3, 4\}$,
— $A = \{a, b, c, d, e, f, g, h\}$,
— α : $\begin{array}{lcl} a & \mapsto & \{1,2\} \\ b & \mapsto & \{1,2\} \\ c & \mapsto & \{2,4\} \\ d & \mapsto & \{2,4\} \\ e & \mapsto & \{3,4\} \\ f & \mapsto & \{3\} \\ g & \mapsto & \{3\} \end{array}$.

Si un arc ou une arête à ses deux extrémités constituées du même sommet, on dit que c'est une *boucle*.

Un graphe est *simple* s'il est non-orienté, s'il a au plus une arête entre deux sommets et s'il n'a pas de boucle.

L'*ordre* d'un graphe est le nombre de sommets $|S|$ et la *taille* d'un graphe est le nombre d'arêtes ou d'arcs.

On appèle *valuation* sur les sommets (resp. sur les arcs ou arêtes) toutes fonctions prenant en argument les sommets (resp. sur les arcs ou arêtes) et renvoyant un réels ou élément dans un ensemble donné.

Soit $G = (S, A)$ un graphe orienté, on associe le graphe non orienté $G' = (S, A')$ ayant le même ensemble de sommets S et dont l'ensemble d'arêtes A' vérifie $\{x, y\} \in A' \iff (x, y) \in A$ ou $(y, x) \in A$.

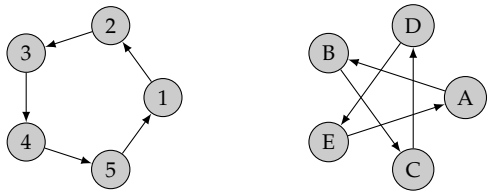
Exemple I.3. Les trois graphes suivants

sont associés au graphe non orienté suivant

I.2.2 Isomorphisme de graphe

Deux graphes orientés $G = (S, A)$ et $G' = (S', A')$ sont *isomorphes* s'il existe une application bijective $\varphi : S \to S'$ telle que pour tout $s, s' \in S$ on $(s, s') \in A \iff (\varphi(s), \varphi(s')) \in A$. L'application φ est alors un *isomorphisme de graphes orientés*.

Exemple I.4. Les deux graphes suivants sont isomorphes par l'isomorphisme $\varphi : 1 \mapsto A, 2 \mapsto B, 3 \mapsto C, 4 \mapsto D, 5 \mapsto E$.



De même, deux graphes non-orientés $G = (S, A)$ et $G' = (S', A')$ sont *isomorphes* s'il existe une application bijective $\varphi : S \to S'$ telle que pour tout $s, s' \in S$ on $\{s, s'\} \in A \iff \{\varphi(s), \varphi(s')\} \in A$. L'application φ est alors un *isomorphisme de graphes non-orientés*.

I.2.3 Degré

Pour un graphe orienté, on appèle *degré entrant* d'un sommet s, noté $d_-(s)$ (resp. *degré sortant* d'un sommet s, noté $d_+(s)$) le nombre d'arcs dont le sommet est prédécesseur (resp. successeur).

Pour un graphe non-orienté, on appelle *degré* d'un sommet s, noté $d(s)$ le nombre d'arêtes dont le sommet est une extrémité.

Théorème I.1 Lemme de la poignée de main

Soit $G = (S, A)$ un graphe orienté. On alors les égalités suivantes :

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

Soit $G = (S, A)$ un graphe non-orienté. On a alors l'égalité suivante :

$$\sum_{s \in S} d(s) = 2|A|.$$

Démonstration : Pour un graphe orienté $G = (S, A)$, chaque arc a un successeur et un prédécesseur d'ou la première égalité.

Pour obtenir la deuxième égalité, il suffit d'orienté le graphe non-orienté et remarquer que pour chaque sommet $d(s) = d_+(s) + d_-(s)$. ■

Une conséquence directe de ce théorème est que dans un graphe, le nombre de sommets dont le degré est impair est toujours pair.

Corollaire I.2

Dans un graphe, le nombre de sommets dont le degré est impair est toujours pair.

I.2.4 Construction de graphes à partir d'un autre

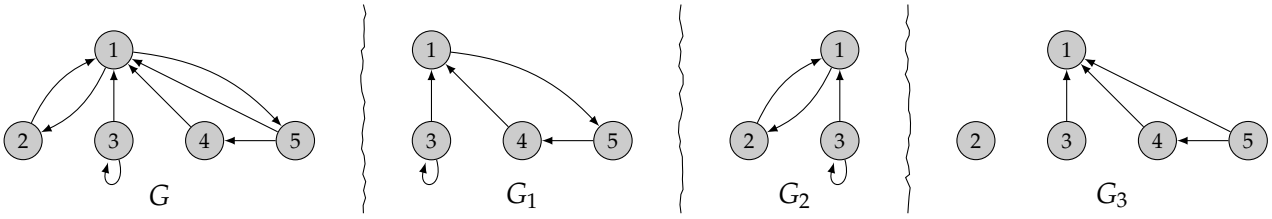
Soit $G = (S, A)$ un graphe (orienté ou non).

Un *sous-graphe* de G est un graphe $G' = (S', A')$ tel que $S' \subset S$ et $A' \subset A$.

Un sous-graphe $G' = (S', A')$ d'un graphe $G = (S, A)$ est un *sous-graphe induit* si A' est formé de tous les arcs (ou arêtes) de G ayant leurs extrémités dans S' (c'est à dire $\forall s, s' \in S', (s, s') \in A'$ si et seulement si $(s, s') \in A$).

Un sous-graphe $G' = (S', A')$ d'un graphe $G = (S, A)$ est *couvrant* s'il contient tous les sommets de G (c'est à dire $S' = S$).

Exemple I.5. On considère un graphe G, un sous-graphe quelconque G_1, un sous-graphe induit G_2 et un sous-graphe couvrant G_3.



I.3 Différents modes de représentation d'un graphe

Compte tenu de l'essor des graphes en informatique, il est naturel de s'intéresser aux différentes manières de les représenter. Différents modes de représentation peuvent être envisagées suivant la nature des traitements que l'on souhaite appliquer aux graphes considérés.

I.3.1 Représentation sagittale

La représentation sagittale est la représentation sous forme d'un dessin. Un même graphe peut avoir des représentations sagittales en apparence très différentes.

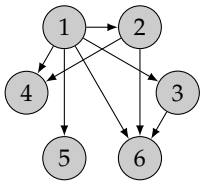
I.3.2 Définition par propriété caractéristique

Une même propriété caractérise les relation entre les différents sommets.

Exemple I.6. On considère le graphe $G = (S, A)$ avec $S = \{1, 2, 3, 4, 5, 6\}$ et pour tout $s, s' \in S$ on a

$$(s, s') \in A \iff s \text{ divise strictement } s'.$$

Sa représentation sagittale est :



I.3.3 Listes d'adjacence

Un graphe peut être représenté à l'aide d'un *dictionnaire* : il s'agit d'une table à simple entrée où chaque ligne correspond à un sommet et comporte la liste des successeurs (ou des prédécesseurs) de ce sommet.

En pratique pour stocker un graphe orienté $G = (S, A)$, on ordonne les sommets $s_1, \ldots, s_n$ et le graphe G est représenté par deux listes d'adjacences (LS, TS) définies par :

— LS : liste de longueur $|A|$ appelé *liste des successeurs*, elle contient les successeurs du sommets s_1, puis ceux de s_2 jusqu'à ceux de s_n, si un sommet n'a pas de successeur, on passe au sommet suivant.

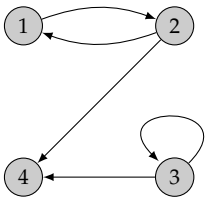
— TS : liste de longueur $|S| + 1$ appelé *liste des têtes successeurs* qui indique la position du premier successeur de chaque sommet dans LS. La liste TS est définit comme suit :

  — $TS(1) = 1$;

  — pour $s_i \in S$, si s_i a un successeur alors $TS(s_i)$ est le numéro de la case de LS du premier successeur de s_i, sinon $TS(s_i) = TS(s_{i+1})$;

  — $TS(n + 1) = |A| + 1$

Exemple I.7. Pour décrire un graphe, il suffit de donner le dictionnaire des successeurs ou bien le dictionnaire des prédécesseurs.



Sommets	Successeurs
1	2
2	1,4
3	3,4
4	∅

Sommets	Prédecesseurs
1	2
2	1
3	3
4	2,3

La représentation sous forme de liste est :
$LS = (2, 1, 4, 3, 4)$ $TS = (1, 2, 4, 6, 6)$

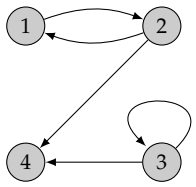
I.3.4 Matrices d'adjacence

Soit $G = (S, A)$ un graphe dont les sommets sont numérotés de 1 à n. La *matrice d'adjacence* de G est la matrice carrée $(m_{i,j})_{(i,j)\in[1,n]^2}$ définie par

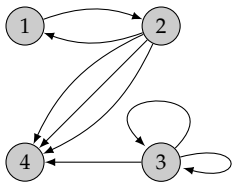
$$m_{i,j} = \begin{cases} k \text{ s'il y a } k \text{ arêtes allant de } i \text{ à } j \\ 0 \text{ sinon} \end{cases}$$

Si le graphe n'est pas orienté, la matrice est symétrique.

Exemple I.8. Exemples de matrices d'adjacence de graphes orientés :

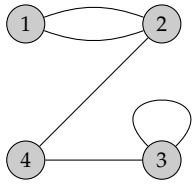


$$M = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

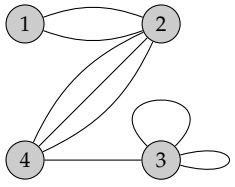


$$M = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 3 \\ 0 & 0 & 2 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

et de graphes non orientés associés :



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



$$M = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 3 \\ 0 & 0 & 2 & 1 \\ 0 & 3 & 1 & 0 \end{pmatrix}$$

I.3.5 Matrice d'incidence

La *matrice d'incidence* d'un graphe orienté $G = (S, A)$ est une matrice à coefficients dans $\{-1, 0, 1\}$ indicée par l'ensemble $S \times A$ tel que pour $(i, j) \in S \times A$ on a $m_{i,j} = 1$ si le sommet i est l'extrémité de l'arête j, $m_{i,j} = -1$ si i est l'origine de j, et 0 sinon. On remarque que, puisque chaque colonne correspond à une arête, il doit y avoir exactement un 1 et un -1 sur chaque colonne.

I.3.6 Comparaison des différentes méthodes

On s'intéresse ici à l'espace nécessaire pour stocker un graphe $G = (S, A)$, les différentes méthodes ont leurs avantages et inconvénients. En voici un aperçu :

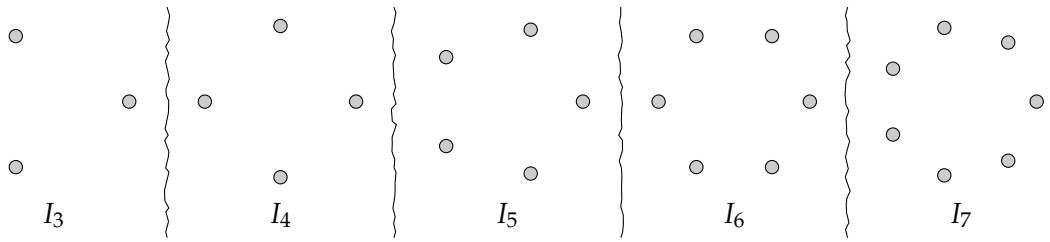
Méthode de représentation	Espace de stockage	Autre avantage
Liste des arcs	$2\|A\|$	
Liste d'adjacence	$\|S\| + \|A\| + 1$	- efficace pour stocker des graphes creux - efficace pour implémenter des algorithmes de parcours (section III.3)
Matrice d'adjacence	$\|S\|^2$	- efficace pour stocker des graphes denses - donne des informations sur la longueur d'un chemin (section II.1.2)
Matrice d'incidence	$\|S\| \times \|A\|$	- utiliser pour le calcule de circuit électique

I.4 Quelques classes de graphe importantes

On s'intéresse ici à définir quelques classes de graphes non-orientés dont la plupart sont simple (non multi-arête et sans boucle).

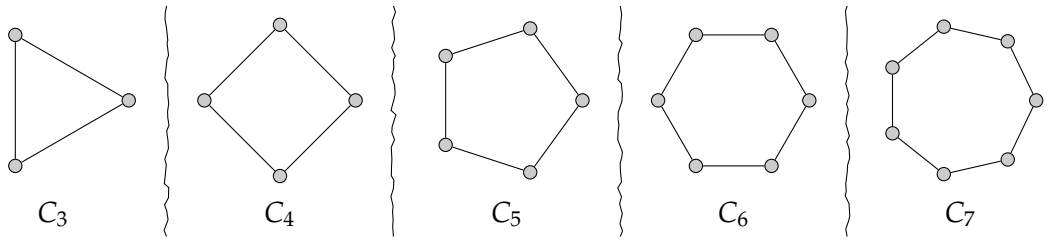
I.4.1 Graphes isolés

Le *graphe isolé d'ordre n* est un graphe à n sommets sans arête, on le note I_n.



I.4.2 Graphes cycliques

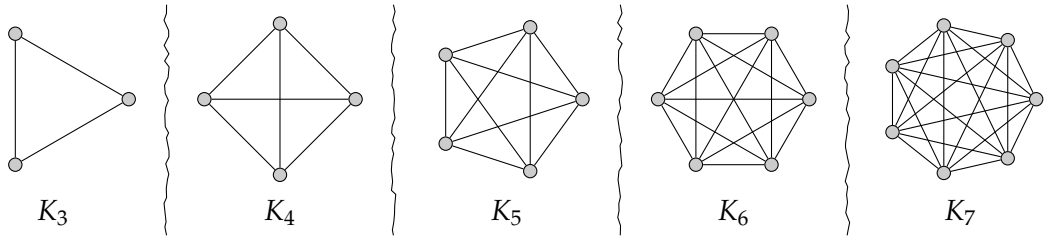
Le *graphe cyclique d'ordre n* est le graphe à n sommets $S = \{s_1, \ldots, s_n\}$ tels que les arêtes sont $A = \{\{s_i, s_{i+1}\} : i \in [1, n]\} \cup \{\{s_n, s_1\}\}$, on le note C_n.



C_3 C_4 C_5 C_6 C_7

I.4.3 Graphes complets

Le *graphe complet d'ordre n* est le graphe simple à n sommets dont tous les sommets sont reliés deux à deux, on le note K_n.

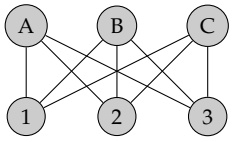


K_3 K_4 K_5 K_6 K_7

I.4.4 Graphe biparti

Un graphe est *biparti* s'il existe une partition de son ensemble de sommets en deux sous-ensembles X et Y telle que chaque arête ait une extrémité dans X et l'autre dans Y.

On définit le *graphe biparti complet* entre un ensemble de n sommets et un ensemble à m sommets comme le graphe simple tel que chaque sommet du premier ensemble est relié à chaque sommet su deuxième ensemble. On le note $K_{n,m}$.

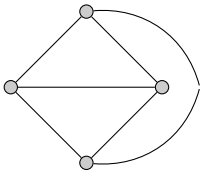


$K_{3,3}$

I.4.5 Graphes planaires

Un graphe non-orienté (pas forcément simple) est *planaire* s'il admet une représentation sagittale dans un plan sans que les arêtes se croisent.

Exemple I.9. K_4 est planaire puisque on peut le représenter de la façon suivante :



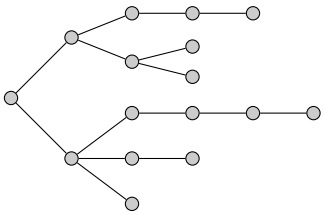
K_4

Est ce que K_5 et $K_{3,3}$ sont planaires ?

I.4.6 Arbres

Définition I.3. Un *arbre* se définit de manière inductive par :
- le graphe formé par un sommet est un arbre ;
- si $G = (S, A)$ est un arbre, alors pour $s \in S$ et x un élément quelconque n'appartenant pas à S, le graphe $G' = (S \cup \{x\}, A \cup \{\{x, s\}\})$ est un arbre.

Un exemple d'arbre :



Remarque I.1. A la section III on verra une définition équivalente liée à la connexité.

Chapitre II

Problèmes de chemins dans un graphe

II.1 Notion de chemin

II.1.1 Définitions

Définition II.1. Soit $G = (S, A)$ un graphe orienté (resp. non-orienté). Un *chemin* (resp. une *chaîne*) dans G est une suite de sommets $C = (s_0, s_1, s_2, \ldots, s_k)$ telle qu'il existe un arc (resp. une arête) entre chaque couple de sommets successifs de C. Ce qui s'écrit :

— si $G = (S, A)$ est orienté alors pour tout $i \in [0, k-1]$ on a $(s_i, s_{i+1}) \in A$,

— si $G = (S, A)$ est non-orienté alors pour tout $i \in [0, k-1]$ on a $\{s_i, s_{i+1}\} \in A$,

On appellera :

Chemin (resp. chaîne) simple **:** un chemin (resp. chaîne) dont tous les arcs (resp. arêtes) sont différents.

Chemin (resp. chaîne) élémentaire **:** un chemin (resp. chaîne) dont tous les sommets sont différents sauf peut être le départ et l'arrivée (pour autoriser les circuits ou cycles).

Circuit dans un graphe orienté **:** un chemin simple finissant à son point de départ.

Cycle dans un graphe non-orienté **:** une chaîne simple finissant à son point de départ.

II.1.2 Longueur d'un chemin

Longueur du chemin (de la chaîne) **:** nombre d'arcs (ou arêtes) du chemin.

Distance entre deux sommets **:** longueur du plus petit chemin (chaîne) entre ces deux sommets.

Diamètre d'un graphe **:** plus grande distance entre deux sommets de ce graphe.

Remarque II.1. Dans le cas d'un graphe valué où l'on associe un réel à chaque arcs (ou arêtes), la longueur d'un chemin correspond à la somme des valeur de chaque arcs (ou arêtes) du chemin.

Exemple II.1. On peut calculer le diamètre des graphes classiques :

— diamètre de K_n : 1 ;

— diamètre de $K_{n,m}$: 2 ;

— diamètre de C_n : $\lfloor \frac{n}{2} \rfloor$.

II.1.3 Longueur d'un chemin et matrice d'adjacence

On cherche a déterminer le nombre de chemins (resp. chaînes) de longueur n reliant deux sommets d'un graphe G. On note M la matrice d'adjacence de G.

Proposition II.1

Soit $G = (S, A)$ un graphe de matrice d'adjacence M, le nombre de chemins (resp. chaînes) de longueur n reliant le sommets i au sommet j correspond au coefficient d'indice (i, j) de la matrice M^n.

Démonstration : *Initialisation :* Les chemins (resp. chaînes) de longueur 1 qui joignent i à j correspondent au coefficient d'indice (i, j) de la matrice d'adjacence M.

Induction : On suppose que le nombre de chemins (resp. chaînes) de longueur n qui joignent deux sommets quelconques i à j correspond au coefficient $M^n_{(i,j)}$. Soit i, j, k trois sommets, le nombre de chemins (resp. chaînes) de longueur $n+1$ allant de i à j tels que le premier arc (resp. arête) soit (i, k) (resp. $\{i, k\}$) correspond au nombre de chemins (resp. chaînes) de longueur 1 allant de i à k fois le nombre de chemins (resp. chaînes) de longueur n allant de k à j, c'est à dire $M_{(i,k)}M^n_{k,j}$. Ainsi le nombre de chemins (resp. chaînes) de longueur $n+1$ qui joignent deux sommets i à j est :

$$\sum_{k \in S} M_{(i,k)} M^n_{(k,j)} = M^{n+1}_{(i,j)}.$$

■

On en déduit une méthode pour calculer la distance entre deux sommets ainsi que le diamètre d'un graphe.

Proposition II.2

Soit $G = (S, A)$ un graphe de matrice d'adjacence M.

La distance entre deux sommets i et j est le plus petit $n \in \mathbb{N}$ tel que le coefficient d'indice (i, j) de M^n soit non nul.

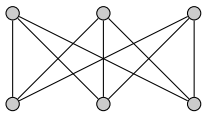
Le diamêtre de G est le plus petit $n \in \mathbb{N}$ tel que tous les coefficients de $(M + Id)^n$ soient non nul.

Démonstration : Seul le deuxième point est non trivial. Cela vient du fait que

$$(M + Id)^n = \sum_{r=0}^{n} C^r_n A^r.$$

■

Exemple II.2. On cherche à compter le nombre de cycles de longueur k dans $K_{n,n}$. Par exemple, pour $n = 3$ on a le graphe suivant :



$$A = \begin{pmatrix} 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 \end{pmatrix}$$

Si k est pair : $A^k = \left(\begin{array}{c|c} n^{k-1} & 0 \\ \hline 0 & n^{k-1} \end{array}\right)$ et si k est impair : $A^k = \left(\begin{array}{c|c} 0 & n^{k-1} \\ \hline n^{k-1} & 0 \end{array}\right)$

Comme $A^k_{(i,j)}$ correspond au nombre de chemin de i à j de longueur k, le nombre de cycles de longueur k est donc :

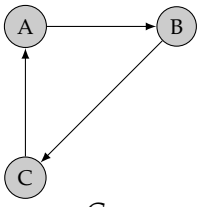
$$\sum_{i=1}^{2n} A^k_{i,i} = \begin{cases} 2n \times n^{k-1} = 2n^k & \text{si } k \text{ est pair} \\ 0 & \text{sinon.} \end{cases}$$

II.2 Connexité

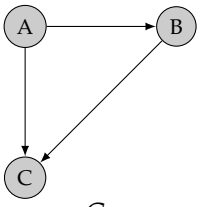
Définition II.2 (Connexité et forte connexité)**.** Un graphe non-orienté est *connexe* si pour tout couple de sommets s et s', il existe une chaîne reliant s à s'.

Un graphe orienté est connexe si le graphe non orienté associé est connexe. Un graphe orienté est *fortement connexe* si pour tout couple de sommets s et s', il existe une chemin reliant s à s'.

Exemple II.3 (Graphe connexe et fortement connexe). G_1 est fortement connexe tandis que G_2 est connexe mais non fortement connexe.



G_1



G_1

Définition II.3 (Composantes connexes et fortement connexes)**.** Une composante connexe (resp. fortement connexe) C d'un graphe $G = (S, A)$ est un sous-ensemble maximal de sommets tels que deux quelconques d'entre eux soient reliés par une chaîne (resp. un chemin). Formellement, si $s \in C$ alors on a :

— pour tout $s' \in C$ il existe une chaîne (resp. un chemin) reliant s à s',
— pour tout $s' \in S \setminus C$, il n'existe pas de chaîne (resp. chemin) reliant s à s'.

Quelques propriétés :

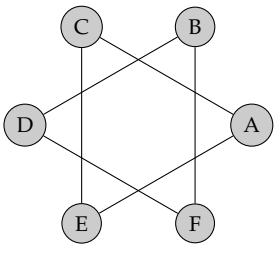
— Les composantes connexes (resp. fortement connexe) d'un graphe $G = (S, A)$ forment une partition de S.
— Un graphe est connexe (resp. fortement connexe) si et seulement s'il a une seule composante connexe (resp. fortement connexe).
— Le sous-graphe induit par une composante connexe (resp. fortement connexe) est connexe (resp. fortement connexe).
— La composante connexe C qui contient un sommet $s \in S$ est

$$C = \{s' \in S \text{il existe une chaîne reliant } s \text{ à } s'\}$$

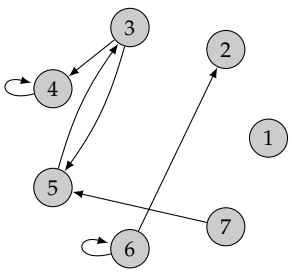
— La composante fortement connexe C qui contient un sommet $s \in S$ est

$$C = \{s' \in S \text{il existe un chemin reliant } s \text{ à } s' \text{ et un chemin reliant } s' \text{ à } s\}$$

Exemple II.4 (Composantes connexes). Les composantes connexes de G_1 sont $\{A, C, E\}$ et $\{B, D, F\}$ tandis que celles de G_4 sont $\{1\}$, $\{2, 6\}$, $\{3, 5, 7\}$ et $\{4\}$.

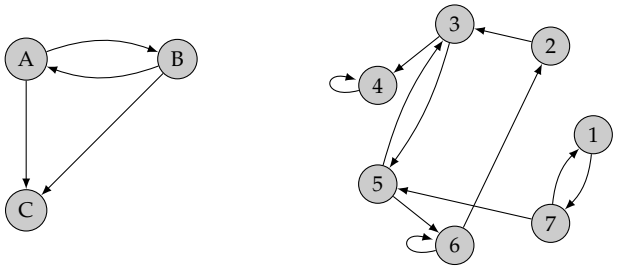


G_1



G_2

Exemple II.5 (Composantes fortement connexes). Les composantes fortement connexes de G_1 sont $\{A, B\}$ et $\{C\}$ tandis que celles de G_2 sont $\{1, 7\}$, $\{2, 3, 5, 6\}$ et $\{4\}$.



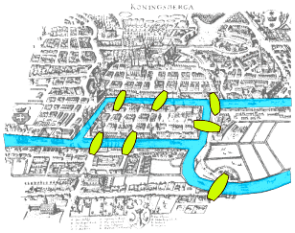
G_1 G_2

II.3 Chemin Eulérien et Hamiltoniens

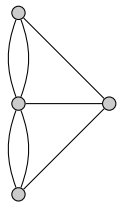
II.3.1 Chemin Eulérien

Problématique

Au $XVIII^{\text{ème}}$ siècle un casse-tête est populaire chez les habitants de Königsberg : est-il possible de se promener dans la ville en ne passant qu'une seule fois par chacun des sept ponts de Königsberg ? C'est le célèbre mathématicien Euler qui montre le premier que ce problème n'a pas de solution, en utilisant pour la première fois la notion de graphe. Le problème se reformule ainsi en terme de graphes : existe-t-il un cycle qui passe exactement une fois par toutes les arêtes dans le graphe (multi-arête) ci-dessous ?



Ville de Königsberg

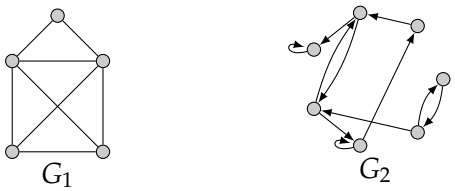


G

Définition II.4. Soit G un graphe non orienté. Une chaîne (resp. un cycle) *eulérienne* est une chaîne (resp. un cycle) qui passe une et une seule fois par toutes les arêtes de G.

On définit les mêmes notions pour un graphe orienté G : un chemin (resp. un circuit eulérien) est un chemin (resp. un circuit) passant une et une seule fois par tous les arcs de G.

Exemple II.6. Le graphe G_1 admet un cycle eulérien. Le graphe G_2 admet un chemin eulérien mais pas un circuit.



G_1 G_2

Caractérisation des chemins eulériens

Avant de prouver la caractérisation des chemin eulériens, on a besoin du résultat suivant.

Proposition II.3

Un graphe dont tous les sommets sont de degré supérieur ou égal à 2 possède au moins un cycle.

Démonstration : La preuve utilise un algorithme de marquage. Initialement tous les sommets sont non marqués. Un sommet s_1 est marqué arbitrairement. L'algorithme construit alors une séquence $s_1, \ldots, s_k$ de sommets marqués en choisissant arbitrairement pour s_{i+1} un sommet non marqué adjacent à s_i. L'algorithme s'arrête lorsque s_k ne possède plus de voisin non marqué. Puisque ce sommet est de degré au moins 2, il possède un voisin $s_j \neq s_{k-1}$ dans la séquence, $j < k-1$. On en déduit que $(s_k, s_j, s_{j+1}, \ldots, s_{k-1}, s_k)$ est un cycle. ■

Théorème II.4

Soit $G = (S, A)$ un graphe non orienté connexe. Il admet un cycle eulérien si et seulement si $d(s)$ est pair pour tout $s \in S$.

Si seulement deux sommets ne vérifient pas les conditions précédentes alors G admet une chaîne Eulériene.

Démonstration : Soit $G = (S, A)$ un graphe connexe. Pour qu'il admette un cycle Eulérien il faut qu'en chaque sommet lorsqu'on arrive par une arête on puisse repartir par un autre arête. On obtient donc que $d(s)$ est pair si le graphe est orienté pour chaque sommet $s \in S$.

Réciproquement, on démontre par récurrence sur le nombre d'arcs que pour un graphe connexe G, si chaque sommet $s \in S$ est de degré pair alors G admet un cycle eulérien.

Initialisation : Si $|A| = 0$, on a un graphe connexe sans arêtes, c'est à dire un seul sommet isolé qui admet un cycle eulérien.

Induction : On suppose que le théorème est vrai pour tout graphe ayant un nombre d'arêtes inférieur ou égal à n (hypothèse de récurrence forte). Soit $G = (S, A)$ un graphe connexe tel que $|A| = n + 1$ et pour chaque sommet $s \in S$ est de degré pair. Comme le graphe est connexe et que le degré de chaque sommet est pair, on en déduit que G admet un cycle élémentaire $C = (s_1, s_2, \ldots, s_k, s_1)$.

Soit G' le sous-graphe de G auquel on a supprimé les arêtes de C. Le graphe G' n'est pas forcément connexe mais vérifie $d(s)$ pairs pour chacun de ses sommet s. On applique l'hypothèse de récurrence sur chacune de ses composantes qui admettent donc des cycles eulériens. On combine alors ces différents cycles eulériens avec le cycle C, pour former un cycle eulérien sur G de la façon suivante : on parcourt C depuis un sommet initial arbitraire et, à chaque fois que l'on rencontre une des composantes connexes de G' pour la première fois, on insère le cycle eulérien considéré sur cette composante. S'agissant d'un cycle, on est assuré de pouvoir poursuivre le parcours de C après ce détour. Il est facile de vérifier qu'on a ainsi bien construit un cycle eulérien sur G.

Si G admet une chaîne Eulérienne et admet un sommet de degré impair, soit c'est le point de départ de la chaîne, soit il arrive un moment où l'on ne pourra plus repartir ce qui constitue le sommet terminal de la chaîne. Ainsi, si seulement deux sommets sont de degré impair il peuvent servir de point de départ et d'arrivé d'un chemin passant par tous les arêtes du graphe, le graphe peut donc admettre une chaîne Eulérienne. ■

Dans le cas orienté on montre de manière similaire le résultat suivant.

Théorème II.5

Soit $G = (S, A)$ un graphe orienté fortement connexe. Il admet un circuit eulérien si et seulement si $d_+(s) = d_-(s)$ pour tout $s \in S$.

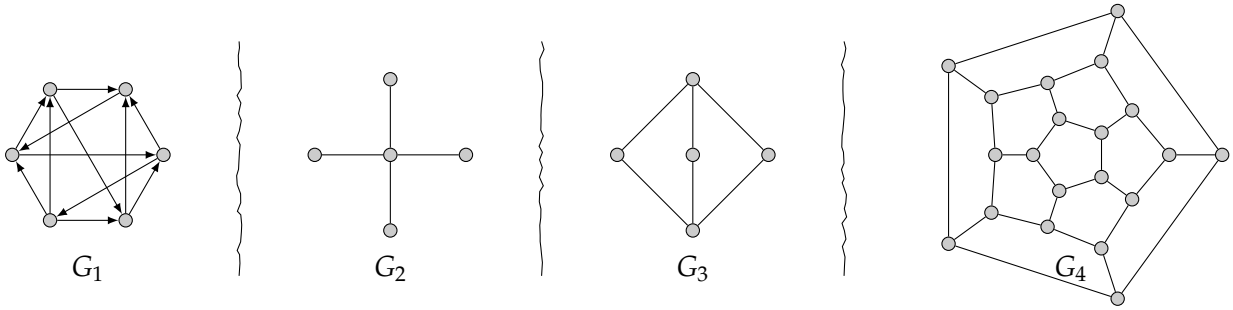
Si seulement deux sommets vérifient $|d_+(s) - d_-(s)| = 1$ *alors G admet un chemin Eulérien.*

II.3.2 Chemins hamiltonien

Définition II.5. Soit G un graphe non orienté. Un *cycle* (respectivement une *chaîne*) *hamiltonien* est un cycle (resp. une chaîne) qui passe une et une seule fois par tous les sommets de G.

On définit les mêmes notions pour un graphe orienté G : un *circuit* ou un *chemin hamiltonien* est un circuit ou un chemin passant une et une seule fois par tous les sommets de G

Exemple II.7. G_1 admet un circuit hamiltonien, G_2 n'admet ni chaîne ni cycle hamiltoniens, G_3 admet une chaîne hamiltonienne mais pas de cycles hamiltoniens et G_4 admet un cycle hamiltonien.



On ne connaît pas de condition nécessaire et suffisante exploitable dans la pratique pour décider si un graphe est hamiltonien ou non. De manière générale, la recherche de cycle, chaîne, circuit ou chemin Hamiltonien est un problème algorithmiquement difficile. En fait, on peut montrer que c'est un problème NP-complet.

II.4 Deux mots sur le Page-rank

To do:

Chapitre III

Graphes acycliques ou sans-circuits

III.1 Notion d'arbres

III.1.1 Nombre d'arêtes d'un graphe acyclique

Proposition III.1

Un graphe connexe d'ordre n comporte au moins $n-1$ arêtes.

Démonstration : On montre le résultat récurrence sur l'ordre du graphe n.

Initialisation : Le résultat est évident pour $n = 1$ et $n = 2$.

Induction : Supposons la propriété prouvée sur les graphes connexes d'ordre n. Soit $G = (S, A)$ un graphe connexe à $n+1$ sommets. La connexité assure que chaque sommet est de degré au moins 1. On a alors deux cas :

— si chaque sommet est de degré au moins 2, alors le lemme de la poignée de main conduit à $2|A| = \sum_{s \in S} d(S) \geq 2n$ donc $|A| \geq n$;

— s'il existe un sommet s de degré 1 alors, le graphe induit G' obtenu en éliminant s et l'arête dont il est l'extrémité, est un graphe connexe de n sommets qui possède exactement une arête de moins que G. D'après l'hypothèse de récurrence, G' possède donc au moins $n-1$ arêtes, d'où G en possède au moins n. ■

Proposition III.2

Un graphe dont tous les sommets sont de degré supérieur ou égal à 2 possède un cycle. En particulier, un graphe acyclique admet un sommet de degré 0 ou 1.

Démonstration : La preuve utilise un algorithme de marquage. Initialement tous les sommets sont non marqués. Un sommet s_1 est marqué arbitrairement. L'algorithme construit alors une séquence $s_1, \ldots, s_k$ de sommets marqués en choisissant arbitrairement pour s_{i+1} un sommet non marqué adjacent à s_i. L'algorithme s'arrête lorsque s_k ne possède plus de voisin non marqué. Puisque ce sommet est de degré au moins 2, il possède un voisin $s_j \neq s_{k-1}$ dans la séquence, $j < k-1$. On en déduit que $(s_k, s_j, s_{j+1}, \ldots, s_{k-1}, s_k)$ est un cycle. ■

Nous pouvons lier cette fois l'absence de cycle dans un graphe avec le nombre d'arêtes.

Proposition III.3

Un graphe acyclique à n sommets possède au plus $n-1$ arêtes.

Démonstration : On va montrer cette propriété par récurrence sur le nombre de sommets du graphe $G = (S, A)$.

Initialisation : Si G est d'ordre 1, comme G est acyclique il n'y a pas de boucle, il ne possède donc aucune arête et la propriété est vérifiée.

Induction : Supposons la propriété vrai au rang n et montrons la au rang $n+1$. Comme G est acyclique, par la propriété III.2, il existe un sommet s de degré 0 ou 1. Considérons le graphe induit G' par les sommets $S \setminus \{s\}$. Ce graphe est acyclique et possède n sommets, par hypothèse d'induction G' a au plus $n-1$ arêtes. On en déduit que G a au plus n arêtes car $d(s) \leq 1$. ■

III.1.2 Arbres et forêts

Définition III.1. Un *arbre* est un graphe non orienté, connexe, sans cycle.

Une *forêt* est un graphe non orienté sans cycle (chacune de ses composantes connexes est un arbre).

Les sommets de degré 1 ou 0 sont appelés *feuilles*, les autres sommets sont appelés *noeuds*.

Théorème III.4

Soit G un graphe non orienté à n sommets. Les propositions suivantes sont équivalentes :

- *G est connexe sans cycle ;*
- *G est connexe et a $n-1$ arêtes ;*
- *G est connexe et la suppression de n'importe quelle arête le déconnecte ;*
- *G est sans cycle et a $n-1$ arêtes ;*
- *G est sans cycle et l'ajout de n'importe quel arête crée un cycle ;*
- *entre toute paire de sommets de G il existe une unique chaîne élémentaire ;*
- *G est défini de manière inductive comme à la définition I.3.*

Théorème III.5

Tout graphe connexe peut s'obtenir par ajout d'un certain nombre d'arêtes à un arbre ayant le même nombre de sommets.

Démonstration : On raisonne par récurrence sur le nombre n de cycles élémentaires du graphe.

Initialisation : Si n = 0, le graphe est connexe et sans cycle, c'est donc un sommet isolé, il s'agit donc d'un arbre.

Induction : Supposons que le résultat soit établi pour tout graphe connexe n'ayant pas plus de n cycles élémentaires. Soit G un graphe connexe avec $n+1$ cycles élémentaires. On considère alors le sous-graphe G' obtenu en enlevant uniquement une arête (s_1, s_2) qui appartient à un cycle $(s_1, s_2, \ldots, s_k)$. Il est clair qu'ainsi on brise au moins un cycle élémentaire parmi ceux de G. De plus, tous les cycles de G' sont des cycles de G, donc G' possède au plus n cycles élémentaires (peut-être en a-t-on brisé plus d'un). De plus, le graphe G' est encore connexe, puisque si l'on veut passer de s_1 à s_2, il suffit de faire le tour via le chemin $(s_2, s_3, \ldots, s_k, s_1)$. D'après l'hypothèse de récurrence, on sait que G' peut-être obtenu à partir d'un arbre T par ajout d'un certain nombre d'arêtes. Il suffit alors d'ajouter l'arête (s_1, s_2) pour retrouver G, ce qui achève la démonstration. ■

III.1.3 Arbres orientés

Les arbres utilisés en algorithmique ont le plus souvent une orientation et un sommet qui joue un rôle particulier, la racine : c'est ce type d'arbre que l'on va voir maintenant.

Définition III.2. Un graphe non orienté est un *arbre enraciné* s'il est connexe sans cycle et si un sommet particulier a été distingué, on l'appellera la racine.

Un arbre enraciné est souvent muni d'une orientation naturelle : on oriente chaque arête de telle sorte qu'il existe un chemin de la racine à tout autre sommet. Le graphe orienté résultant est aussi appelé *arbre orienté*.

Proposition III.6

Un graphe orienté est un arbre enraciné si et seulement si
— *il est connexe,*
— *il a un unique sommet sans prédécesseur (la racine),*
— *et tous ses autres sommets ont exactement un prédécesseur.*

Remarque III.1. Un graphe orienté sans circuit n'est pas forcément un arbre orienté.

On appellera :
— *racine de l'arbre* : le sommet qui n'a pas de prédécesseur
— *feuilles de l'arbre* : les sommets qui n'ont pas de successeur ;
— *nœuds de l'arbre* : tous les autres sommets ;
— *branche de l'arbre* : tout chemin de la racine vers une feuille,
— *descendant de s* : les successeurs de s,
— *ascendant de s* : le prédécesseur de s.

Lorsque chaque sommet a au plus 2 successeurs on parle aussi d'arbre binaire.

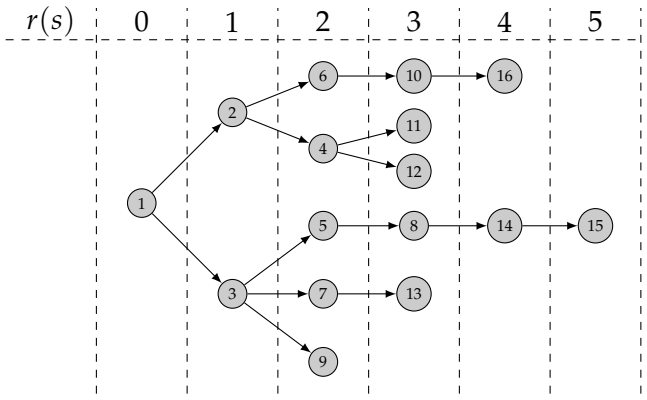
Proposition III.7

Un arbre à n sommets peut être défini par une liste de n éléments, appelé liste des prédécesseurs*, qui contient le prédécesseur de chaque sommet (ou $\emptyset$ pour la racine de l'arbre) :*

$$\text{pour tout } s \in S \text{ on pose } \mathbf{Pred}(s) = \begin{cases} \emptyset & \text{si } s \text{ est la racine de l'arbre} \\ s' & \text{si } (s', s) \end{cases}$$

Ainsi stocker un arbre n'est pas trop gourmand d'un point de vu informatique.

Exemple III.1. La liste $\mathbf{Pred} = [\emptyset, 1, 1, 2, 3, 2, 3, 5, 3, 6, 4, 4, 7, 8, 14, 10]$ représente l'arbre suivant :



Le sommet 1 est la racine, les sommets 9, 11, 12, 13, 15 et 16 sont les feuilles, une branche de l'arbre est $(1, 3, 5, 8, 14, 15)$.

III.1.4 Notion de rang dans un graphe orienté sans circuit

Théorème III.8

Un graphe orienté G est sans circuit si et seulement si on peut attribuer à chaque sommet s un nombre $r(s)$, appelé le rang *de s, tel que pour tout arc (s, t) de G on ait $r(s) < r(t)$.*

Démonstration : Si $G = (S, A)$ comporte un circuit C, il n'est pas possible de trouver une telle fonction $r : S \to \mathbb{R}$. Sinon, il existe $t \in S$ tel que $r(t) = max\{r(s) : s \in C\}$ et en considérant l'arc $(t, u) \in C$, on aurait $r(t) \leq r(u)$ ce qui est en contradiction avec la définition du rang.

Réciproquement, si G n'a pas de circuit, il existe au moins un sommet sans prédécesseur dans G (sans cela, en remontant successivement d'un sommet à un prédécesseur, on finirait par fermer un circuit). Ainsi, on peut attribuer séquentiellement des valeurs aux sommets du graphe à l'aide de l'algorithme 1, ce qui conclura la démonstration. ■

Algorithm 1: Algorithme de calcul du rang

Data: Un graphe orienté sans circuit $G = (S, A)$
Result: Une fonction rang $r : S \to \mathbb{N}$ de G

rang$\leftarrow 0$;
$X \leftarrow S$;
$R \leftarrow$ ensemble des sommets de X sans prédécesseur dans X ;
while $X \neq \emptyset$ **do**
  $r(v) \leftarrow$ rang pour tout sommet $v \in R$;
  $X \leftarrow X \setminus R$;
  $R \leftarrow$ les sommets sans prédécesseur du graphe induit par les sommets X ;
  $r \leftarrow r + 1$;

III.2 Initiation à la théorie des jeux

III.2.1 Jeux combinatoires

Voici un jeu simple qui se joue à deux, sur un graphe orienté :
— On place un pion sur un sommet du graphe.
— A tour de rôle, chaque joueur doit déplacer le pion en suivant un arc du graphe.
— Le premier joueur qui ne peut pas déplacer le pion a perdu.

On cherche à savoir s'il existe une stratégie gagnante pour l'un des joueurs, c'est à dire s'il existe une méthode qui permet de le faire gagner quel que soit les coups réalisé par l'adversaire.

Ce jeu simple permet en fait de modéliser toute une classe de jeux : les *jeux combinatoires à deux joueurs et à information complète*. Par combinatoire on entend de réflexion, c'est-à-dire que ce n'est pas un jeu d'habileté (type fléchettes) et sans hasard (ce qui exclut

quasiment tous les jeux de cartes ou de dés). A deux joueurs signifie que les deux joueurs jouent à tour de rôle (ce qui exclut des jeux type pierre-feuille-ciseau). A information complète signifie que à tous moments les joueurs ont accès à l'état exact du jeu, il n'y a pas d'éléments cachés.

Cette classe de jeux comprend par exemple les échecs, les dames, le jeu de go, othello, puissance 4, morpion, tic-tac-toe, jeu de petits carreaux...

III.2.2 Modélisation

Etant donné un jeu combinatoire à information parfaite à deux joueurs, on lui associe un graphe orienté de la façon suivante (on laisse de côté la possibilité de parties nulles) :

— l'ensemble des sommets est l'ensemble des états possibles du jeu,

— deux sommets sont reliés par un arc s'il existe un coup amenant de la première position à la deuxième.

Il existe des parties qui ne se termine jamais si et seulement si le graphe admet un cycle. C'est pour cela que certain jeux comme le go ou les échecs interdisent de se retrouver plusieurs fois dans la même situation.

Si le jeu admet des parties nulles, on peut modifier le problème en considérant que le joueur ne doit pas perdre.

Remarque III.2. Si on joue à qui perd gagne à un jeu combinatoire à information parfaite à deux joueurs, alors on peut se ramener à un jeu combinatoire à information parfaite.

III.2.3 Noyau d'un graphe

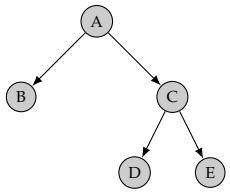
On cherche un ensemble N tel que quel que soit le coup de l'adversaire, on peut toujours se ramener à un sommet de N.

Définition III.3. Soit $G = (S, A)$ un graphe orienté, on dit que $N \subset S$ est un noyau de G s'il vérifie :

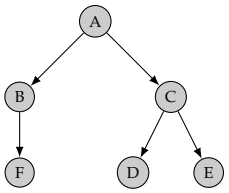
— pour tout $s \in N$ les successeurs de s ne sont pas dans N (on dit que N est *stable*),

— pour tout $s \in S \setminus N$ alors s admet un successeur dans N (on dit que N est *absorbant*).

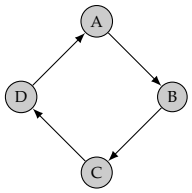
Exemple III.2. On a les exemples suivants de noyaux :



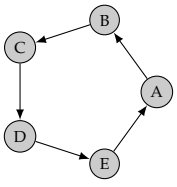
Noyau : $\{B, D, E\}$



Noyau : $\{A, D, E, F\}$



Noyau : $\{A, C\}$
et aussi $\{B, D\}$



Pas de noyau

Un graphe ne possède pas nécessairement de noyau. En général c'est un problème difficile (NP-complet) de décider si un graphe donné admet un noyau. Par contre dans le cas des graphes sans circuits, on a le résultat suivant :

Théorème III.9

Tout joueur dont la position initiale n'est pas dans le noyau a une stratégie non perdante.

Démonstration : On montre qu'un joueur qui peut choisir s dans le noyau ne peut pas perdre. Si s n'a pas de successeur, l'adversaire ne peut plus jouer, il a perdu. Sinon, l'adversaire va choisir un sommet s' dans les successeurs de s. On a $s' \in S \setminus N$ donc s' admet au moins un successeur dans N. ■

Théorème III.10

Un graphe orienté sans circuit possède un unique noyau.

Démonstration : On remarque que tout graphe sans circuit admet un puits et que tous les puits doivent appartenir au noyau.

On va raisonner par récurrence sur le nombre n de sommets.

Initialisation : Si $n = 1$, l'unique sommet est un puits et donc le seul élément du noyau.

Induction : Soit s un puits du graphe G sans circuit. Notons $P(s)$ l'ensemble des prédecesseurs de s. Par hypothèse de récurrence, le graphe G privé du sommet s et de ceux de $P(s)$ a un noyau unique N. On en déduit que $N \cup \{s\}$ est l'unique noyau de G. ■

Remarque III.3. Il est facile d'adapter l'énoncé pour prendre en compte les nulles : un des deux joueurs a une stratégie non-perdante.

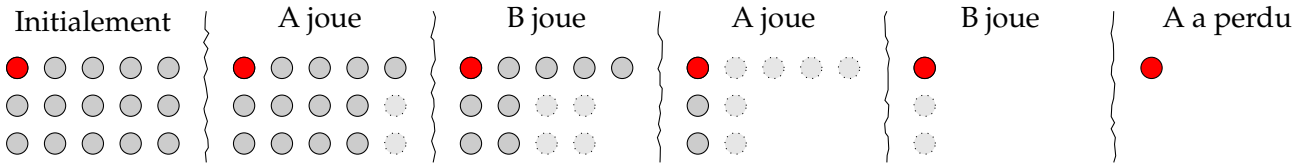
Remarque III.4. Le graphe d'un jeu est en général tellement énorme qu'il est impossible de déterminer une stratégie gagnante (et heureusement). Un jeu est résolu quand une stratégie gagnante a été déterminée (exemple de jeu résolu : puissance 4).

III.2.4 Exemples de jeux

Chomp

Principe du jeu Chomp est joué avec une "tablette de chocolat", c'est-à-dire un rectangle composé de blocs carrés. Les joueurs choisissent un carré à tour de rôle, et le "mange", ainsi que tous les carrés situés à sa droite ou plus bas. Le carré en haut à gauche est empoisonné et celui qui le mange perd la partie.

Voici un exemple de partie à partir d'une tablette de taille 3x5 :



Stratégie gagnante Comme le graphe associé est sans cycle, on sait qu'un des deux joueurs a une stratégie gagnante. Par un argument de "vol de stratégie", on peut montrer que le joueur 1 a une stratégie gagnante. En effet, supposons que le joueur 2 possède une stratégie gagnante contre tous les premiers coups possibles du premier joueur. Supposons ensuite que le joueur 1 effectue son premier coup en mangeant le carré en bas à droite. Le joueur 2 répond avec sa stratégie gagnante en mangeant un certain carré (n, m). Mais dans ce cas, le joueur 1 aurait pu lui-même jouer le coup (n, m) dès le début, et appliquer ensuite lui-même la stratégie gagnante. Ceci prouve que le deuxième joueur ne peut pas posséder de stratégie gagnante. On parle de preuve par vol de stratégie parce que le deuxième joueur se fait voler toute stratégie potentielle possible par le premier.

Cependant à part une exploration informatique, on ne connait pas la stratégie gagnante.

Jeux de Nim

Les jeux de Nim sont des jeux très courants. Chaque jeu se joue à deux au tour par tour. Il s'agit en général de déplacer ou de prendre des objets selon des règles qui indiquent comment passer d'une position du jeu à une autre, en empêchant la répétition cyclique des mêmes positions. Le nombre de positions est fini et la partie se termine nécessairement, le joueur ne pouvant plus jouer étant le perdant (ou selon certaines variantes, le gagnant).

Le jeu de Nim trivial ou (jeu de Nim à un seul tas) Ce est constitué d'un seul tas de n allumettes, chaque joueur prenant le nombre d'allumettes qu'il veut. Celui qui ne peut plus prendre à perdu. La stratégie gagnante consiste évidemment à prendre toutes les allumettes.

Le graphe associé est $S = \{0, \dots, n\}$ et $A = \{(x, y) \in S^2 : y < x\}$ le noyau est réduit à $\{0\}$.

Le jeu de Nim un peu moins trivial (Fort Boyaux) Il consiste à prendre entre 1 et m allumette dans un tas de n allumettes. Celui qui ne peut plus prendre d'allumette a perdu. Le deuxième joueur gagne si et seulement si $m + 1$ divise n.

Le graphe associé est $S = \{0, \dots, n\}$ et $A = \{(x, y) \in S^2 : x - m \leq y < x\}$ le noyau est réduit à $(m + 1)\mathbb{N} \cap S$. On en déduit que le joueur 1 a une stratégie gagnante si $n \notin (m + 1)\mathbb{N}$.

Le jeu de Nim un peu moins trivial inversé C'est le ''qui perd gagne" du jeu précédent. Ainsi celui qui prend la dernière allumette a perdu.

Le graphe associé est $S = \{1, \dots, n\}$ et $A = \{(x, y) \in S^2 : x - m \leq y < x\}$ le noyau est réduit à $(m + 1)\mathbb{N} + 1 \cap S$. On en déduit que le joueur 1 a une stratégie gagnante si $n - 1 \notin (m + 1)\mathbb{N}$.

Jeu de Nim classique ou jeu de Marienbad C'est les mêmes règles que précédemment mais avec plusieurs tas et à chaque coup, on ne peut prendre des allumette que dans un seul tas.

Jeu de Grundy Le jeu de Grundy se joue en séparant l'un des tas en deux tas de taille distincte, jusqu'à ce qu'il ne reste que des tas à un objet.

Jeu de Wythoff Le jeu de Wythoff se joue à deux tas. Chaque joueur réduit d'un même nombre d'objets les deux tas à la fois, ou bien réduit un seul tas du nombre d'objets qu'il veut.

Sprouts

Principe du jeu Sprouts (germe en anglais) se joue à deux joueurs avec un stylo et une feuille de papier. Au départ, il y a n points sur la feuille. Chaque joueur, à tour de rôle, relie deux points existants par une ligne et ajoute un nouveau point sur cette ligne de telle sorte que :

— les lignes ne peuvent se croiser (le graphe doit rester planaire),
— un point ne peut pas être relié à plus de trois lignes (le degré maximal des sommets est 3).

Celui qui ne peut plus jouer sans enfreindre les deux contraintes a perdu. Il existe également une version misère, où celui qui ne peut plus jouer est cette fois le gagnant.

Le nombre de points tracés sur la feuille augmente à chaque coup, on peut donc se demander si la partie se termine en un nombre fini de coups.

Proposition III.11

Toute partie de Sprout à partir de n sommets se termine en au plus $3n-1$ coups.

Démonstration : On appelle liberté d'un sommet s le nombre $3 - d(s)$. Etant donné une configuration de Sprout, lorsqu'on on relie deux sommet on perd deux libertés correspondant aux sommets reliés et on rajoute une liberté correspondant au nouveau sommet. Ainsi, après avoir joué le nombre total de liberté a baissé de un. Le jeu s'arrête nécessairement s'il reste une seule liberté.

Ainsi le nombre de coup correspond au plus au nombre de liberté initiale moins 1, c'est à dire $3n-1$. ■

On peut aussi contrôler la durée d'une partie et montrer qu'une partie se termine au minimum en $2n$ coups.

Stratégie gagnante Si $n = 1$, le joueur 1 est certain de perdre. En effet, il ne peut que faire une boucle sue le sommet et le joueur 2 relie les deux sommets.

En partant de deux points, $n = 2$, l'analyse du jeu est déjà moins évidente, mais on peut établir la liste de toutes les configurations et le joueur qui commence perdra toujours si son adversaire joue convenablement.

On a établit des stratégie gagnante informatiquement pour des valeurs de n inférieure à 50, la conjecture actuelle étant qu'avec n points au départ, le jeu sprout est gagnant pour le second joueur si $n = 0$, 1 ou 2 modulo 6 et il est gagnant pour le joueur 1 dans les autres cas.

III.3 Parcours dans un graphe

III.3.1 Notion générale

Un parcours de graphe est un algorithme consistant à explorer les sommets de proche en proche à partir d'un sommet initial. Dans cette section on considèrera que les graphes traités sont orientés. Les algorithmes fonctionnent pour le cas non-orienté en transformant chaque arête en deux arcs à double sens.

Soit $G = (S, A)$ un graphe et $s \in S$ un sommet, un parcours du graphe G à partir de s est une visite de chaque sommet accessible depuis s. Un parcours peut être représenté par un sous-graphe de G qui est un arbre de racine s. Lors d'un parcours de graphe, on doit marquer les sommets visités pour ne pas les traiter plusieurs fois. Lorsqu'on marque un sommet on réalise le traitement de ce sommet, le moment où l'on réalise ce marquage peut donner des parcours différents.

D'un point de vue algorithmique, un parcours correspond à la procédure suivante.

Il reste à préciser dans quel ordre on prend les sommets de L. On définira alors le parcours en largeur et le parcours en profondeur.

Algorithm 2: Algorithme de parcours

Data: Un graphe orienté $G = (S, A)$ et un sommet s

$L \leftarrow$ Liste des sommets à traiter (vide au départ);
Mettre s dans L (Début traitement de s);
while $L \neq \emptyset$ **do**
  sortir x le premier sommet de L (x en cours de traitement);
  for *y voisin non marqué de x* **do**
    $P(y) \leftarrow x$;
    Mettre y dans L (Début traitement de y);
  Fin du traitement de x;

III.3.2 Parcours en largeur

A partir d'un sommet s, un *parcours en largeur* traite d'abord les voisins de s pour ensuite les explorer un par un. Ce mode de fonctionnement utilise donc une file dans laquelle on ajoute les voisins non encore explorés par le bas (enfiler) et on retire les sommets à traiter par le haut (défiler).

Si on veut récupérer la liste des prédécesseurs P qui permet de retrouver l'arbre de parcours en largeur depuis le sommet s on utilise l'algorithme suivant :

Algorithm 3: Algorithme de parcours en largeur

Data: Un graphe orienté $G = (S, A)$ et un sommet s

$L =$ File des sommets à traiter (vide au départ);
$P =$ Liste de taille $|S|$ où toutes les valeurs sont affectées de $\emptyset$ (liste des prédécesseurs dans l'arbre de parcours);
Marquer le sommet s et l'enfiler dans L;
while $L \neq \emptyset$ **do**
  défiler x le premier sommet de L;
  for *y voisin non marqué de x* **do**
    Marquer y;
    $P(y) \leftarrow x$;
    enfiler y dans L;

Remarque III.5. La liste L des sommets à traiter est l'exemple type d'une Pile de type FIFO (First In, First Out) :

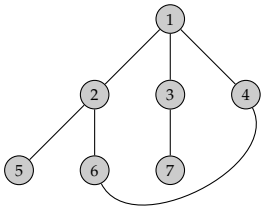
— on ajoute les éléments "par le bas" de la file,
— on retire les éléments "par le haut" de la file.

Le parcours en largeur explore tous les sommets accessibles depuis le sommet initial. Il permet de calculer les composantes connexes du graphe avec une complexité linéaire.

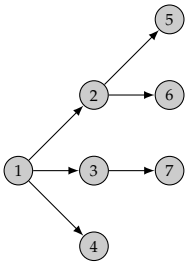
De plus, lors de ce parcours, les sommets sont explorés par distance croissante au sommet de départ. Grâce à cette propriété, on peut utiliser l'algorithme pour résoudre

le plus simple des problèmes de cheminement : calculer le plus court chemin entre deux sommets.

Exemple III.3. On réalise un parcours en largeur de G en commençant par le sommet 1 et en choisissant les sommet voisins dans l'ordre croissant.



G



Arbre de parcours $P = (\emptyset, 1, 1, 1, 2, 2, 3)$

III.3.3 Parcours en profondeur

C'est un algorithme de recherche qui progresse à partir d'un sommet s en s'appelant récursivement pour chaque sommet voisin de s. Le nom d'algorithme en profondeur est dû au fait que, contrairement à l'algorithme de parcours en largeur, il explore en fait "à fond" les chemins un par un : pour chaque sommet, il marque le sommet actuel, et il prend le premier sommet voisin jusqu'à ce qu'un sommet n'ait plus de voisins (ou que tous ses voisins soient marqués), et revient alors au sommet père.

Comme pour le parcours en largeur on peut écrire l'algorithme permettant de récupérer l'arbre de parcours en profondeur :

Algorithm 4: Algorithme de parcours en profondeur

Data: Un graphe orienté $G = (S, A)$ et un sommet s

$L =$ Pile des sommets à traiter (vide au départ);
$P =$ Liste de taille $|S|$ où toutes les valeurs sont affectées à $\emptyset$ (liste des prédécesseurs dans l'arbre de parcours);
Enfiler s dans L;
while $L \neq \emptyset$ **do**
    dépiler x le premier sommet de L;
    if *x non marqué* **then**
        for *y voisin de x non marqué* **do**
            $P(y) \leftarrow x$;
            Mettre y au début de L;
        Marquer x;

Cet algorithme admet aussi une formulation récursive plus simple à programmer :

Remarque III.6. La liste L des sommets à traiter est l'exemple type d'une Pile de type LIFO (Last In, First Out) :

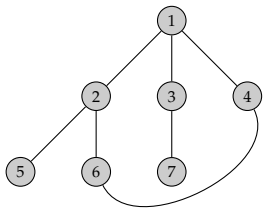
— on ajoute les éléments "par le haut" de la pile,
— on retire les éléments "par le haut" de la pile.

Algorithm 5: Algorithme de parcours en profondeur récursif

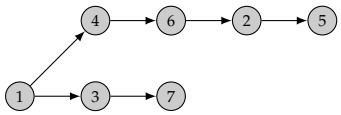
Data: Un graphe orienté sans circuit $G = (S, A)$ et un sommet s
Data: $P = DFS(G, s)$

Marquer s;
for *x voisin non marqué de s* **do**
  $P(y) \leftarrow x$;
  $ParcoursProfondeur(G, x)$;

Exemple III.4. On réalise un parcours en profondeur de G en commençant par le sommet 1 et en choisissant les sommet voisins dans l'ordre croissant.



G



Arbre de parcours $P = (\emptyset, 6, 1, 1, 2, 4, 3)$

Chapitre IV

Problèmes de coloriages

IV.1 Coloriage de sommets

IV.1.1 Position du problème

Définition IV.1. Soit $G = (S, A)$ un graphe non orienté simple (sans boucle et pas multi-arêtes). Un *coloriage* de G consiste à assigner une couleur (ou un nombre) à chaque sommet de telle sorte que deux sommets adjacents soient de couleurs différentes. Un graphe G est *k-coloriable* s'il existe un coloriage avec k couleurs.

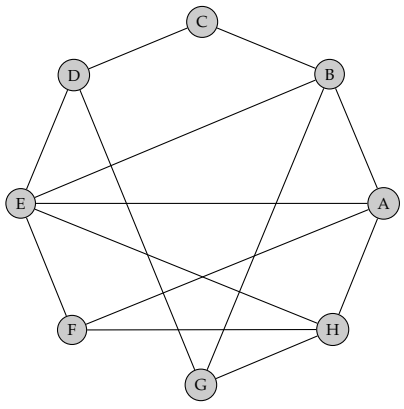
Le *nombre chromatique* du graphe G, noté $\chi(G)$ est le nombre minimal de couleurs nécessaire pour colorier un graphe.

IV.1.2 Exemples d'applications

Problème de compatibilité Dans un groupe de 14 étudiants, on doit former des groupes de telle sorte que les étudiants d'un même groupe ne s'entendent pas trop mal. On connaît les incompatibilités suivantes :

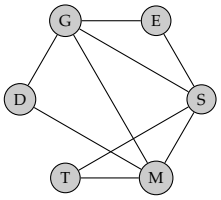
l'étudiant	A	B	C	D	E	F	G	H
ne s'entend pas avec	B,E,F,H	A,C,E,G	B,D	C,E,G	A,D,F,H	A,E,H	B,D,H	A,E,F,G

Le nombre minimal de groupes nécessaire correspond au nombre chromatique du graphe des incompatibilités.



Problème d'emploi du temps Pendant un festival, on veut organiser des tournois de scrable (S), échecs (E), go (G), dames (D), tarot (T) et master-mind (M). Plusieurs personnes se sont inscrites à la fois pour les tournois E, S, G, d'autres personnes pour les tournois G, D, M, et enfin d'autres personnes pour les tournois M, T, S. Il est entendu qu'une participation simultanée à plusieurs tournois est impossible et que les organisateurs veulent satisfaire tout le monde.

Quel est le nombre maximum de tournois qui pourraient se dérouler en même temps ?



Coloriage de carte On cherche à colorier une carte de telle sorte que deux pays frontaliers soient de couleurs différentes. Pour résoudre ce problème, plus historique qu'autre chose, on peut se ramener au coloriage d'un graphe planaire construit de la façon suivante : les sommet correspondent aux pays et il y a une arête entre deux sommets si les pays correspondant sont frontaliers.

IV.1.3 Nombre chromatique de graphes classiques

Il est facile de déterminer le nombre chromatique de certains graphes classiques :
— graphe isolé d'ordre n : $\chi(I_n) = 1$;
— graphe cyclique d'ordre n : $\chi(C_n) = 2$ si n pair et 3 si n impair ;
— graphe complet d'ordre n : $\chi(K_n) = n$;
— G graphe biparti avec au moins une arête : $\chi(G) = 2$ (en fait un graphe est 2-coloriable si et seulement s'il est biparti) ;
— G arbre avec au moins une arête : $\chi(G) = 2$.

IV.1.4 Comment calculer un nombre chromatique ?

Il est intéressant d'avoir des outils pour encadrer le nombre chromatique. On note qu'obtenir un coloriage à k couleurs d'un graphe G permet d'affirmer que $\chi(G) \leq k$. La difficulté réside pour trouver une minoration.

Proposition IV.1

Soit G un graphe et G' un sous graphe, on a $\chi(G') \leq \chi(G)$.

On va introduire deux nouvelles notions.

Définition IV.2. Soit G un graphe non orienté.
Une *clique* est un sous-graphe complet de G.
Une *stable* est un sous-graphe induit de G sans arcs (ou arêtes).

Ces notions donnes des informations sur le nombre chromatique :
— les sommets d'une même clique doivent être coloriés d'une couleur différente, ainsi trouver une clique à k sommets permet d'affirmer que $\chi(G) \geq k$;
— les sommets d'une même stable peuvent être coloriés de la même couleur.

IV.2 Résolution algorithmique

Dans cette section on s'intéresse aux algorithmes qui permettent de trouver un coloriage ou le nombre chromatique.

IV.2.1 Algorithme glouton

On considère ici un coloriage comme une fonction des sommets dans les entiers. L'algorithme glouton nous donne facilement un coloriage du graphe, le principe consiste à prendre les sommets les uns après les autres et pour chaque sommet s d'affecter la couleur minimale qui n'apparait pas dans les voisins coloriés de s.

Algorithm 6: Algorithme glouton de coloriage d'un graphe

Data: Un graphe $G = (S, A)$
Result: Une coloration $\varphi : S \to \mathbb{N}^*$ de G

for $s \in S$ **do**
  $\varphi(s) \leftarrow$ plus petite couleur non utilisé par les voisins de s;

Terminaison L'algorithme termine une fois que l'on a visité tous les sommets.

Correction A chaque fois que l'on attribue une couleur à un sommet, elle est différentes des couleurs des sommets voisins pour lesquels on a attribué une couleur. Ainsi le coloriage obtenu est valide.

Complexité On passe $|S|$ fois dans la boucle, chaque fois que l'on passe dans la boucle on regarde tous les voisins du sommet considéré, on a au plus $\Delta(G)$ voisin à regarder où $\Delta(G)$ est le degré maximal du graphe. Dans le pire des cas, on une complexité $O(\Delta(G)|S|)$.

A t'on un coloriage optimal avec cet algorithme ? Le résultat dépend généralement de l'ordre dans lequel on choisit les sommets et il est facile de trouver des exemples ou l'ordre donné ne donne pas un coloriage optimal. On peut jouer sur l'ordre des sommets choisis, par exemple les prendre dans l'ordre des degrés décroissants.

IV.2.2 Algorithme de Welsh-Powell

Il est possible d'améliorer cet algorithme en coloriant d'abord les sommets qui imposent le plus de contraintes (sommet de plus haut degré) et en utilisant la couleur que l'on vient d'utiliser là ou cela est possible. On appèle ce principe l'algorithme de Welsh-Powell. Pour certaine classe de graphe cet algorithme donne même systématiquement le

coloriage optimal.

Algorithm 7: Algorithme de Welsh-Powell pour colorier un graphe

Data: Un graphe $G = (S, A)$
Result: Une coloration $\varphi : S \to \mathbb{N}$ de G

$L \leftarrow$ liste des sommets ordonnés par degré décroissant ;
couleur-courante $\leftarrow 0$;
while $L \neq \emptyset$ **do**
  couleur-courante $\leftarrow$ couleur-courante $+1$;
  Colorier s le premier sommet de L avec couleur-courante;
  Eliminer s de L;
  $V \leftarrow$ voisins de s;
  for $x \in L$ **do**
    if $x \notin V$ **then**
      Colorier x avec la couleur-courante;
      Eliminer x de L;
      Ajouter les voisins de x à V;

Terminaison Il est clair que, puisque le nombre de sommets dans L (et donc non coloriés) diminue d'au moins une unité à chaque fois que l'on exécute la boucle.

Correction Cette algorithme fournit bien un coloriage de G, en effet chaque fois que l'on colorie un sommet, on place dans V les sommets voisins à ce sommet de telle sorte que l'on ne colorie plus de cette couleur les sommets de V. Ainsi deux sommets voisins sont de couleurs différentes.

Complexité De manière grossière, on passe $|S|$ fois dans la boucle **while** puis $|S|$ fois dans la boucle **for**, on a donc une complexité grossière en $O(|S|^2)$. Cependant, on peut être plus précis. Dans la preuve de la proposition IV.2, on voit que l'on passe au maximum $\Delta(G) + 1$ fois dans la boucle **while**. On a donc une complexité en $O(\Delta(G)|S|)$.

Proposition IV.2

Soit $\Delta(G)$ le degré maximal d'un graphe G, on a $\chi(G) \leq \Delta(G) + 1$.

Démonstration : Soit s le dernier sommet colorié par l'algorithme 6. Si s n'a pas été colorié avant, c'est que pour chacune des couleurs précédentes, un sommet adjacent à s a été colorié de cette couleur. Par suite, le nombre de couleurs utilisées avant de colorier s ne peut dépasser $d(s)$. Ainsi, en tenant compte de la couleur de s, on déduit que le nombre total de couleurs utilisées par l'algorithme ne dépasse pas $d(s) + 1$. ■

A t'on un coloriage optimal avec cet algorithme ? Là encore il existe des exemples ou cet algorithme n'est pas optimal même si dans la majorité des cas il donne un coloriage optimal.

IV.2.3 Existe t'il un algorithme pour trouver le nombre chromatique d'un graphe ?

On cherche un algorithme qui prend en argument un graphe $G = (S, A)$ et renvoie le nombre chromatique de ce graphe. Pour cela on teste tous les 2-coloriages, il y en a $2^{|S|}$ s'il y a en a un valide, on a $\chi(G) = 2$, sinon on teste tous les 3-coloriages et ainsi de suite. L'algorithme termine car il y a un coloriage à $\Delta(G) + 1$ couleurs et il nous donne un coloriage optimal car on a essayer toutes les possibilités avec moins de couleurs.

Cependant cet algorithme a une complexité en $O((\Delta(G) + 1)^{|S|})$ dans le pire des cas, cette complexité est par exemple atteinte pour le graphe complet. Cette complexité est exponentielle en la taille du graphe et en pratique, pour des graphes un peu grand, il faut attendre des temps extrêmement long pour le voir terminer. On estime que les complexités qui permettent d'avoir un algorithme utilisable sont les complexité en $O(n^d)$ pour une valeur d donnée. Pour le problème du nombre chromatique on ne sait pas s'il existe un algorithme polynomial qui permet de le résoudre.

Toutefois, il existe des classes de graphes pour lesquelles l'algorithme glouton (et donc de complexité polynomiale) donne même systématiquement le coloriage optimal. En TD on verra qu'un algorithme glouton avec un bon ordre sur les sommets donne un coloriage optimal pour les graphes d'intervalles.

Remarque IV.1. En général on s'intéresse aux problèmes de décisions, par exemple :

Problème 1 : Etant donné $a, b, c \in \mathbb{Z}$, est ce que $ax^2 + bx + c = 0$ admet une solution rélle ?

Problème 2 : Etant donné un graphe G est ce que G admet un 3-coloriage ?

On s'intéresse aux complexités qui résolvent ces problèmes, on définit les classes de problèmes suivant :

— Classe $\mathcal{P}$: classe de problèmes que l'on peut résoudre en temps polynomial (par exemple Problème 1) ;

— Classe $\mathcal{NP}$: classe de problèmes tel que si on donne une solution on peut vérifier que c'est bien une solution du problème (par exemple Problème 2) ;

— Classe $\mathcal{E}xp$: classe de problèmes que l'on peut résoudre en temps exponentiel.

On a $\mathcal{P} \subset \mathcal{NP} \subset \mathcal{E}xp$. On sait que $\mathcal{P} \neq \mathcal{E}xp$ mais on ne sait pas si $\mathcal{P} = \mathcal{NP}$, c'est **le** problème ouvert de l'informatique théorique.

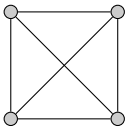
Il existe une autre classe, la classe des problèmes $\mathcal{NP}$-complet, ce sont les problèmes tels que si on les résout en temps polynomial, on résout tous les problèmes $\mathcal{NP}$ en temps polynomial. En particulier le problème de 3-coloriage est $\mathcal{NP}$-complet.

IV.3 Cas des graphes planaires

Les graphes planaires sont une classe graphe avec des propriétés intéressantes du point de vu du coloriage.

Définition IV.3 (Graphe planaire)**.** Un graphe $G = (S, A)$ est *planaire* s'il existe une représentation dans le plan où les arêtes ne s'intersectent pas.

Exemple IV.1. Le graphe suivant est planaire si on déplace les sommets



G_3

Etant donné une représentation planaire d'un graphe, les arêtes délimitent des régions que l'on appellera *face*. La formule d'Euler, montré en 1758, relie le nombre de sommets, d'arêtes et de face.

Théorème IV.3 Formule d'Euler

Soit G un graphe planaire connexe dont une représentation planaire possède s sommets, a arêtes et f faces. On a

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

Si G possède k composantes connexes, on a alors

$$s - a + f = 1 + k$$

Démonstration : Soit G un graphe planaire connexe. D'après la propriété III.5, il suffit de prouver que la formule est vraie pour les arbres et que la quantité s-a+f reste invariante par ajout d'une arête en restant planaire. On ramarque que si G est planaire tout graphe qui permet de construire G par ajout d'arête est lui-même planaire.

Pour un arbre, s'il y a s sommets alors il y a $a = s - 1$ arêtes. De plus, la seule face est la face non bornée, puisque toute face bornée ferait apparaître un cycle. Donc f = 1 et on a $s - a + f = s - (s - 1) + 1 = 2$.

Si la formule est vraie pour un graphe connexe planaire G' qui admet s' sommets, a' arêtes et f' faces, et que l'on ajoute une arête sans briser la planarité. Le nouveau graphe possède $s = s'$ sommets, $a = a' + 1$ arêtes. De plus, la nouvelle arête partage une face en deux nouvelles faces (puisqu'elle ne traverse aucune autre arête, elle est entièrement contenue dans une des anciennes faces et comme G' est connexe, aucune extrémité de cette arête n'est isolée). Le nouveau graphe a donc $f = f' + 1$ faces. Ainsi $s - a + f = s' - (a' + 1) + (f' + 1) = 2$.

Supposons maintenant que G ait k composantes connexes. Puisqu'elles sont deux à deux disjointes, on peut les représenter de sorte que chacune appartienne à la face non bornée de chacune des autres. On peut utiliser la formule précédente pour chacune des composantes connexes. En sommant toutes ces relations, la somme des nombres de sommets (resp. d'arêtes) donne exactement le nombre total de sommets (resp. d'arêtes) de G, et la somme des faces donne exactement le nombre total de faces de G augmenté de $k - 1$ unités puisque la face non bornée a été comptée k fois en tout. On obtient donc $s - a + f + (k - 1) = 2k$, c'est à dire $s - a + f = 1 + k$. ■

En TD on utilisera ce résultat pour montrer que $K_{3,3}$ et K_5 ne sont pas planaire. On peut aussi facilement montrer qu'un graphe planaire est 5-coloriable.

En fait, le nombre de couleur maximal pour colorié un graphe planaire est 4. Ce théorème est connu comme l'un des premiers ou la preuve nécessite un ordinateur pour explorer l'explosion combinatoire des différents cas de base.

Théorème IV.4

Tout graphe planaire est coloriable avec 4 couleurs, son nombre chromatique est donc inférieur ou égal à 4.

Chapitre V

Problèmes d'optimisation pour des graphes valués

Dans cette section on considère un graphe $G = (S, A)$ orienté ou non pour lequel chaque arête est attribué d'un certain poids $\lambda : A \to \mathbb{R}$ appelé *valuation*. Etant donné un sous-graphe $G' = (S', A')$ ($S' \subset S$ et $A' \subset A$) on définit le poids du graphe G'

$$\lambda(G') = \sum_{a \in A'} \lambda(a).$$

Un graphe simple valué est donné par la matrice de poids $W = (w_{i,j})_{(i,j)\in S^2}$ tel que $w_{i,j}$ est la valeur de l'arc allant de i à j.

On s'intéresse à différents problème d'optimisation qui consiste à chercher un sous graphe avec une certaine propriété qui minimise ou maximise son poids.

V.1 Recherche d'arbre couvrant de poids maximal/minimal

V.1.1 Problème

Quand on travaille sur un graphe connexe, certains problèmes obligent à transformer ce graphe en un arbre (graphe connexe sans cycle) qui contient tous les sommets du graphe et quelques arêtes. Un *arbre couvrant* d'un graphe non orienté $G = (S, A)$ est un sous graphe de G dont les sommet sont S et qui est un arbre. On a vu que tout graphe admet un arbre couvrant (théorème III.5).

Si G admet une valuation $\lambda : A \to \mathbb{R}$, parmi les arbres couvrant, il en existe un de poids minimal (resp. maximal). Les algorithmes de Prim et Kruskal permettent de trouver ces arbres.

Exemple V.1. Un réseau maritime peut être modélisé par un graphe, chercher un arbre couvrant revient à le simplifier au maximum. On peut alors s'intéresser à supprimer les liaisons maritimes les moins rentables en préservant l'accessibilité aux différents ports.

On va considérer deux approches pour résoudre ce problème :

— Approche locale : à chaque étapes, parmi les sommets connectés, on rajoute l'arête optimale qui relie un sommet déjà connecté à un sommet non connecté. On utilisera cette approche dans l'algorithme de Prim.

— Approche globale : on choisit l'arête optimale de façon que l'are rajouté ne relie pas des sommets déjà connectés. On utilisera cette approche dans l'algorithme de Kurskal.

V.1.2 Algorithme de Prim

Description de l'algorithme L'algorithme de Prim consiste à choisir arbitrairement un sommet et à faire croître un arbre à partir de ce sommet de telle sorte que chaque augmentation se fait en prenant l'arête de poids optimal (maximal ou minimal suivant le cas).

Algorithm 8: Algorithme de Prim

Data: Un graphe non orienté $G = (S, A)$ valué par $\lambda : A \to \mathbb{R}$ et un sommet s
Result: Un arbre $T = (S, A')$

Poids$\leftarrow$ 0 (Poids total de l'arbre couvrant);
$A' \leftarrow \emptyset$;
Marquer s;
while *il reste des sommets non marqués* **do**
  $a \leftarrow$ arête de poids minimal joignant un sommet marqué x et un sommet non marqué y;
  Marquer y;
  $A' \leftarrow A' \cup \{a\}$;
  Poids$\leftarrow$Poids$+\lambda(a)$;

Terminaison Chaque fois que l'on passe dans la boucle on marque un sommet, l'algorithme s'arête quand on a marqué tous les sommets. Comme il y a un nombre fini de sommet l'algorithme termine.

Correction L'algorithme de Prim repose sur le résultat suivant :

Proposition V.1

Si on cherche un arbre à coût minimal contenant un sous-arbre G' imposé, alors il existe parmi les solutions optimales contenant G', une solution qui contient l'arête (ou une des arêtes) de coût minimal adjacente à G' et ne formant pas de cycle avec G' (c'est-à-dire une extrémité dans A et l'autre à l'extérieur de G').

Démonstration : On va faire un raisonnement par l'absurde. Considérons un graphe $G = (S, A)$ et considérons un sous-graphe $G_1 = (S_1, A_1)$ qui soit un arbre. Soit $G_2 = (S, A_2)$ un arbre de recouvrement de G qui contient l'arbre G_1, mais qui ne contient aucune des arêtes de coût minimal dont une de ses extrémités est dans S_1 et l'autre extrémité dans $S \setminus S_1$.

Soit a une des arêtes de coût minimal entre $S \setminus S_1$ et S_1. Si on l'ajoute à l'arbre G_2, on crée obligatoirement un cycle. Ce cycle contient exactement deux arêtes ayant une extrémité dans $S \setminus S_1$ et une extrémité dans S_1, a et une autre arête b. Si on enlève b, on casse ce cycle. On a donc un graphe sans cycle de $n - 1$ arêtes, c'est donc un arbre et comme le coût de a est strictement plus petit que celui de b, on a construit un arbre de recouvrement de poids strictement inférieur à celui de l'arbre minimal considéré qui n'était donc pas minimal. ■

Complexité On passe $|S|$ fois dans la boucle, et chaque fois que l'on passe dans la boucle, on teste au plus $|A|$ arêtes. La complexité est donc $O(|A||S|)$.

Cette complexité est obtenue avec une représentation des graphes naïve, comme une simple liste d'adjacence et des recherches dans celle-ci. Cependant, la complexité de l'algorithme dépend fortement de la manière dont est implémenté le choix de l'arête/sommet à ajouter dans l'ensemble à chaque étape. Si l'on utilise un tas min binaire, la complexité devient alors $O(|A| \log |S|)$. En utilisant un tas de Fibonacci, on peut encore descendre à $O(|A| + |S| \log |S|)$.

V.1.3 Algorithme de Kruskal

Description de l'algorithme L'algorithme consiste à ranger par ordre de poids croissant ou décroissant les arêtes d'un graphe, puis à retirer une à une les arêtes selon cet ordre et à les ajouter à l'arbre couvrant cherché tant que cet ajout ne fait pas apparaître un cycle dans l'arbre couvrant.

Algorithm 9: Algorithme de Kruskal

Data: $G = (S, A, \lambda)$ graphe valué
Result: Un arbre $T = (S, A')$

Poids$\leftarrow$ 0 (Poids total de l'arbre couvrant);
$A' \leftarrow \emptyset$;
for $s \in S$ **do**
  $E(s) \leftarrow \{s\}$ ($E(s)$: sommets reliés à s)
for $\{s, s'\} \in A$ *dans l'ordre croissant* **do**
  if $E(s) \neq E(s')$ **then**
    $A' \leftarrow A' \cup \{s, s'\}$ Poids$\leftarrow$Poids$+\lambda(\{s, s'\})$;
    $F \leftarrow E(s) \cup E(s')$;
    for $z \in F$ **do**
      $E(z) \leftarrow F$;

Terminaison Chaque fois que l'on passe dans la boucle on prend une nouvelle arête. Comme il y a un nombre fini d'arêtes l'algorithme termine.

Résultat sur lequel repose l'algorithme L'algorithme de Kruskal repose sur la proposition suivante :

Proposition V.2

Parmi les arbres de recouvrement minimaux du graphe $G = (S, A)$ pour lesquels le sous-ensemble d'arêtes A' est imposé, il en existe au moins un qui contient une des plus petites arêtes de $A \setminus A'$ qui ne crée pas de cycle lorsqu'on l'ajoute à A'.

Démonstration : Le théorème est évidemment vrai lorsque A' est vide.

Supposons maintenant que A' soit non vide et contienne moins de $n - 1$ arêtes. On suppose que toutes des arêtes de coût minimal qui ne sont pas déjà dans A' tout en ne créant pas de cycles avec A' ne soit pas retenue.

Considérons un arbre de coût minimal T qui contient A' et pas les arêtes refusées précédemment. Soit a une des arêtes refusées précédemment. Si on l'ajoute à T, on crée un cycle. Comme a ne créait pas de cycle en l'ajoutant à A'. Obligatoirement, une des arêtes de ce cycle, adjacente à a, n'est pas dans A' et a donc un coût strictement supérieur à a qui était une des arêtes de coût minimal. Soit b l'une des arêtes adjacente à a et ajoutée à A' après avoir refusé a. En ôtant b de T et en ajoutant a, on supprime le cycle et on garde la connexité, ce qui nous fournit un arbre de coût strictement inférieur à l'arbre supposé de coût minimal. ■

Complexité

V.2 Problème de plus court chemin

V.2.1 Position du problème

Soient $G = (S, A, \lambda)$ un graphe valué et $s, s' \in S$ deux sommets de G. On appelle distance de s à s' et on note $d(s, s')$ le minimum des valuations des chemins (resp. chaînes) allant de s à s'. On recherche plus court chemin (resp. plus courte chaîne) de s à s'.

De nombreux problèmes concrets peuvent se modéliser comme des recherches de plus courts chemins dans des graphes valués. Par exemple :

— recherche de l'itinéraire le plus rapide en voiture entre deux villes, ou en méro entre deux stations ;
— routage dans des réseaux de télécommunications ;
— certains problèmes d'ordonnancement font aussi appel à des recherches de plus longs chemins.

On étudiera des algorithmes qui résolvent le problème suivant : étant donné un sommet s, déterminer pour chaque sommet s' la distance et un plus court chemin de s à s'. Plusieurs cas se présentent :

— il n'y a pas de chemins (chaînes) de s à s' ;
— il existe un ou plusieurs plus courts chemins (chaînes) de s à s' ;
— il existe des chemins (chaînes)de s à s' mais pas de plus court (dans le cas où il y a un cycle négatif).

Remarque V.1. Dans un graphe non orienté, on a toujours $d(s, s') = d(s', s)$, et toute plus courte chaîne de s à s' parcourue à l'envers est une plus courte chaîne de s' à s.

Un circuit *absorbant* est un circuit de valuation négative.

Proposition V.3

Soit G un graphe orienté valué n'ayant pas de circuits absorbants, et s et s' deux sommets de G. S'il existe un chemin allant de s à s', alors la distance $d(s, s')$ est bien définie et il existe au moins un plus court chemin de s à s'.

On définit de la même manière un cycle absorbant dans un graphe non orienté. Le théorème reste vrai en remplaçant chemin par chaîne.

Dans la suite, les graphes seront donc sans circuits absorbants.

Proposition V.4

Tout sous-chemin d'un plus court chemin est un plus court chemin.

Démonstration : Soit $C_0 = (s_0, s_1, s_2, \ldots, s_n)$ un plus court chemin entre x_0 et x_n. Supposons qu'il existe $C = (s_p, s_{p+1}, \ldots, s_{q-1}, s_q)$ un sous-chemin de C_0, avec $0 \leq p \leq q \leq n$ qui n'est pas un plus court chemin entre s_p et s_q. Il existe un autre chemin $C' = (s_p, s'_1, s'_2, \ldots, s'_{r-1}, s'_r, s_q)$ entre s_p et s_q, et dont la longueur est strictement plus petite que celle de C. Or le chemin $C_1 = (s_0, s_1, \ldots, s_{p-1}, s_p, s_p, s'_1, s'_2, \ldots, s'_r, s_q, s_{q+1}, \ldots, s_n)$, obtenu en remplaçant C par C' dans C_0, est alors strictement plus court que C_0, ce qui est absurde. ■

Proposition V.5

S'il existe un plus court chemin entre deux sommets s et s', alors il existe un plus court chemin élémentaire entre s et s'.

Démonstration : Soit $C_0 = (s_0, s_1, s_2, \ldots, s_n)$ un plus court chemin entre x_0 et x_n. Si C_0 n'est pas élémentaire, il existe deux indices p et q, $0 \leq p < q \leq n$, tels que $s_p = s_q$. Le sous-chemin $C1 = (s_p, s_{p+1}, \ldots, s_{q-1}, s_q)$ est alors un circuit, et c'est aussi un plus court chemin d'après la propriété précedente. Il est donc au moins aussi court que le chemin trivial (s_p), de valuation 0. Si la valuation de C_1 est strictement négative, alors C_1 est un circuit absorbant, et il n'existe pas de plus court chemin entre $s = s_0$ et $s' = s_n$, ce qui est absurde. Si la valuation de C_1 est nulle, le chemin $C'_0 = (s_0, s_1, \ldots, s_{p-1}, s_p, s_{q+1}, s_{q+2}, \ldots, s_{n-1}, s_n)$ a la même longueur que C_0, c'est donc encore un plus court chemin. On construit ainsi un plus court chemin élémentaire entre s et s'. ■

Cette proposition implique que étant donné un sommet initial, les chemins optimaux pour aller de ce sommet à tous les autres peut être représenté par un arbre enraciné.

V.2.2 Principe des algorithmes étudiés

Les algorithmes étudiés prennent en entrée un graphe valué et renvoie tous les plus courts chemin allant d'un sommet initial s à tous les autres sommets. On stoque toute l'information dans deux listes de taille $|S|$:

— *Dist* qui à la fin de l'algorithme donne $d(s, x)$ pour tout sommet $x \in S$;
— *Pred* qui à la fin de l'algorithme donne le prédécesseur du sommet $x \in S$ dans l'arbre des plus court chemin.

Les trois algorithmes que nous allons étudier fonctionnent de la façon suivante :

— on initialise les tableaux *Dist* et *Pred*
— on calcule $Dist(s)$ et $Pred(s)$ par approximations successives, ce qui signifie qu'à chaque étape, on essaye d'améliorer les valeurs obtenues précédemment ;
— l'amélioration, au niveau local, se vérifie ainsi : pour un sommet s et un successeur s' de s, on compare la valeur $Dist(s')$ obtenue à l'étape précédente avec la valeur qu'on obtiendrait en passant par s, c'est-à-dire $Dist(s) + W(s, s')$ on a alors deux cas :
  — si cette deuxième valeur est plus petite, alors $Dist(s') \leftarrow Dist(s) + W(s, s')$ et $Pred(s') \leftarrow s$;
  — sinon on ne fait rien.

  Cette technique est appelée technique du relâchement.

V.2.3 Algorithme de Bellman-Ford-Kalaba

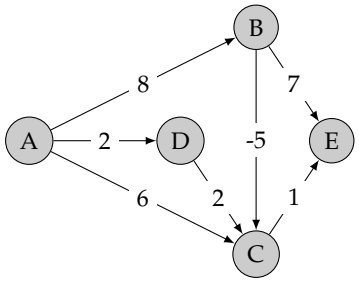
Description de l'algorithme

Exemple V.2. On cherche les distance depuis le sommet A.

Algorithm 10: Algorithme de Bellman-Ford

Data: W matrice des distances d'un graphe orienté $G = (S, A)$ et un sommet s
Result: L'algorithme donne deux tableaux de taille $1 \times |S|$:
- *Dist* : table des distance telle que $Dist(s')$ est la distance de s à s'.
- *Pred* : table des prédécesseurs telle que $Dist(s')$ est le prédécesseur de s' dans le chemin optimal depuis s.

Pred tableau des prédécesseur initialisé à $\emptyset$;
Dist tableau des distances initialisé à $+\infty$;
W matrice des poids des arcs;
$Dist(s) \leftarrow 0$;
$k \leftarrow 1$;
while $k \leq n$ *et il y'a eu des modifications à l'étape précédente* **do**
  for $x \in S$ **do**
    for y *successeur de* x **do**
      if $Dist(x) + W(x, y) < Dist(y)$ **then**
        $Dist(y) \leftarrow Dist(x) + W(x, y)$;
        $Pred(y) \leftarrow x$;
  $k \leftarrow k + 1$;



Itération	sommets traités	$Dist(A)$	$Dist(B)$	$Dist(C)$	$Dist(D)$	$Dist(E)$	$Pred(A)$	$Pred(B)$	$Pred(C)$	$Pred(D)$	$Pred(E)$
Initialisation		0	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
1	A	0	8	6	2	$+\infty$	$\emptyset$	A	A	A	$\emptyset$
1	D	0	8	6	2	$+\infty$	$\emptyset$	A	D	A	$\emptyset$
1	C	0	8	6	2	7	$\emptyset$	A	D	A	C
1	E	0	8	6	2	7	$\emptyset$	A	D	A	C
1	B	0	8	3	2	7	$\emptyset$	A	C	A	C
2	A, D	0	8	3	2	7	$\emptyset$	A	C	A	C
2	C	0	8	3	2	4	$\emptyset$	A	C	A	C
2	E, B	0	8	3	2	4	$\emptyset$	A	C	A	C
3	A, D, C, E, B	0	8	3	2	4	$\emptyset$	A	C	A	C

Terminaison On passe au plus $|S|$ fois dans la boucle While.

Correction On fait d'abord les remarques suivantes :
- les valeurs de $Dist(x)$ ne peuvent que diminuer au cours du déroulement de l'algorithme ;
- à chaque étape de l'algorithme, pour tout sommet x, la valeur $Dist(x)$ est soit $+\infty$, soit égale à la longueur d'un chemin de s à x.
- à chaque étape de l'algorithme $Dist(x) \geq d(s, x)$.

— quand $Dist(x)$ atteins la valeur $d(s,x)$, elle ne varie plus dans la suite de l'algorithme.

Montrons par récurrence la propriété suivante : $\mathbf{P}_k$: si un plus cout chemin élémentaire comporte k arc alors après k passages dans la boucle $Dist(x) = d(s,x)$.

— L'initialisation est claire car seul s peut être atteint avec 0 arcs.
— On suppose la que $\mathbf{P}_k$ est vrai et soit x un sommet tel que le plus court chemin de s à x comporte au plus $k+1$ arcs. Soit p un prédécesseur de x. Le plus court chemin reliant s à p comporte donc k arcs. Après k passages dans la boucle on a $Dist(p) = d(s,p)$ d'après l'hypothèse de récurrence. Après le k-ième passage, on compare $Dist(s)$ et $Dist(p) + W(p,x) = d(s,p) + W(p,x) = d(s,x)$ et on affecte à $Dist(s)$ la valeur $d(s,x)$ si ce n'est pas le cas. Ceci prouve l'hypothèse de récurrence au rang $k+1$.

Complexité $O(|S|^3)$ étapes sont nécessaire dans le pire des cas.

Remarques L'algorithme permet de détecter la présence de circuits absorbants : si les valeurs $d(s)$ ne sont pas stabilisées après $|S|$ passages de boucles, alors le graphe contient au moins un circuit absorbant.

V.2.4 Algorithme de Bellman

Si le graphe n'a pas de circuit, il est possible de renuméroter les sommets de façon à ne jamais revenir en arrière. Pour cela on réalise d'abord un tri topologique :

Algorithm 11: Algorithme de tri

Data: Un graphe orienté $G = (S, A)$ valué par $\lambda : A \to \mathbb{R}$
Result: Une numérotation des sommet $r : S \to \mathbb{N}$

$k \leftarrow 1$;
while $k < n$ **do**
  for $x \in S$ *dont tous les prédécesseurs sont numéroté* **do**
    $r(x) \leftarrow k$;
  $k \leftarrow k + 1$;

L'algorithme de Bellman est un algorithme de type glouton, c'est-à-dire que, contrairement à l'algorithme de Bellman-Ford, il ne revient jamais en arrière. A chaque étape, on trouve un plus court chemin pour un nouveau sommet en se basant sur les prédécesseurs qui sont déjà tous traités. Ceci est possible grâce à la possibilité d'effectuer un tri topologique. On obtient l'algorithme suivant :

Exemple V.3. On cherche les distance depuis le sommet A.

Algorithm 12: Algorithme de Bellman

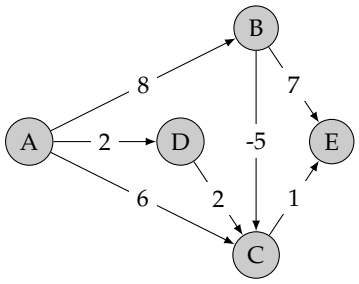
Data: W matrice des distances d'un graphe orienté $G = (S, A)$, sommet s, une fonction niveau $r : S \to \mathbb{N}$ donné par l'algorithme de tri topologique.

Result: L'algorithme donne deux tableaux de taille $1 \times |S|$:

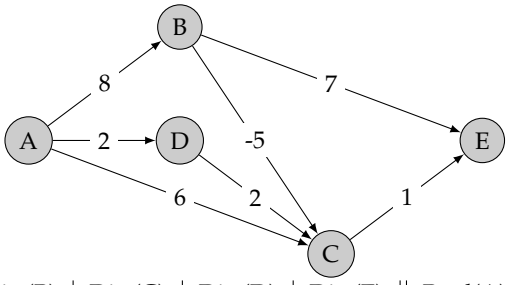
— *Dist* : table des distance telle que $Dist(s')$ est la distance de s à s'.

— *Pred* : table des prédécesseurs telle que $Dist(s')$ est le prédécesseur de s' dans le chemin optimal depuis s.

Pred tableau des prédécesseur initialisé à $\emptyset$;
Dist tableau des distances initialisé à $+\infty$;
W matrice des poids des arcs;
$Dist(s) \leftarrow 0$;
for $k = r(s) + 1$ *jusqu'à* $\max(r)$ **do**
  for $y \in S$ *tel que* $r(y) = k$ **do**
    for x *prédecesseur de* y **do**
      if $Dist(x) + W(x, y) < Dist(y)$ **then**
        $Dist(y) \leftarrow Dist(x) + W(x, y)$;
        $Pred(y) \leftarrow x$;



On commence par ordonner les sommets



Niveau	Dist(A)	Dist(B)	Dist(C)	Dist(D)	Dist(E)	Pred(A)	Pred(B)	Pred(C)	Pred(D)	Pred(E)
Initialisation	0	$+\infty$	$+\infty$	$+\infty$	$+\infty$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
1	0	8	$+\infty$	2	$+\infty$	$\emptyset$	A	$\emptyset$	A	$\emptyset$
2	0	8	3	2	∞	$\emptyset$	A	B	A	$\emptyset$
3	0	8	3	2	4	$\emptyset$	A	B	A	C

V.2.5 Algorithme de Dijkstra-Moore

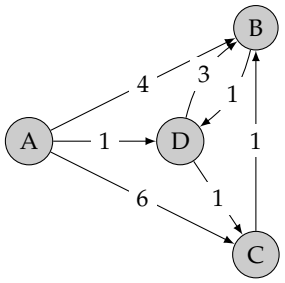
Description de l'algorithme

Algorithm 13: Algorithme de Dijkstra-Moore

Data: W matrice des distances d'un graphe orienté $G = (S, A)$ et un sommet s
Result: L'algorithme donne deux tableaux de taille $1 \times |S|$:
— $Dist$: table des distance telle que $Dist(s')$ est la distance de s à s'.
— $Pred$: table des prédécesseurs telle que $Dist(s')$ est le prédécesseur de s' dans le chemin optimal depuis s.

$Pred$ tableau des prédécesseur initialisé à $\emptyset$;
$Dist$ tableau des distances initialisé à $+\infty$;
W matrice des poids des arcs;
$Dist(s) \leftarrow 0$;
$D \leftarrow \emptyset$ (listes des sommets déjà traités);
while $D \neq S$ **do**
  $x \leftarrow$ sommet de $S \setminus D$ tel que $Dist(x)$ minimal;
  $D \leftarrow \{x\} \cup D$;
  for $y \notin D$ *et* y *successeur de* x **do**
    if $Dist(x) + W(x, y) < Dist(y)$ **then**
      $Dist(y) \leftarrow Dist(x) + W(x, y)$;
      $Pred(y) \leftarrow x$;

Exemple V.4. On cherche les distance depuis le sommet A.



x	D	$Dist(A)$	$Dist(B)$	$Dist(C)$	$Dist(D)$	$Pred(A)$	$Pred(B)$	$Pred(C)$	$Pred(D)$
	$\emptyset$	0	$+\infty$	$+\infty$	$+\infty$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
A	$\{A\}$	0	4	6	1	$\emptyset$	A	A	A
D	$\{A, D\}$	0	4	2	1	$\emptyset$	A	D	A
C	$\{A, D, C\}$	0	3	2	1	$\emptyset$	C	D	A
B	$\{A, D, C, B\}$	0	3	2	1	$\emptyset$	C	D	A

Terminaison On passe au plus $|S|$ fois dans la boucle While.

Correction La correction découle du résultat suivant :

Proposition V.6

Après chaque passage dans la boucle, les deux propriétés suivantes sont vérifiées :
— *pour $x \in D$, $Dist(x) = d(s, x)$ et le chemin le plus court de s à x reste dans D ;*

— *pour $v \notin D$, $Dist(x) \geq d(s,x)$, et $Dist(x)$ est la longueur du plus court chemin de u_0 vers v dont tous les noeuds internes sont dans S.*

■

Démonstration :

Complexité L'algorithme de Dijkstra sur un graphe se termine en un temps de l'ordre $O(|S|^2)$.

V.2.6 Remarques

Le tableau ci-dessous récapitule les graphes sur lesquels chaque algorithme peut s'appliquer ainsi que leur complexité. On notera n le nombre de sommets et a le nombre d'arcs du graphe. La complexité de l'algorithme de Dijkstra peut être améliorée en choisissant une structure de données plus perfectionnée (file de Fibonacci).

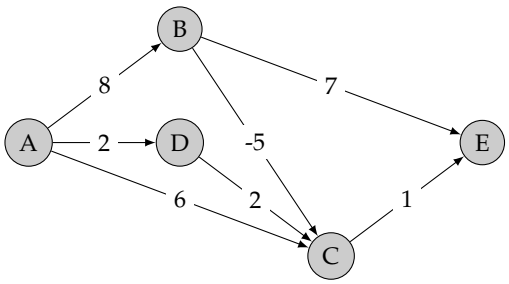
Algorithme	Type de graphe	Complexité
Bellman-Ford	tout type de graphe	$O(n^3)$
Bellman	graphe sans circuit	$O(n^2)$
Dijkstra	graphe de valuation positive	$O(n^2)$

V.2.7 Ordonnancement et gestion de projet

Les algorithmes de recherche de plus court chemin sont très utilisé dans l'ordonnancement

Tâches	Opérations et contraintes	Durée en jour
A	Début du projet	1
B	commence au plus tôt 7 jour après la fin de la tâche A commence au plus tard 5 jour après le début de la tâche C	3
C	commence au plus tôt 6 jour après le début de la tâche A commence au plus tôt 1 jour après la fin de la tâche D	1
D	commence au plus tôt 1 jour après la fin de la tâche A	1
E	commence après la fin de la tâche C commence 4 jours après la fin de la tâche B Fin du projet	0

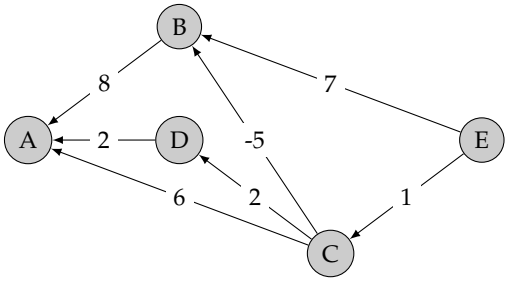
Si on traduit les contraintes sur un graphe, on obtient :



Si on recherche l'ordonnancement au plus tôt, cela revient à chercher les chemins maximaux. On décompose en niveau et on utilise Bellmon simplifié. Pour l'exemple on obtient :

	A	B	C	D	E
$Dist$	0	8	2	6	15
$Pred$	$\emptyset$	A	A	A	B

Si l'on veut réaliser le projet en 17 jours au plus et que l'on cherche l'ordonnancement au plus tard on inverse les arêtes et on applique Bellman pour rechercher les chemins les plus longs (attention les niveaux peuvent changer).



	A	B	C	D	E
Dist	15	7	1	3	0
Pred	B	E	E	C	$\emptyset$

Ainsi la tâche A doit être faite dans $[0, 2]$, la tâche B doit être faite dans $[8, 10]$, la tâche C doit être faite dans $[2, 16]$, la tâche D doit être faite dans $[6, 15]$ et la tâche E doit être faite dans $[15, 17]$.

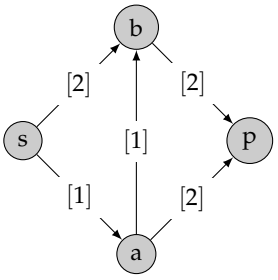
V.3 Flots dans les transports

V.3.1 Position du problème

Définition V.1. Un réseau de transport $R = (S, A, s, p, c)$ est définit par :
- un graphe orienté $G = (S, A)$ sans circuit,
- un sommet $s \in S$ appelé source ;
- un sommet $p \in S$ appelé puits ;
- une fonction capacité $c : A \to]0, +\infty[$;
- il existe au moins un chemin de s à p.

Ce type de graphe permet de modéliser de nombreuses situations :
- Réseau routier ; les capacités représentent le nombre maximal de voitures par heure
- Réseau de distribution d'eau, électricité, etc. ; les capacités représentent alors le débit maximal pouvant être fourni par chaque partie du réseau.

Exemple V.5. Le graphe suivant est un réseau de transport :



Le problème qui nous intéresse est alors d'optimiser le parcours global du réseau en tenant compte des contraintes données par les capacités limitées de chaque partie du réseau.

Définition V.2. Soit un *réseau de transport* $R = (S, A, s, p, c)$. Un *flot* sur R est une fonction $f : A \to \mathbb{R}_+$ telle que

— pour tout arcs $a \in A$, $f(a) \leq c(a)$;

— pour tout $s \in S$ et $x \in S \setminus \{s, p\}$, on a

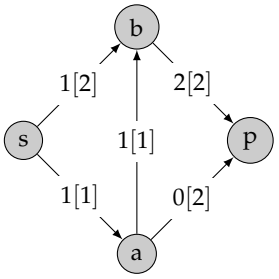
$$\underbrace{\sum_{z \in \text{Pred}(x)} f(z,x)}_{\text{flot entrant dans } x} = \underbrace{\sum_{y \in \text{Succ}(x)} f(x,y)}_{\text{flot sortant de } x}$$

On appelle *valeur* du flot la quantité

$$v(f) = \sum_{y \in \text{Succ}(s)} f(s,y) = \sum_{z \in \text{Pred}(p)} f(z,p)$$

Un arc $a \in A$ est *saturé* par le flot f si $f(a) = c(a)$.

Exemple V.6. On indique sur chaque arc la valeur du flot à côté de la capacité. Le graphe ci-dessous représente un flot f dans le réseau de transport de l'exemple précédent qui a pour valeur $v(f) = 2$.



V.3.2 Lemme de la coupe

Définition V.3. Une *coupe* sur un réseau de transport R est un sous-ensemble X de S tel que $s \in X$ et $p \notin X$.

On définit alors la *capacité* d'une coupe comme la somme des capacités des arcs allant de X à $S \setminus X$ noté par

$$C(X) = \sum_{(x,y) \in A, x \in X, y \in \overline{X}} c(x,y)$$

Exemple V.7. Les coupes possibles de l'exemple V.5 sont :

X	$\overline{X}$	$C(X)$
$\{s\}$	$\{a, b, p\}$	3
$\{s, a\}$	$\{b, p\}$	5
$\{s, b\}$	$\{a, p\}$	3
$\{s, a, b\}$	$\{p\}$	4

Il est clair qu'un flot ne pourra pas avoir une valeur supérieure à la capacité d'une coupe. La recherche d'une coupe de capacité la plus petite possible nous permettra donc de connaître les limites du réseau. La réciproque est vraie, on a le résultat suivant.

Théorème V.7

Soit $R = (S, A, s, p, c)$ un réseau de transport, il existe un flot maximal f tel que

$$v(f) = \min_{\text{coupe } X} C(X).$$

V.3.3 Algorithme de Ford-Fulkerson

On peut alors se demander comment calculer concrètement un flot de valeur maximale ainsi qu'une coupe de capacité minimale associée. L'idée est de considérer un chemin et d'augmenter progressivement les valeurs du flots jusqu'à arriver à saturation. On continue ensuite sur les autres chemins.

Définition V.4. Etant donné un réseau de transport $R = (S, A, s, p, c)$ et un flot f, un chemin γ de s à p est appelé *chemin augmentant* si pour tout arc a du chemin γ, on a $f(a) < c(a)$. On peut alors augmenter le flot sur ce chemin. On appelle *valeur résiduelle* d'un chemin γ de s à p le nombre $r(\gamma) = \min_{a \in \gamma}\{c(a) - f(a)\}$.

On a alors l'algorithme de Ford-Fulkerson :

Algorithm 14: Algorithme de Ford-Fulkerson

Data: Un réseau de transport $R = (S, A, s, p, c)$
Result: Un flot maximal

$f \leftarrow$ flot de départ (éventuellement nul);
while *il existe un chemin γ augmentant* **do**
  augmenter f le long du chemin γ;

Chapitre VI

Notion de théorie des langages

VI.1 Notion de langage

VI.1.1 Exemples de problèmes

La notion de langage est utilisé pour modéliser différents problèmes où l'information est stoquer sous une forme de chaine de caractères. Voici quelques exemples :

- Langage naturel : chaque mot est formé par un ensemble de lettre concaténé. L'ensemble des mots forme un dictionnairs. Puis ces mots sont organisé pour formé des phrases. Dans ce cas une structure apparaît qui est régit par la grammaire de la langue utilisé.
- Stoquer de l'information sur un disque dur : Toutes information stoquer sur un disque dur est codé par une succession de bits (0 ou 1), que ce soit du texte, image, musique... On peut se demander s'il est possible de compresser cette information, c'est à dire trouver une fonction qui a un chaine de $\{0, 1\}$ renvoie de manière bijective une chaîne plus courte.
- Recherche de chaîne de caractères dans un texte.
- Compilation : Un programme est une suite de caractère. Un compilateur s'intéresse essentiellement aux deux choses suivantes :
  - Analyse lexicale : on cherche les éléments de bases qui structure le programme (If, For, While, affectation...).
  - Analyse syntaxique : vérifie que les expressions sont correctes (ex : $var + var * var$ va être interprété comme $var + (var * var)$).
- Bio-informatique : L'ADN code l'information génétique à l'aide de 4 bases azotées : adénine (A), cytosine (C), guanine (G) ou thymine (T).

VI.1.2 Mots sur un alphabet fini

Mots sur un alphabet fini Soit Σ un *alphabet* fini. Un *mot* est une suite fini d'éléments de Σ on le note $u = u_1 u_2 \dots u_n$ et n est la longueur du mot u, noté $|u|$. Le mot vide est noté ε.

On note Σ^* l'ensemble des mots sur Σ et Σ^+ l'ensemble des mots sans le mot vide.

Opérations sur les mots Soient u et v deux mots de Σ^*, on définit la concaténation $w = u.v$ comme le mot de longueur $|u| + |v|$ tel que $w = u_1 u_2 \dots u_{|u|} v_1 v_2 \dots v_{|v|}$.

On dit que v est un *préfixe* de u s'il existe un mot w tel que $u = v.w$ et v est un *suffixe* de u s'il existe un mot w tel que $u = w.v$.

Ordre sur les mots Il existe différentes notions pour ordonner les mots d'un langage. Le plus connu est certainement l'ordre lexicographique qui permet de faire un dictionnaire.

Si on a un ordre $\leq$ sur Σ on définit l'*ordre lexicographique* sur Σ^* par

$$u \leq_{lex} v \iff (u \text{ préfixe de } v) \text{ ou } (\exists m \in \mathbb{N} \text{ tel que } u_1 \dots u_m = v_1 \dots v_m \text{ et } u_{m+1} \leq v_{m+1})$$

Distance sur les mots On peut définir différentes distances sur les mots. On s'intéressera ici à la distance édition définie comme étant le plus petit nombre d'opérations d'édition élémentaires nécessaires pour transformer le mot u en le mot v. Les opérations d'édition élémentaires sont la suppression ou l'insertion d'un symbole.

De multiples variantes de cette notion de distance ont été proposées, qui utilisent des ensembles d'opérations différents et/ou considèrent des poids variables pour les différentes opérations. Pour prendre un exemple réel, si l'on souhaite réaliser une application qui « corrige » les fautes de frappe au clavier, il est utile de considérer des poids qui rendent d'autant plus proches des séquences qu'elles ne diffèrent que par des touches voisines sur le clavier, permettant d'intégrer une modélisation des confusions de touches les plus probables.

L'utilitaire Unix `diff` implante une forme de calcul de distances. Cet utilitaire permet de comparer deux fichiers et d'imprimer sur la sortie standard toutes les différences entre leurs contenus respectifs.

VI.1.3 Langage

Un *langage* $\mathcal{L}$ sur un alphabet fini Σ est un ensemble de mots définis sur Σ autrement dit $\mathcal{L} \subset \Sigma^*$.

Exemple VI.1. Exemple d'un langage sur $\Sigma = \{0, 1\}$:

- $\emptyset$;
- $\{\varepsilon\}$;
- $\Sigma^* = \{\varepsilon, 0, 1, 00, 01, 10, 11, 000, 001, \dots\}$;
- $\Sigma^+ = \{0, 1, 00, 01, 10, 11, 000, 001, \dots\}$;
- tout ensemble fini de mots ;
- $\{0^n : n \in \mathbb{N}\}$;
- $\{0^n : n \in \mathbb{N}\}$;
- $\{0^n 1^m : n, m \in \mathbb{N}\}$;
- $\{0^n 1^n : n \in \mathbb{N}\}$;
- $\{0^p : p \in \mathbb{N} \text{ nombre premier}\}$;
- $\{u \in \Sigma^* : u \text{ est le codage en binaire d'un nombre premier}\}$;
- $\{u \in \Sigma^* : u \text{ est un palindrome}\}$;
- $\{u \in \Sigma^* : u \text{ est un code html certifié}\} \neq \{u \in \Sigma^* : u \text{ est un code html bien interprété par Firefox}\}$;
- $\{u \in \Sigma^* : u \text{ est le codage en MP3 de votre chanson préférée}\}$;
- $\{u \in \Sigma^* : u \text{ est le codage en assembleur d'un programme qui s'arête sur l'entrée vide}\}$;
- ...

Soient $\mathcal{L}$, $\mathcal{L}_1$ et $\mathcal{L}_2$ des langages sur un alphabet fini Σ. On peut définir différentes opérations sur les langages :

- *Union* : $\mathcal{L}_1 \cup \mathcal{L}_2$ langage comportant des mots de $\mathcal{L}_1$ ou de $\mathcal{L}_2$;

— *Intersection* : $\mathcal{L}_1 \cap \mathcal{L}_2$ langage comportant des mots de $\mathcal{L}_1$ et de $\mathcal{L}_2$;
— *Compémentaire* : $\overline{\mathcal{L}}$ langage comportant des mots de Σ^* qui ne sont pas dans $\mathcal{L}$;
— *Concaténation* : $\mathcal{L}_1.\mathcal{L}_2$ langage comportant les mots formés en concaténant un mot $\mathcal{L}_1$ à un mot de $\mathcal{L}_2$

$$\mathcal{L}_1.\mathcal{L}_2 = \{u_1.u_2 : u_1 \in \mathcal{L}_1 \text{ et } u_2 \in \mathcal{L}_2\};$$

— *Puissance* : On définit par récurrence la puissance $n^{\text{ème}}$ de $\mathcal{L}$, noté $\mathcal{L}^n$ par

$$\mathcal{L}^0 = \{\varepsilon\} \text{ et } \mathcal{L}^{n+1} = \mathcal{L}.\mathcal{L}^n.$$

Attention, il ne faut pas confondre, on a $\mathcal{L}^n = \{u \in \Sigma^* : \text{ il existe } u_1, u_2, \ldots, u_n \in \mathcal{L} \text{ tel que } u = u_1.u_2.\cdots.u_n\}$ qui en général est différent de $\{u^n : n \in \mathbb{N}\}$.
— *Fermeture de Kleene* : On définit

$$\mathcal{L}^* = \bigcup_{n\geq 0} \mathcal{L}^n \qquad \text{et} \qquad \mathcal{L}^+ = \bigcup_{n>0} \mathcal{L}^n.$$

VI.2 Langage rationnel

Langage rationnel On définit les langages rationnels comme le plus petit ensemble de langages qui contient tout les singletons et clos par les opérations usuelles : union, intersection et concaténation.

Définition VI.1. Soit Σ un alphabet fini. Les langages *rationnels* sont définis inductivement par :
— $\emptyset$ et $\{\varepsilon\}$ sont des langages rationnels ;
— pour tout $a \in \Sigma$, $\{a\}$ est un langage rationnel ;
— si $\mathcal{L}_1$ et $\mathcal{L}_2$ sont des langages rationnels, alors $\mathcal{L}_1 \cup \mathcal{L}_2$, $\mathcal{L}_1.\mathcal{L}_2$ et $\mathcal{L}_1^*$ sont des langages rationnels.

Exemple VI.2. Voilà quelques exemples de langages rationnels :
— tous les langages finis sont rationnels et en particulier Σ ;
— Σ^* est rationnel ;
— $\Sigma^+ = \Sigma.\Sigma^*$;
— le langage des mots sur $\Sigma = \{0, 1\}$ qui contient au moins une fois le mot 111 est rationnel car il s'écrit $\Sigma^*.\{111\}.\Sigma^*$;
— le langage des mots sur $\Sigma = \{0, 1\}$ qui contient un nombre pair de fois la lettre 1 est rationnel car il s'écrit $(\{b\}^*.\{a\}.\{b\}^*.\{a\}.\{b\}^*)^*$;

Expression rationnelle Les expressions rationnelles définissent un système de formules qui simplifient et étendent ce type de notation des langages rationnels.

Définition VI.2. Soit Σ un alphabet on définit de manière inductive les expressions rationnelles :
— $\emptyset$ et ε sont des expressions rationnelles ;
— pour tout $a \in \Sigma$, a est une expression rationnelle ;
— si e_1 et e_2 sont deux expressions rationnelles, alors $e_1 + e_2$, $e_1.e_2$ et e_1^* sont également des expressions rationnelles.

Tout langage rationnel peut être représenté par une expression rationnelle : ε dénote le langage $\{\varepsilon\}$ et $\emptyset$ dénote le langage vide, a dénote le langage $\{a\}$ pour $a \in \Sigma$, $e_1 + e_2$ (resp. $e_1.e_2$, e_1^*) dénote l'union (resp. la concaténation, l'étoile) des langages dénotés par e_1 et par e_2.

Une question naturelle est : Etant donné deux expressions rationnelles est ce que les deux langages associés sont identiques ?

Application : Recherche de motifs dans un texte Les expressions rationnelles constituent un outil pour décrire des langages simples (rationnels). Ces formules sont par exemple utilisé pour effectué des recherche des occurrence d'un motif. On notera deux applications concrètes notables :

— Sous UNIX, il existe par exemple un utilitaire `grep` qui permet de rechercher les occurrences d'un motif dans un fichier texte. La commande suivante imprime sur la sortie standard toutes les lignes du fichier 'cours.pdf' contenant au moins une occurrence du mot graphe :

```
> grep 'graphe' cours.pdf
```

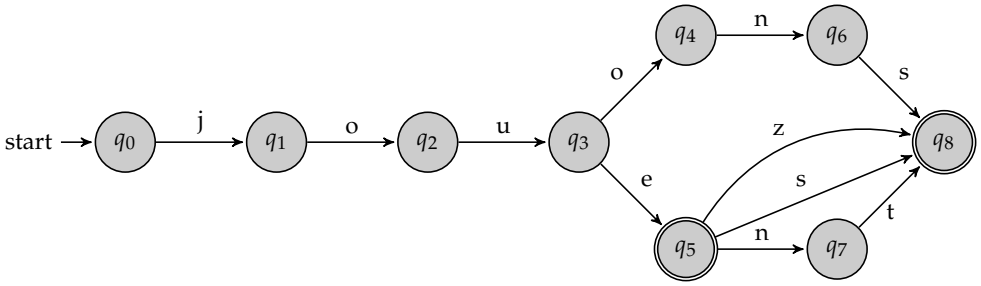
Il est possible de faire des recherche de répétition (`e+`) ...

— L'analyse lexicale se trouve tout au début de la chaîne de compilation. C'est la tâche consistant à décomposer une chaîne de caractères en lexèmes, qui vont ensuite être analysé par l'analyseur syntaxique qui va ensuite les interpréter.

VI.3 Automates fini

VI.3.1 Définitions

Les automates finis est un modèle qui permet de définir la notion de calcul de langage simple. On les représente par un graphe orienté dont les arcs sont étiquetés par des symboles. On part d'un état initial et on parcoure le graphe telle sorte qu'on lise le mot.



Définition VI.3. Un *automate fini* $\mathcal{A}$ est définit par un un quintuplet $(\Sigma, Q, i, F, \Delta)$ où :

— Σ est un alphabet fini ;
— Q est un ensemble d'états ;
— $i \in Q$ est l'état initial ;
— $F \subset Q$ est l'ensemble des états finaux ;
— $\Delta \subset Q \times \Sigma \times Q$ est l'ensemble des transition de $\mathcal{A}$.

Un automate fini est complet si de tout état p et toute lettre a il existe une transition de la forme (p, a, q).

On représente un automate fini par un graphe dont les sommets sont les états et on a un arcs de l'état q_1 à l'état q_2 avec la valuation $a \in \Sigma$ si $(q_1, a, q_2) \in \Delta$.

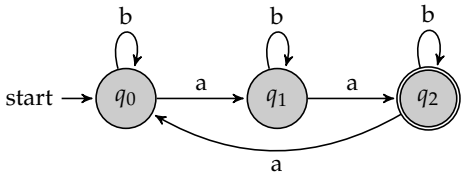
Quitte à avoir un état puits, il est possible de complété un automate fini en rajoutant des transition vers cet état puits lorsqu'elle manque.

Définition VI.4. Un mot $u = u_1 \dots u_n$ est *reconnu* par un automate $\mathcal{A} = (\Sigma, Q, q_0, F, \Delta)$ s'il existe une suite d'états $q_0, q_1, \dots, q_n$ tel que $(q_{i-1}, u_i, q_i) \in \Delta$ pour tout $i \in [1, n]$. Autrement dit s'il existe un chemin dans le graphe de l'état initial à l'état final tel que les valuations correspondent au mot u.

Définition VI.5. Etant donné un automate $\mathcal{A} = (\Sigma, Q, q_0, F, \Delta)$, on définit le *langage reconnu* par $\mathcal{A}$, noté $\mathcal{L}(\mathcal{A})$, comme l'ensemble des mots reconnus par l'automate $\mathcal{A}$.

Un langage est *régulier* s'il existe un automate fini qui le reconnaît.

Exemple VI.3. Considérons l'automate suivant $\mathcal{A} = (\Sigma, Q, q_0, F, \Delta)$ où
— $\Sigma = \{a, b\}$;
— $Q = \{q_0, q_1, q_2\}$;
— $F = \{q_2\}$;
— $\delta = \{(q_0, b, q_0), (q_0, a, q_1), (q_1, b, q_1), (q_1, a, q_2), (q_2, b, q_2), (q_2, a, q_0)\}$.
Sa représentation graphique est :



Des exemples de mots reconnu : *aa*, *abaabbaba*... Des exemples de mots non reconnu ; *abaa*, *ab*...

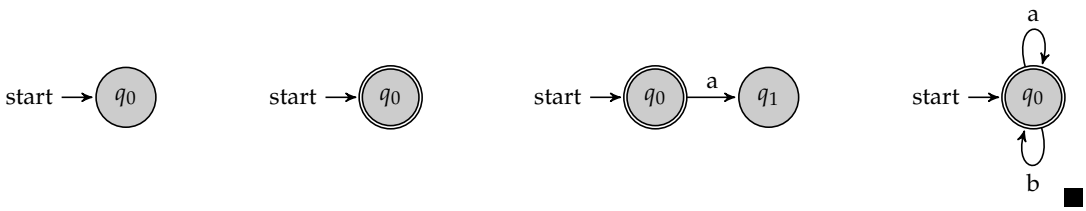
En fait $\mathcal{L}(\mathcal{A}) = \{u \in \Sigma^* : u \text{ contient } 3n + 2 \text{ lettres } a \text{ avec } n \in \mathbb{N}\}$.

VI.3.2 Stabilité aux opérations usuelles

Proposition VI.1

Les langages suivants sont réguliers :
— $\emptyset$;
— $\{\epsilon\}$;
— $\{a\}$ *pour* $a \in \Sigma$;
— Σ^*.

Démonstration : Les automates finis suivants reconnaissent les langages de la proposition :



■

Proposition VI.2

L'union de deux langages réguliers est un langage régulier.

Démonstration : Soient $\mathcal{A}_1 = (\Sigma, Q_1, i_1, F_1, \Delta_1)$ et $\mathcal{A} = (\Sigma, Q_2, i_2 F_2, \Delta_2)$ deux automates finis. Quitte à re-numéroter les états, on peut supposer que $Q_1 \cap Q_2 = \emptyset$. On définit l'automate $\mathcal{A} = (\Sigma, Q, i, F, \Delta)$ tel que :

— $Q = Q_1 \cup Q_2 \cup \{i\}$ où $i \notin Q_1 \cup Q_2$;
— $F = F_1 \cup F_2$;
— $\Delta = \Delta_1 \cup \Delta_2 \cup \{(i, a, q) \in Q \times \Sigma \times Q$ tel que $(i_1, a, q) \in \Delta_1$ où $(i_2, a, q) \in \Delta_2\}$.

Si $u \in \mathcal{L}(\mathcal{A})$ alors il existe un chemin dans $\mathcal{A}$ de i dans F. Si l'état final est dans F_1, il suffit de remplacer l'état initial par i_1 pour trouver un chemin de i_1 à F_1 dans $\mathcal{A}_1$ qui reconnait u. De même, si l'état final est dans F_2 on trouve un chemin dans $\mathcal{A}_2$ qui reconnait u. Ainsi $\mathcal{L}(\mathcal{A}) \subset \mathcal{L}(\mathcal{A}_1) \cup \mathcal{L}(\mathcal{A}_2)$.

Réciproquement si $u \in \mathcal{L}(\mathcal{A}_1)$ on trouve un chemin de i_1 à F_1 dans $\mathcal{A}_1$ qui reconnait u. En remplaçant le premier état par i on a bien un chemin acceptant de u dans $\mathcal{A}$ d'ou $\mathcal{L}(\mathcal{A}_1) \subset \mathcal{L}(\mathcal{A})$. De même si $u \in \mathcal{L}(\mathcal{A}_2)$. Ainsi $\mathcal{L}(\mathcal{A}_1) \cup \mathcal{L}(\mathcal{A}_2) \subset \mathcal{L}(\mathcal{A})$.

On en déduit que $\mathcal{L}(\mathcal{A}_1) \cup \mathcal{L}(\mathcal{A}_2) = \mathcal{L}(\mathcal{A})$. ■

Proposition VI.3

L'intersection de deux langages réguliers est un langage régulier.

Démonstration : Soient $\mathcal{A}_1 = (\Sigma, Q_1, i_1, F_1, \Delta_1)$ et $\mathcal{A} = (\Sigma, Q_2, i_2 F_2, \Delta_2)$ deux automates finis. Quitte à re-numéroter les états, on peut supposer que $Q_1 \cap Q_2 = \emptyset$. On définit l'automate $\mathcal{A} = (\Sigma, Q, i, F, \Delta)$ tel que :
— $Q = Q_1 \times Q_2$;
— $i = (i_1, i_2)$,
— $F = F_1 \times F_2$;
— $\Delta = \{((p_1, p_2), a, (q_1, q_2)) \in Q \times \Sigma \times Q$ tel que $(p_1, a, q_2) \in \Delta_1$ et $(p_2, a, q_2) \in \Delta_2\}$
On vérifie que $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}_1) \cap \mathcal{L}(\mathcal{A}_2)$ ■

Proposition VI.4

Le complémentaire d'un langage régulier est un langage régulier.

Démonstration : Soit $\mathcal{A} = (\Sigma, Q, i, F, \Delta)$ un automate fini. D'abord on le complète à l'aide d'un état poubelle $poub \notin Q$: pour tout état $q \in Q$ et lettre $a \in \Sigma$, soit il existe une transition $(q, a, q') \in \Delta$ soit on rajoute la transition $(q, a, poub)$.

Ensuite on définit l'automate $\overline{\mathcal{A}} = (\Sigma, Q \cup \{poub\}, i, (Q \cup \{poub\}) \setminus F, \Delta)$.

Si u est reconnu par $\mathcal{A}$ alors il n'est pas reconnu par $\overline{\mathcal{A}}$ et réciproquement s'il est reconnu par $\overline{\mathcal{A}}$ alors il n'est pas reconnu par $\mathcal{A}$. ■

Proposition VI.5

La concaténation de deux langages réguliers est un langage régulier.

Démonstration : Soient $\mathcal{A}_1 = (\Sigma, Q_1, i_1, F_1, \Delta_1)$ et $\mathcal{A} = (\Sigma, Q_2, i_2 F_2, \Delta_2)$ deux automates finis. Quitte à re-numéroter les états, on peut supposer que $Q_1 \cap Q_2 = \emptyset$. On définit l'automate $\mathcal{A} = (\Sigma, Q, i, F, \Delta)$ tel que :
— $Q = Q_1 \cup Q_2$;
— $i = i_1$;
— $F = F_1 \cup F_2$ si $i_2 \in F_2$ ou bien $F = F_2$ sinon ;
— $\Delta = \Delta_1 \cup \Delta_2 \cup \{(p, a, q) \in Q \times \Sigma \times Q$ tel que $p \in Q_1$ et $(i_2, a, q) \in \Delta_2\}$.
On vérifie que $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{A}_1).\mathcal{L}(\mathcal{A}_2)$. ■

Proposition VI.6

L'étoile d'un langage régulier est un langage régulier.

Démonstration : Soit $\mathcal{A} = (\Sigma, Q, i, F, \Delta)$ un automate. Soit $q \notin Q$ un nouvel état, on définit l'automate $\mathcal{A}_* = (\Sigma, Q \cup \{i'\}, i', F \cup \{i'\}, \Delta')$ tel que $i' \notin Q$ et

$$\Delta' = \Delta \cup \{(p, a, q) \in Q \times \Sigma \times Q \text{ tel que } p \in F \cup \{i'\} \text{ et } (i, a, q) \in \Delta_2\}.$$

On vérifie que $\mathcal{L}(\mathcal{A}_*) = \mathcal{L}(\mathcal{A})^*$. ■

Proposition VI.7

La différence symétrique de deux langages réguliers est un langage régulier.

Démonstration : En effet, si $\mathcal{L}_1$ et $\mathcal{L}_2$ sont régulier alors $\mathcal{L}_1 \setminus \mathcal{L}_2 = \mathcal{L}_1 \cap \overline{\mathcal{L}_2}$ est aussi régulier d'après les propriétés précédentes. ■

VI.3.3 Théorème de Kleene

Nous allons établir le lien entre langage régulier et langage rationnel.

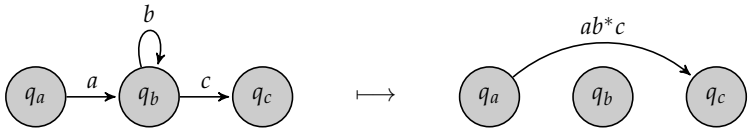
Théorème VI.8 Théorème de Kleene

Un langage est régulier si et seulement s'il est rationnel.

Démonstration : L'ensemble des langages réguliers contient les langages $\emptyset$, $\{\varepsilon\}$ et $\{a\}$ pour $a \in \Sigma$ et qu'il est stable par union, concaténation et étoile. On en déduit que tout langage rationnel est régulier.

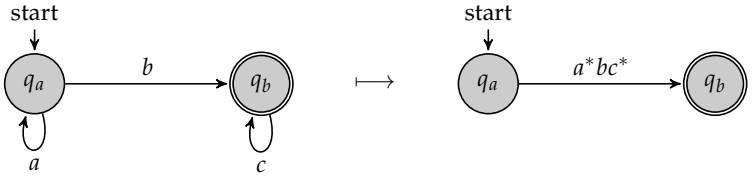
Etant donné un langage reconnu par un automate $\mathcal{A} = (\Sigma, Q, q_0, F, \Delta)$, pour donner l'expression régulière correspondant au langage $\mathcal{L}(\mathcal{A})$ on cherche à éliminer pas à pas les sommets et les transitions suivant deux règles :

1. supprimer les états intermédiaires :



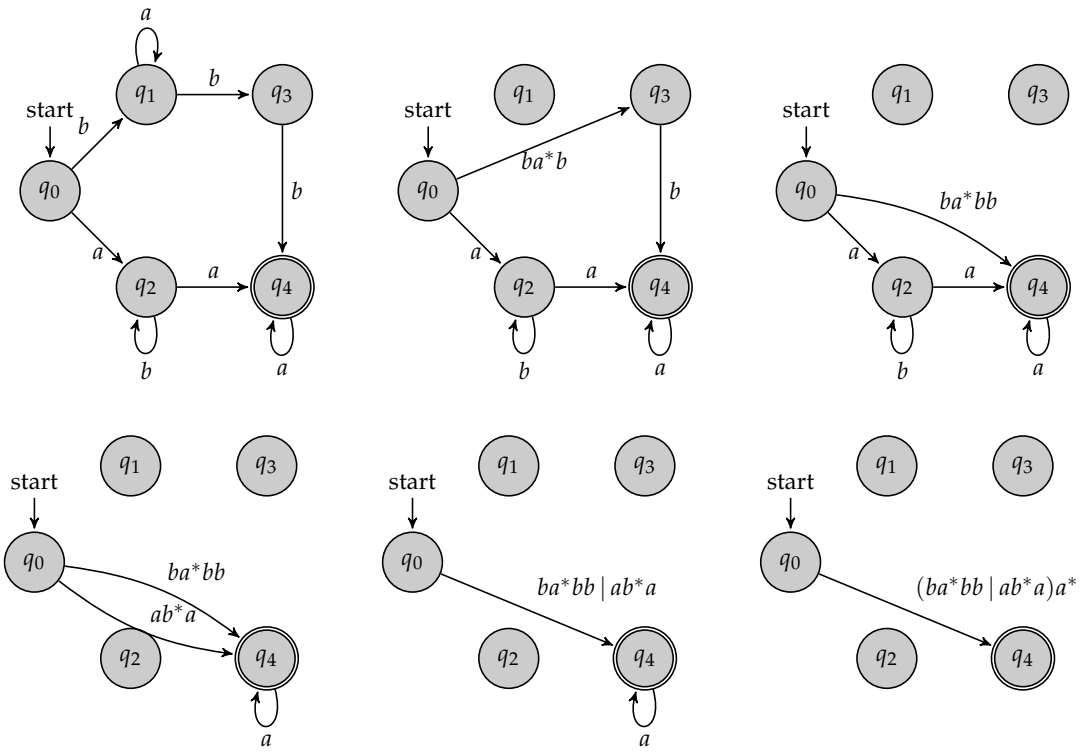
2. supprimer les transitions multiples apparus après l'étape 1.

L'algorithme consiste à supprimer des arcs afin d'aboutir à un automate de la forme suivante dont l'expression régulière associée est a^*bc^* :



On prouve la correction de cet algorithme par récurrence. ■

Exemple VI.4. Cherchons l'expression rationnelle associée à l'automate suivant



Corollaire VI.9

L'ensemble des langages rationnels est stable par complémentaire, intersection, différence symétrique.

VI.4 Comment montrer qu'un langage n'est pas rationnel ?

Exemple VI.5. Considérons le langage $\mathcal{L} = \{a^n b^n\}$ on peut se demander si ce langage est rationnel. Supposons qu'il le soit, il existe un automate $\mathcal{A} = (\Sigma, Q, i, F, \Delta)$ qui le reconnait. Cet automate reconnait donc le mot $a^{|Q|}b^{|Q|}$. Lors de la lecture du mot $a^{|Q|}$, on passe deux fois par le même état on a une boucle de longueur $i \geq 1$, ainsi le mot $a^{|Q|+i}b^{|Q|}$ est reconnu par l'automate mais n'est pas dans $\mathcal{L}$.

On utilise la stabilité de l'intersection et cet exemple pour montrer que le langage HTML certifié n'est pas rationnel.

On peut généraliser cet exemple avec le résultat suivant.

Théorème VI.10 Lemme de pompage

Soit $\mathcal{L}$ un langage rationnel. Il existe un entier k (le nombre d'état d'un automate qui reconnait $\mathcal{L}$) tel que pour tout mot $x \in \mathcal{L}$ on a une factorisation $x = uvw$ ou u,v et w sont des mots qui vérifient :

- *$|v| \geq 1$,*
- *$|uv| \leq k$,*
- *pour tout $i \in \mathbb{N}^*$ on a uv^iw dans $\mathcal{L}$.*

Démonstration : Soit $\mathcal{A}$ un automate fini à k états reconnaissant $\mathcal{L}$. S'il n'existe pas de mot de $\mathcal{L}$ de longueur supérieure ou égale à k, le résultat est immédiat. Sinon, soit x un mot de $\mathcal{L}$, de longueur supérieure ou égale à k. La reconnaissance de x dans $\mathcal{A}$ correspond à une suite d'états $q_0 \dots q_{|x|}$. Comme $\mathcal{A}$ n'a que k états, le préfixe de longueur k de cette séquence

contient nécessairement deux fois le même état q, aux indices i et j, avec $0 \leq i < j \leq k$. Si l'on note $u = x_1 \dots x_i$ et $v = x_{i+1} \dots x_j$ alors on a bien les deux première condition tu du théorème. Comme lors du calcul de x on est sur q au début de la lecture de v et on termine sur l'état q, ainsi en court-circuitant ou en itérant les parcours le long du circuit $q \dots q$ on en déduit que le mot $uv^i w$ est accepté par $\mathcal{A}$. ■

Exemple VI.6. On peut montrer avec ce lemme que les langages suivants ne sont pas rationnels :

— $\{u\overline{u} \text{ tel que } u \in \Sigma^*\} = \{u \in \Sigma^* \text{ tel que } u \text{ est un palindrome}\}$ où $\overline{u}$ est le mot miroir de u ;
— $\{a^n : n \text{ premier}\}$;
— $\{a^n b^n c^n : n \in \mathbb{N}\}$;
— ...

Si un langage est infini, alors il contient des mots de longueur arbitrairement grande. Ce que dit ce lemme, c'est essentiellement que dès qu'un mot de $\mathcal{L}$ est assez long, il contient un facteur différent de ϵ qui peut être itéré à volonté tout en restant dans $\mathcal{L}$. En d'autres termes, les mots ''longs'' de $\mathcal{L}$ sont construits par répétition d'un facteur s'insérant à l'intérieur de mots plus courts. Lorsque le langage n'est pas infini, le lemme devient trivial car on peut choisir un k qui majore la longueur de tous les mots du langage. Avec le même type de raisonnement, on peut montrer a proposition suivante.

Proposition VI.11

Si $\mathcal{A}$ est un automate fini contenant k états :
— *$\mathcal{L}(\mathcal{A})$ est non vide si et seulement si $\mathcal{A}$ reconnaît un mot de longueur strictement inférieure à k.*
— *$\mathcal{L}(\mathcal{A})$ est infini si et seulement si $\mathcal{A}$ reconnaît un mot u tel que $k \leq |u| < 2k$.*

On en déduit un algorithme pour savoir si un automate à k états reconnait l'ensemble vide, un lavage fini ou un langage infini.

Pour savoir si deux automates $\mathcal{A}_1$ et $\mathcal{A}_2$ reconnaissent le même langage, il suffit de construire l'automate qui reconnait $\mathcal{L}(\mathcal{A}_1) \setminus \mathcal{L}(\mathcal{A}_2) = (\mathcal{L}(\mathcal{A}_1) \cap \overline{\mathcal{L}(\mathcal{A}_2)}) \cup (\overline{\mathcal{L}(\mathcal{A}_1)} \cap \mathcal{L}(\mathcal{A}_2))$ et de voir si cet automate reconnait le mot vide.

VI.5 Déterminisation, minimisation, epsilon transition

VI.5.1 D'autres modèles de calcul pour défini les langages rationels

On peut modifier la définition d'automate fini pour définir d'autre type de reconnaissance

Définition VI.6. Un *automate fini à ε-transition* $\mathcal{A}$ est définit par un un quintuplet $(\Sigma, Q, i, F, \Delta)$ où :

— Σ est un alphabet fini ;
— Q est un ensemble fini d'états ;
— $i \in Q$ est l'état initial ;
— $F \subset Q$ est l'ensemble des états finaux ;
— $\Delta \subset Q \times (\Sigma \cup \{\varepsilon\}) \times Q$ est l'ensemble des transition de $\mathcal{A}$.

Définition VI.7. Un *automate fini déterministe* $\mathcal{A} = (\Sigma, Q, i, F, \Delta)$ est un automate fini tel que pour tout état $p \in Q$ et tout lette $a \in \Sigma$, il existe au plus un état q tel que $(p, a, q) \in \Delta$. On représente alors Δ par une fonction $\delta : Q \times \Sigma \to Q$.

Ainsi un automate fini déterministe est définit par un un quintuplet $(\Sigma, Q, i, F, \delta)$ où :
— Σ est un alphabet fini ;
— Q est un ensemble fini d'états ;
— $i \in Q$ est l'état initial ;
— $F \subset Q$ est l'ensemble des états finaux ;
— $\delta : Q \times \Sigma \longrightarrow Q$ est la fonction de transition de $\mathcal{A}$.

On défini la notion de langage reconnu de façon analogue au langage reconnu par un automate fini non déterministe.

Théorème VI.12

Les langages reconnus par les automates déterministe ou les automates à ε-transition sont exactement les langages rationnels.

VI.5.2 Déterminisation

Soit $\mathcal{A} = (\Sigma, Q, i, F, \Delta)$ un automate fini, on définit $\det(\mathcal{A}) = (\Sigma, \mathcal{P}(Q), \{i\}, F', \delta)$ où
— Σ est un alphabet fini ;
— $\mathcal{P}(Q)$ est l'ensemble des parties de Q ;
— $\{i\}$ est un élément de $\mathcal{P}(Q)$, c'est le singleton contenant l'état initial de $\mathcal{A}$;
— $F' = \{M \in \mathcal{P}(Q) \text{ tel que } M \cap F \neq \emptyset\}$ est l'ensemble des parties de Q qui contient au moins un état final ;
— $\delta : \mathcal{P}(Q) \times \Sigma \longrightarrow \mathcal{P}(Q)$ tel que pour $M \in \mathcal{P}(Q)$ et $a \in \Sigma$ on pose

$$\delta(M, a) = \{q \in Q \text{ tel qu'il existe } p \in M \text{ vérifiant } (p, a, q) \in \Delta\} \in \mathcal{P}(Q),$$

c'est la fonction de transition de $\det(\mathcal{A})$.

Clairement $\det(\mathcal{A})$ est déterministe, nous allons montrer que $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\det(\mathcal{A}))$.

VI.5.3 Automate minimal

VI.6 Applications

Il existe de nombreuses applications que l'on verra en TD :
— recherche d'un motif dans un texte (adn, éditeur de texte…) ;
— analyse lexicale qui se trouve tout au début de la chaîne de compilation ;
— linguistique ;
— reconnaissance d'ensemble d'entiers ;
— décidabilité de formule arithmétique (Presburger).

VI.7 D'autres types de langages

On a vu que tout les langages ne sont pas rationnels, dans cette section on va donner une ouverture sur les différents types de langages.

VI.7.1 Langage décidable/indécidable

Un langage $\mathcal{L}$ est *décidable* s'il existe un programme qui prend en entrée un mot u et renvoie 1 si $u \in \mathcal{L}$ et 0 sinon. En fait il existe des langages *indécidables*, c'est à dire qu'il n'existe pas de programmes qui décide si un mot est dans le langage ou non.

Considérons le langage

$$\mathcal{L}_{\texttt{Turing}} = \{u \in \Sigma^* : u \text{ est le code d'un programme qui s'arête sur l'entrée vide}\}.$$

Supposons que ce langage est décidé par un programme `SuperProg`. Construisons le programme `Bug` de la manière suivante :

$$\texttt{Bug}(u) = \begin{cases} 1 & \text{si } \texttt{SuperProg}(\texttt{Bug}) = 0 \\ \text{boucle infini} & \text{sinon} \end{cases}$$

Que vaut $\texttt{Bug}(\emptyset)$?
— si $\texttt{Bug}(\emptyset)$ s'arrête alors $\texttt{SuperProg}(\texttt{Bug}) = 1$ et on devrait avoir une boucle infini,
— sinon $\texttt{SuperProg}(\texttt{Bug}) = 0$ et $\texttt{Bug}(\emptyset)$ s'arrête sur l'entrée vide.
De même on montre que

$$\mathcal{L}'_{\texttt{Turing}} = \{u.v \in \Sigma^* : u \text{ est le code d'un programme qui s'arête sur l'entrée } v\}$$

est indécidable.

VI.7.2 Grammaires

Définition VI.8. Une *grammaire* formelle est constituée des quatre objets suivants :
— un ensemble fini de symboles V appelé *vocabulaire* formé de deux sous ensembles $V = T \cup N$:
— des symboles *terminaux* $T = \Sigma \cup \{\varepsilon\}$, où Σ est l'alphabet du langage),
— des symboles *non-terminaux* N (notés conventionnellement par des ma- juscules) ;
— Un élément de l'ensemble des non-terminaux, appelé *axiome*, noté S_0,
— Un ensemble de *règles* de la forme $\alpha \to \beta$ où $\alpha \in V^*NV^*$ contient un non-terminal et $\beta \in V^*$ est une concaténation de terminaux et de non-terminaux.

Le langage associé à une grammaire est l'ensemble des mots contenant uniquement des symboles terminaux obtenus en appliquant successivement les règles en partant de S_0.

Exemple VI.7.
— $\Sigma = \{a, b\}$ et $N = \{S_0\}$ on définit pour règle $S_0 \to \varepsilon$ et $S_0 \to aS_0b$. On obtient le langage $\{a^n b^n : n \in \mathbb{N}\}$.
— $\Sigma = \{a, b\}$ et $N = \{S_0\}$ on définit pour règle $S_0 \to \varepsilon$, $S_0 \to aaS_0$ et $S_0 \to bS_0$. On obtient le langage $\{u \in \Sigma^* :$ tout block de a contient un nombre pair de $a\}$.
— $\Sigma = \{a, b\}$ et $N = \{S_0, S_1\}$ on définit pour règle $S_0 \to \varepsilon$, $S_0 \to bS_0$, $S_1 \to bS_1$, $S_0 \to aS_1$ et $S_1 \to aS_0$. On obtient le langage $\{u \in \Sigma^* : u$ contient un nombre pair de $a\}$.
— $\Sigma = \{a, b, c\}$ et $N = \{S_0\}$ on définit pour règle $S_0 \to \varepsilon$, $S_0 \to aS_0S_1b$, $S_1b \to bS_1$, $bS_1 \to c$ et $cS_1 \to c$. On obtient le langage $\{a^n b^n c^n : n \in \mathbb{N}\}$.

A partir de là on peut définir la *hiérarchie de Chomsky*. Soit $\mathcal{L}$ un langage formel sur un alphabet Σ défini par un vocabulaire $V = N \cup T$ où $T = \Sigma \cup \{\varepsilon\}$ et $N = \overline{T}$. Suivant les propriétés des règles, $\mathcal{L}$ appartient à une des catégorie suivante :

Langages de type 3 ou rationnel : Ce sont les langages dont la grammaire est linéaire à gauche (ou à droite). C'est à dire que les règles sont de la forme $A \to aB$ où $A \to a$.

Langage de type 2 ou langage algébriques : Ce sont les langages dont les règles sont de la forme $A \to aBb$ ou $A \to a$ avec $A, B \in N$ et $a, b \in T$.

Langage de type 1 ou langage contextuels : Ce sont les langages dont les règles sont de la forme $\alpha A\beta \to \alpha\gamma\beta$ avec $A \in N$ et $\alpha, \beta, \gamma \in V$ avec $\beta \neq \varepsilon$.

Langage de type 0 ou langage récursivement écumérable : Tous les langages décrient par une grammaire formelle.