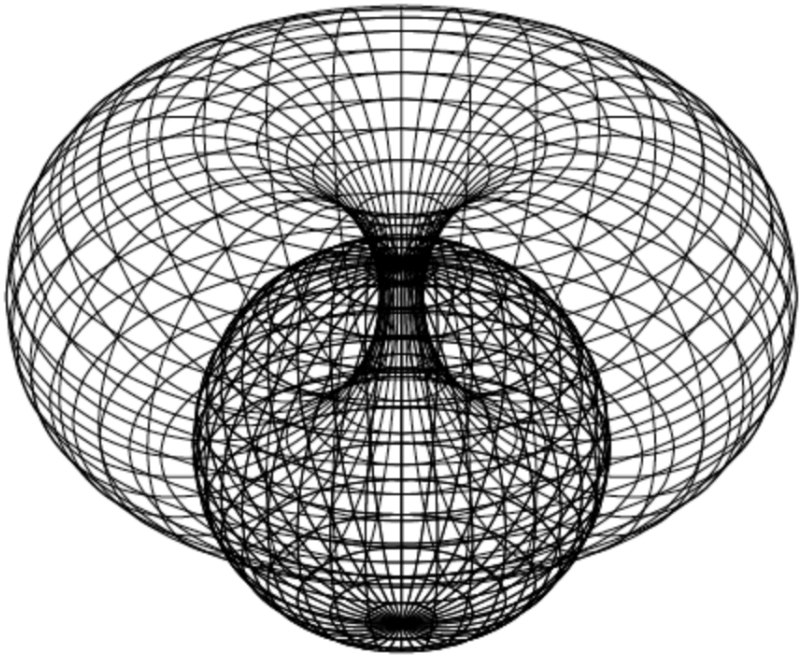
Fiches de Cours
Licence en Mathématiques



Université Jean François Champollion, Albi
Années universitaires 2022-2025

12 septembre 2025

Table des matières

Annexe 1 - Graphes et Théorie des Langages

Chapitre 1

Théorie des Graphes

Contents

Fiche réalisée grâce au cours de Thierry Montaut et Laura Brillon.
Dans ce cours, on note $G = (X, E)$ un graphe.

1.1 Graphes, Représentations et Parcours

1.1.1 Définitions - Graphes Orientés et Non Orientés

Vocabulaire

Définition (**Graphe non orienté**) . On appelle graphe non orienté un couple d'ensembles finis $G = (X, E)$ où $X = \{1, \dots, n\}$ représente les sommets du graphe et $E = \{(x_i, y_j), \dots\}, x_i, y_j \in X$ l'ensemble des arrêtes du graphe. Une arrête est une liaison entre deux sommets.
Un graphe est dit **simple** s'il n'existe pas de double arrêtes entre deux sommets ou de bouble (i.e une arrête de la forme (x, x)). Autrement, on parle de **multigraphe.**

Définition (**Graphe Orienté**) . On appelle graphe orienté un couple d'ensembles finis $G = (X, E)$ où $X = \{1, \dots, n\}$ représente les sommets du graphe et $E = \{(x_i, y_j), \dots\}$, $x_i, y_j \in X$ l'ensemble ordonné des arrêtes du graphe. Pour un graphe orienté, les notions de graphe simple et multiraphe sont les même que pour le cas non orienté.

Exemple (Graphes et leur représentation graphique) Soient G_1 et G_2 deux graphes, en voici unz représentation :

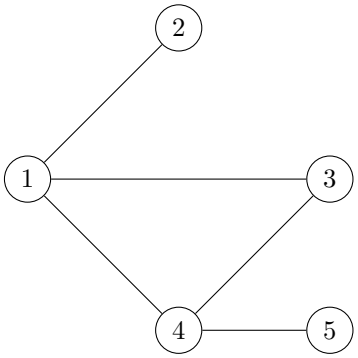


FIGURE 1.1 – Graphe non orienté

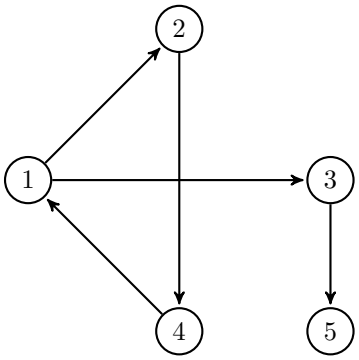


FIGURE 1.2 – Graphe orienté

Définition (**Planaire**) . Une graphe G est dit planaire s'il existe une représentation de G en deux dimensions telle qu'aucun de ses sommets ne se croisent.

Voisinnage et degré

Définition (**Voisinnage**) . Le voisinnage d'un sommet x de G est l'ensemble des sommets y de G tels qu'il existe une arrête entre x et y dans G. On le note $V(x)$.

Remarque Si x est dans le voisinnage de y, on dira que x est adlascent à y et inversement.

Définition (**Degré**) . Le degré d'un sommet x de G est le cardinal du voisinnage x. On le note $d(x)$. Un sommet de voisinnage nul est dit **isolé** et un sommet de voisinnage égal à 1 est dit **pendant**.

Définition (**Voisinnage entrant et sortant**) . Soit G un graphe orienté et x un sommet de G. On définit deux types de voisinnages :

- **Voisinnage entrant :** noté $V^-(X)$ est l'ensemble des prédécesseurs de x.
- **Voisinnage sortant :** noté $V^+(X)$ est l'ensemble des successeurs de x.

On définira comme précédemment le degré sortant et le degré entrant d'un sommet x.

Graphes Remarquables (non orientés)

Dans cette sous-section, nous ne parlerons que de graphes non-orientés.

Définition (**Graphe complet**) . On appelle graphe complet un graphe tel que pour tous sommets x et y de G il existe une arrête entre x et y dans G. Les graphes complets n sommets sont notés K_n

Remarque Autre définition de graphe complet et un petit peu d'histoire ne fera pas mal...

- On peut aussi définir un graphe complet comme étant un graphe dont tous ses sommets sont adjascents.
- *La notation K pourrait avoir deux origines, la première étant en hommage à Kazimierz Kuratowski, un éminent mathématiciens polonais ayant beaucoup contribué à la théorie des graphes. La seconde, plus simple, K proviendrait de sa traduction en Allemand komplett.*

Exemple Représentation des graphes complets K_4 et K_5 :

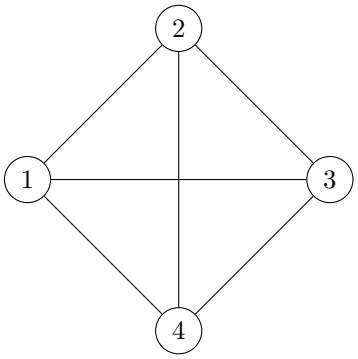


FIGURE 1.3 – Graphe complet K_4

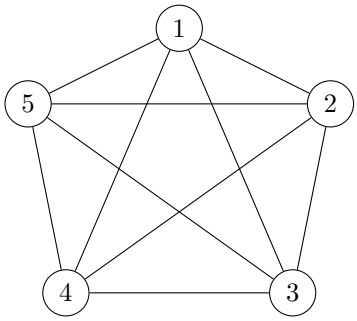


FIGURE 1.4 – Graphe complet K_5

Définition (**Bipartisme**) . Un graphe $G = (X, E)$ est dit biparti s'il existe une partition de X en ensembles X_1 et X_2 non vides et disjoints tels que pour toute arrête (x, y) de G, x et y soient des ensembles différentes.

Remarque G est dit k parti, s'il existe une partition en k ensembles de X vérifiant la définition ci-dessus.

Exemple Soit G un graphe à 5 sommets biparti, alors :

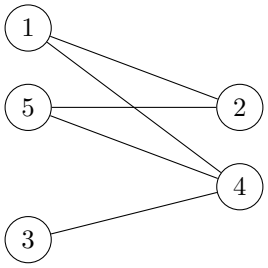


FIGURE 1.5 – Graphe biparti

Exemple (Graphe de Petersen) Sans doute l'un des graphes les plus connus en théorie des graphes, le graphe de Pertersen en hommage à Julius Petersen qui l'étudia en 1898, possédant 10 sommets et 15 arrêtes possède beaucoup de propriétés interressantes (notamment la connexité que nous verrons par la suite). Il est un contre-exemple pour beaucoup de propriétés et est très utile pour vérifier un algorithme en cours de développement ou une intuition.

Propriétés

Propriété () . Soit G un graphe non orienté simple à n sommets.

- Si G est complet, il possède $m = \frac{n(n-1)}{2}$ arrêtes.
- G vérifie donc toujours $m \leq \frac{1}{2}n(n-1)$
- $\sum_{x \in X} d(x)$ est le nombre d'extrémités d'arrêtes, c'est aussi deux fois le nombre d'arrêtes.
- Il y a un nombre pair de sommets de degré impairs.

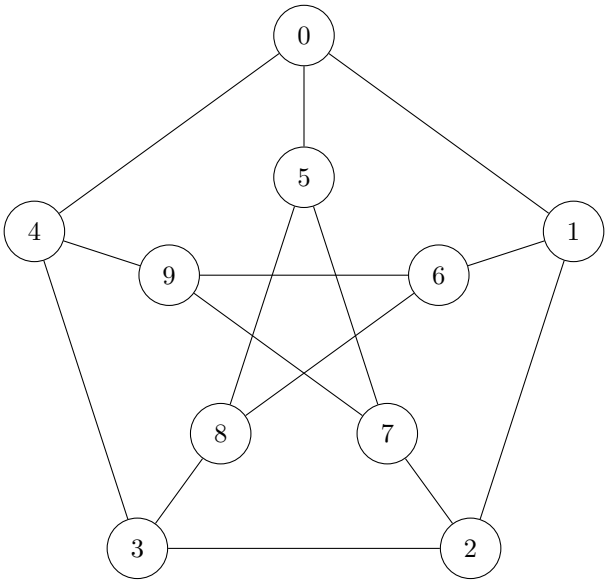


FIGURE 1.6 – Graphe de Petersen

Graphes partiels et sous-graphes

Définition (**Graphe partiel**) . Soit $G = (X, E)$ un graphe. Le graphe partiel $G' = (X, E')$ de G est tel que $E' \subset E$. Autrement dit, le graphe partiel d'un graphe G est le même graphe mais avec quelques arrêtes en moins.

Définition (**Sous-graphe**) . Soit $G = (X, E)$ un graphe. Le graphe $G' = (X', E')$ de G est tel que $X' \subset X$ et $E' = \{(x, y) : x \in X', y \in X', (x, y) \in E\}$. Autrement dit, le sous-graphe d'un graphe G est le même graphe mais avec quelques sommets en moins (et donc quelques arrêtes en moins aussi).

Définition (k-**clique**) . Soit G un graphe. On appelle k-clique, $k \leq n$, un sous-graphe complet de Gde taille k.

Chaines et Cycles d'un graphe non orienté

Définition (**Chaine**) . On appelle chaine de G de longueur n toute suite alternée de sommets et d'arrêtes de G telle que :

$$c = (x_0, a_1, x_1, \ldots, a_n, x_n), \text{ telle que } \forall i \in [\![1, n]\!], a_i = (x_{i-1}, x_i)$$

Ici, n représente le nombre d'arrêtes de la chaine. Dans le cas d'un graphe simple, on notera les chaines de la façon suivante :

$$c = (x_0, \ldots, x_n) \quad \text{ou} \quad c = x_0 - x_1 - \cdots - x_n$$

Définition (**Accessibilité**) . Soit $G = (X, E)$ un graphe. On a :

- Soient x et y deux sommets de G. On dit que y est accessible à partir de x s'il existe une chaine joignant x et y dans G.
- G est dit **connexe** ssi $\forall x \in X, \forall y \in X$, y est accessible à partir de x.
- L'accessibilité est un relation d'équivalence entre les sommets.
  Ses classes d'équivalences sont les composantes connexes de G.

Définition (**Chaîne Simple**) . Une chaine est **simple** si elle ne passe pas deux fois par le même arrête et elle est dite élémentaire si elle ne passe pas deux fois pas le même sommet. On remarquera facilement qu'une chaîne élémentaire est simple.

Définition (**Cycle**) . Un cycle de G est une chaine simple dont le départ et l'arrivée sont le même sommet. Un cycle est donc de la forme :

$$c = x_0 - x_1 - \cdots - x_{n-1} - x0$$

Théorème (**Propriétés des cycles et chaînes**) . Soit G un graphe.

- Toute chaine élémentaire a une longueur inférieure à $n - 1$
- Toute cycle élémentaire a une longueur inférieure à n
- De toute chaîne, on peut en extraire une chaîne élémentaire.

Chemins, circuits d'un graphe orienté et forte connexité

On utilise la même définition de chemin et circuit. Seulement, si le graphe est simple, il sera inutile de préciser les arrêtes par lesquelles on passe.

Définition (**Forte Connexité**) . Un graphe $G = (X, E)$ orienté est dit fortement connexe si pour tout somet x et y de G, y est accessible à partir de x.

Définition (**Arbre**) . On appelle arbre un graphe non orienté, connexe et sans cycle. Un graphe non orienté et sans cycle, i.e une union d'arbres et appelé **forêt**.

Théorème (**Caractérisation d'un arbre**) . Soit T un graphe à n sommets et m arrêtes. T est un arbre ssi :

T est sans cycle et $m = n - 1$
$\iff$ T est connexe et $m = n - 1$
$\iff$ T est sans cycle et maximal au sens des arrêtes
$\iff$ T est connexe et minimal au sens des arrêtes
$\iff$ Deux sommets quelconques de T sont reliés par un unique chemin.

1.1.2 Représentations d'un graphe

Représentation par liste d'arrêtes

Définition (**Liste d'arrêtes**) . On appelle liste d'arrêtes de G la liste des couples (x, y) avec $x \in X$ et $y \in X$ tels que $(x, y) \in E$

Exemple Représentations par liste d'arrêtes/d'arcs

- Le graphe orienté G_1 est représenté par la liste d'arcs :

$$[[1,5],[2,1],[2,4],[3,2],[4,3],[5,2],[5,4]]$$

- Le multigraphe non orienté G_2 est représente par la liste d'arrêtes :

$$[[1,2],[1,5],[1,5],[2,1],[2,4],[2,4],[2,3],[3,2],[3,3],[3,4],[4,2],[4,2],[4,3],[4,5],[5,1],[5,1],[5,4]]$$

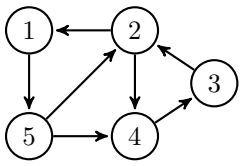


FIGURE 1.7 – Graphe Orienté G_1

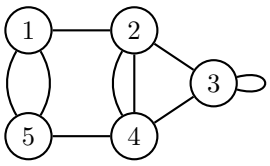


FIGURE 1.8 – Multigraphe non orienté G_2

Représentation Matricielle

Définition (**Matrice d'adjascence**) . On appelle matrice d'adjascence d'un graphe G la matrice $A = (a_{ij})$ telle que $\forall (i,j) \in X \times X$, on ait :

- Si G est non orienté, a_{ij} est égal à 1 si $(i,j) \in E$ et 0 sinon. La mrtice est donc symétrique.
- Si G est orienté, a_{ij} est égal à 1 si $(i,j) \in E$
- Dans le cas d'un multigraphe, le coefficient a_{ij} de la matrice d'adjascence de G représente le nombre d'arrêtes entre les sommets i et j de G. Les coefficients diagonaux de la matrice représentent donc les boucles de G.

Propriété () . Soit G un graphe non orienté, et A sa matrice d'adjascence.

- Puisque G est non orienté, alors $A \in \mathcal{S}_n(\mathbb{N})$ i.e A est symétrique.
- La somme des éléments de la ligne i est dégale à $d^-(i)$
- La somme des éléments de la colonne i esy égale à $d^+(i)$

Exemple (Matrices d'adjascences) La matrice d'adjascence de G_1 :

$$M_1 = \begin{pmatrix} 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \end{pmatrix}$$

La matrice d'adjascence de G_2 :

$$M_2 = \begin{pmatrix} 0 & 1 & 0 & 0 & 2 \\ 1 & 0 & 1 & 2 & 0 \\ 0 & 1 & 1 & 1 & 0 \\ 0 & 2 & 1 & 1 & 1 \\ 2 & 0 & 0 & 1 & 0 \end{pmatrix}$$

Remarque (Algèbre Linéaire) Comme dans toutes les représentations matricielles de concepts (que ce soit des applications ou des graphes), elles nous permettent d'invoquer facilement tous les résultats d'algèbre linéaires tels que la réduction, les noyaux, rangs, la composition...tout en préservant les propriétés des objets étudiés.

Propriété () . Soit $A^k, k \in \mathbb{N}$ où A est la matrice d'adjascence d'un graphe G. Alors le coefficient d'indice (x, y) de A^k est le nomdre de chemins de longueur k entre les sommets x et y.

Remarque Toutefois, la représentation matricielle d'un graphe n'est pas optimale pour parcourir ces derniers puisque, ainsi, les algorithmes de parcours auront forcément une complexité temporielle/spatiale en $\Theta(n^2)$.

Représentation par liste d'adjascence

Définition (**Liste d'adjascence**) . La liste d'adjascence d'un graphe G est un vecteur L indexé par X tel que pour tout sommet $x \in X$, $L[x]$ est la liste des successeurs de x dans G.

Exemple Reprenons les graphes G_1 et G_2 représentés plus haut. On a alors :

- $G_1 = \{1 : [5], 2 : [1, 4], 3 : [2], 4 : [3], 5 : [2, 4]\}$
- $G_1 = \{1 : [2, 5, 5], 2 : [1, 3, 4, 4], 3 : [2, 3, 4], 4 : [2, 2, 3, 5], 5 : [1, 1, 4]\}$

Remarque On remarque vite que cette représentation sera plus optinale. Premièrement, le parcours d'un graphe se fera en $\Theta(n)$ et pour chaque sommet, on obtiendra sa liste de successeurs en $\Theta(1)$, si on représente une liste d'adjascence sous forme de dictionnaire en Python.

Propriété () . Pour chaque graphe à n sommets et m arcs, l'espace mémoire utilisé est un $\Theta(n + m) = \Theta(\max\{n, m\})$.

1.1.3 Parcours d'un graphe

Parcours en profondeur (Depth First Search)

Le parcours en profondeur est à voir comme le parcours d'un chien fou dans un labyrinthe. Celui-ci va partir dans un couloir et à chaque intersection va aller sur le chemin le plus proche jusqu'à arriver à une impasse. A chaque impasse, il va revenir sur ses pas pour parcourir les autre chemins les plus proches. Le tout jusqu'à parcourir tout le labyrinthe.
Ainsi, cet algorithme de parcours se concevra de manière récurisive où chaque appel récurisif représentera l'envoi du chien fou dans un chemin (ici une arrête). D'où l'algorithme suivant :

```
1 Visite est initialise a l'ensemble vide
2 Profond (G,x) :
3     x est visite
4     {Traiter x en premiere visite}
5     Pour chaque voisin y de x faire :
6         Si y n'est pas visite, alors :
7             Profond(G,y)
8         {Sinon on detecte une revisite de y}
9     {Traiter x en derniere visite}
```

A partir du parcours en profondeur, on peut définir un ordre de parcours qui sera représenté par une liste de sommets. Nous avons donc l'ordre de parcours en première visite (que l'on mettra à jour ligne 3 de l'algorithme) et l'ordre de parcours en derniere visite (mis à jour ligne 9).
Pour un graphe G non connexe ou non fortement connexe, le parcours en profondeur lancé à partir d'un unique sommet ne permettra pas d'accéder à tous ses sommets. D'où le parcours dit généralisé où l'on itère sur tous les sommets de G et s'il n'est pas visité, on lance un parcours en profondeur à partir de celui-ci.
Ainsi, le parcours en profondeur généralisé d'un graphe G permet de définir une arborescence de parcours.

Définition (**Arborescence de parcours - profondeur**) . L'arborescence de parcours en profondeur de G à partir d'un sommet $x \in X$ est une arbre A enraciné en x, orienté, tel qu'il existe un arc (x, y) dans A ssi l'appel à profond(G,x) a engendré récurisivement un appel à profond(G,y).

A la fin d'un parcours en profondeur généralisé d'un graphe G, on obtient donc une forêt de visite contenant tous les sommets de G, i.e tous les sommets de G ont étés visités.

Parcours en largeur (Breadth First Search)

Le parcours en largeur est fondamentalement différent de celui en profondeur. Tout d'abord, nous pouvons reprendre la comparaison avec le parcours d'un labyrinthe par un chien. Ici, notre chien sera vieux et plus malin. Pour chaque intersection (sommet), il va se rappeler de tous les chemins à proximité (sommets voisins). Il se dirigera donc vers l'intersection (sommet) la plus ancienne qu'il ait retenue pour ensuite lister toutes ses voisinnes, et ainsi de suite. Pour ne pas tourner en rond, à chaque fois qu'il voit une intersection (sommet) voisinne, il vérifie qu'il ne l'aie pas visitée avant de la retenir. Le vieux chien d'arrête donc dès qu'il n'a plus d'intersection (sommet) à visiter en mémoire.
On peut donc implémenter cette algorithme de façons itérative où un ensemble représentera les sommets déjà visités et une file d'attente les sommets à visiter. Le premier sommet de file sera donc le prochain sommet à visiter.

```
1  Visite est initilise a l'ensemble vide
2  Largeur(G,x):
3      F = [X]
4      Visite[x] = vrai
5      Tant que F n'est pas vide, faire :
6          Considerer la tete y de F et l'enlever de F
7          {Traiter y}
8          Pour chaque successeur z de y, faire :
9              Si z n'est pas visite, alors
10                 z est visite
11                 ajouter z a la fin de la file F
```

De même que le parcours en profondeur, on peut définir un ordre de parcours, en première ou derniere visite. Comme le parcours en profondeur, parfois nous auront besoin de lancer un parcours généralisé à tout le graphe pour parcourir tous ses sommets.

Définition (**Arborescence de parcours - largeur**) . L'arborescence de parcours en largeur de G à partir du sommet x est un arbre A enraciné en x, orienté tel qu'il existe un arc (y, z) dans A ssi le traitement de y ajoute le sommet z dans la liste d'attente F.

La tableau Visite est intilisé en $\Theta(n)$. Le parcours est appelé exactement une seule fois pour chaque sommet x de G et a pour complexité $\Theta(1)$ pour le traitement de x et $\Theta(d(x))$ pour l'exploration des successeurs de x. La complexité des deux parcours est donc :

$$\begin{aligned} C(n) &= \Theta(n) + \sum_{x=1}^{n} (d(x) + \Theta(1)) \\ &= \Theta(N) + \sum_{x=1}^{n} d(x) \\ &= \Theta(n) + \Theta(m) \\ &= \Theta(\max\{n, m\}) \end{aligned}$$

Maintenant que l'on sait parcourir un graphe, on peut étudier ses propriétés plus facilement. Voici quelques exemples d'applications interressantes :

Classification des arcs

Une fois le parcours réalisé, on remarque que la forêt de parcours est composée de différents types d'arcs. Soit (x, y) un arc de G, il est dit...

- **Couvrant** ssi (x, y) est un arbre de la forêt de visite.
- **En avant** ssi il existe un chemin de x à y dans la forêt de visite de G.
- **En arrière** ssi il est chemin de y à x dans la forêt de visite.
- **Transverse** ssi ce n'est pas un arc ci-dessus.

La présence de cycle dans un graphe se manifestera par un arc arrière dans l'arbre de visite et un arc avant peut être vu comme un raccourci dans G. Notons que pour chaque parcours, certains arcs sont présents et d'autres non.

Existence de chemins et connexité

Les deux parcours d'un graphe à partir d'un sommet x nous permettent de trouver tous les sommets accessibles à partir de x. Donc on peut facilement adapter un algorithme pour déterminer s'il existe un chemin entre deux sommets x et y de G, et même en trouver.
Dans le cas d'un graphe **non orienté**, si le parcours à partir d'un sommet x atteint tous les autres sommets y de G, alors G est connexe et réciproquement.

"A partir d'ici, vous ne vous perdez plus dans les labyrinthes." Thierry Montaut

1.2 Modélisation et Graphes

1.2.1 Chemins et circuits Eulérien

Les 7 ponts de Königsberg

Le problème des 7 ponts de Königsberg consiste à savoir s'il est possible de déterminer une promenade passant par tous les ponts de la ville de Königsberg en passant une et une seule fois par chaque ponts de la ville et en revenant à son point de départ. Ce problème est sûrement le plus connu de l'histoire de la théorie des graphes et fut résolut par **Leonhard Euler** en 1735. Il explique que le problème n'est pas résoluble pour la ville de Königsberg et établit ainsi l'un des tout premiers théorèmes de théorie des graphes.
La démonstration mathématique du théorème d'Euler ne fut énoncée qu'en 1873 par Carl Hierholzer.
Des problèmes similaires existent tels que celui du facteur chinois qui cherche à effectuer sa tournée de distribution en passant une et une seule fois par chaque rue et en revenant à son point de départ (Mai-Ko-Kwan, 1962).

Définition (**Graphe Eulérien/Circuit Eulérien**) . Soit $G = (X, E)$ un graphe non orienté, un circuit eulérien dans G est un circuit passant une et une seule fois par chaque arrête et revenant au sommet de départ.
Un graphe est dit eulérien ssi il possède un circuit eulérien.

Théorème d'Euler

Théorème (**Théorème d'Euler**) . Soit G un graphe non orienté **connexe**.

- G admet un circuit eulérien ssi tous ses sommets sont de degré pair.
- G admet un circuit eulérien ssi tous ses sommets ont un degré pair seuf deux sommets a et b alors, tous les circuits eulériens de G on a/b comme sommet de départ et b/a comme sommet d'arrivée.

Algorithme de recherche de circuits eulérien

L'objectif de l'algorithme est de construire un circuit eulérien dans un graphe en considérant un graphe partiel du graphe initial. A chaque étape, on cherche un chemin maximal partant du premier sommet non saturé du chemin précédent et qui revient à ce sommet. Puisque il existe un nombre pair de sommets de degré impair il existe un tel sommet à chaque itération.
A la fin de l'algorithme on "recolle" tous les chemins bout à bout en un seul chemin simple non extensible passant par toutes les arrêtes, créant ainsi un circuit eulérien.
L'idée est de créer une "copie" du graphe et, lors de la création d'un chemin à une certaine étape, d'enlever les arrêtes concernées par le chemin pour éviter qu'elles ne soient réutilisées.

1.2.2 Problème de coloration

Les problèmes de colorations de graphes, en plus d'êtres difficiles à résoudre, sont, cependant, très facile à imaginer. L'exemple le plus typique est celui de la coloration d'une carte géographique. En essayant de colorier une carte de l'Europe d'une telle façon que deux pays limitrophes n'aient pas la même couleur, on s'apperçoit vite que cela peut être compliqué, surtout si on essaye d'utiliser le moins de couleurs possibles.
C'est exactement ce que modélisent les problèmes de coloration de graphes. On essaye d'attribuer à chaque sommet une couleur de telle sorte que tous ses voisins n'aient pas la même couleur que lui en utilisant le moins de couleurs possibles.

Définition (k-**coloration**) . Soit G un graphe non orienté. On dit que G admet une k-coloration s'il existe k couleurs différentes telles que deux sommets adjacents de G n'aient pas la même couleur.

Propriété (**Propriétés des colorations**) .

- Si G est k-parti alors G est k-colorable.
- Si G est un graphe complet de taille n alors G ne peut pas être colorié avec moins de n couleurs.
- Si G admet une k-clique, alors G ne peut pas être colorié avec moins de k couleurs.

Définition (**Nombre Chromatique**) . On appelle nombre chromatique $\gamma(G) = k$ d'un graphe G le plus petit entier positif tel que G admette une k-coloration.

Coloration Naïve

Premier algorithme de coloration de graphe, il est très facile à comprendre et à mettre en oeuvre mais est particulièrement gourmand en couleurs.
L'idée est de parcourir le graphe dans l'ordre naturel des sommets et d'attribuer à chaque sommet la plus petite couleur non attribuée à ses voisins.
Il faut donc une fonction permettant de parcourir le graphe et une autre déterminant la plus petite couleur non attribuée aux voisins d'un sommet. La coloration est représentée par un dictionnaire dont les clés sont les sommets du graphe et les valeurs, les couleurs attibuées aux sommets.
C'est un algorithme très peut coûteux d'une complexité en $\Theta(m)$ mais pas très optimal, puisque en fonction de l'ordre de visite des sommets, le nombre de couleurs utilisées peut varier. Il est très peu efficace pour de gros graphes.

Coloration Gloutonne

Prenez vos crayons de couleurs préférés et essayer de colorier un graphe. Vous alez d'abord colorier tous les sommets possibles avec un certaine couleurs. Puis une fois que vous ne pouvez plus colorier de sommets de cette couleur tel qu'aucun de ses voisins n'est déjà colorié vous prenez une autre couleur et refaites de même. Le tout jusqu'à ce que tous les commets soient coloriés.
C'est le principe de l'algorithme de coloration Gloutonne. A chaque étape, on choisit une couleur et on essaye de colorier le maximum de sommets non adjacents avec cette même couleur. Pour cela, nous allons considérer le noyau d'une graphe.

Définition (**Noyau**) . Soit $G = (X, E)$ un graphe. On appelle noyau de G un ensemble maximal de sommets non adjacents deux à deux.

On va donc déterminer tous les noyaux de G en détruisant petit à petit le graphe. Et, pour chaque noyau de trouvé, on colorie tous les sommets du noyau avec la même couleur.

1.2.3 Ordonnancement

Ici, un graphe représente un système de dépendances entre différentes tâches d'un même projet. Chaque sommet repésente une tâche à effectuer et une arrête d'un sommet x vers un sommet y indique que la tâche y doit attendre que la tâche x est terminée avant d'être commencée.
On va donc essayer d'établir un graphe d'ordonnancement des tâches visant à indiquer dans quel ordre les différentes tâche devront être traitées. Mais pour cela, il faut définir certaines conditions sur le graphe de dépendances.

Proposition Soit $G = (X, E)$ un graphe.

- Si G est sans circuit, alors tous ses sous-graphes sont sans circuits.
- Si G est sans circuit, le graphe inverse de G est sans circuit.
- G est sans circuit ssi tous ses chemins sont élémentaires.

Définition (**Sommets remarquables d'un graphe sans circuit**) . Soit $G = (X, E)$ un graphe orienté sans circuit. On définit :

- une **source** comme étant un sommet dont le degré entrant est nul.
- un **puit** comme étant un sommet dont le degré sortant est nul.

Théorème (**Fondamental de l'ordonnancement**) . Tou graphe sans circuit possède une source et un puit.

Tri Topologique - Ordonnancement séquentiel

Définition (**Tri Topologique**) . On appelle tri topologique d'un graphe orienté une numérotation des des sommets respectant l'ordre des arcs.

Théorème (**Condition d'existence d'un tri topologique**) . Soit $G = (X, E)$ un graphe orienté. G admet un tri topologique ssi il est sans circuit.

Le principe de l'algorithme de tri topologique est **itératif**. Il repose sur le fait que tout graphe sans circuit **possède une source**. On va donc, à chaque étape, chercher une source du graphe et l'insérer dans notre tri topologique. L'étape suivante, on **considère le sous-graphe** du graphe initial auquel on a enlevé la source et tous les arcs sortants.
Si on ne souhaite pas détruire le graphe, on va considérer les **degrés entrants** de chaque sommets dans un **dictionnaire** des degrés.

```
1  S:=[];T:=[];
2  Pour x de 1 a n faire
3      Degre[x]:=d^-(x) dans G;
4      Si Degre[x]=0 alors ajouter x a S;
5  {S contient alors toutes les sources de G}
6
7  Tant que S<>[] faire
8      x:=enleverTete(S);
9      ajouterFin(x,T)
10
11     {Mettre a jour les degres}
12     Pour chaque successeur y de x dans G faire
13         Degre[y]:=Degre[y]-1;
14         si Degre[y] = 0 alors ajouterFin(y,S)
```

L'algorithme repose donc sur un **parcours en largeur**. On utilise une **file** pour stocker les sommets de degré entrant nul (i.e les sources). A chaque itération, on décrémente tous les degrés entrants des fils de la source, puis on ajoute la source à la liste représentant le tri topologique.

Tri par Niveaux - Ordonnancement en parallèle

Ici, comparé à un tri topologique, on va construire un graphe d'ordonnancement en considérant que l'on peut effectuer plusieurs tâches en même temps.

Définition (**Tri par niveaux**) . Soit $G = (X, E)$ un graphe orienté, sans circuit. Un tri par niveaux de G est une partition de G en niveaux tels que pour chaque sommet d'un même niveaux, son exécution n'est pas dépendante d'un sommet du même niveau ou d'un niveau suivant.

L'algorithme est sensiblement le même que le tri topologique. Il suffit juste de l'adapter pour **traiter en même temps** toutes les sources de G. On définit alors **deux listes N1 et N2** représentant deux niveaux successifs. Le traitement des sources de N1 permet de déterminer les sources de N2.

```
N1:=[];T:=[];
Pour x de 1 a n faire
    Degre[x]:=d^-(x) dans G;
    Si Degre[x]=0 alors ajouter x a N1;
{S contient alors toutes les sources de G}

tant que N1<>[] faire
    ajouterFin(N1,T);
    N2:=[];
    pour tout x dans N1 faire
        {Mettre a jour les degres des successeurs de x et calcul}
        Pour chaque successeur y de x dans G faire
        Degre[y]:=Degre[y]-1;
        si Degre[y] = 0 alors ajouterFin(y,N2)
    N1:=N2;
```

1.2.4 Arbre couvrant de poids minimum

Ici, on considère un **graphe non orienté valué** pour lequel on va chercher un sous-arbre couvrant de poids minimal. Cela peut éventuellement modéliser le problème d'un électricien qui doit relier différentes pièces par un câble et cherche à utiliser le moins de câble électrique possible.

Définition (**Valuation**) . Soit $G = (X, E)$ un graphe non orienté. Une valuation est une fonction

$$p : E \longrightarrow \mathbb{R}$$

qui, à chaque arrête lui associe une valeur appelée poids.
Un graphe muni qu'un valuation est appelé graphe valué.

Pour **représenter** un graphe valué, nous allons toujours utiliser une liste d'adjacence. Mais nous allons rajouter une matrice de poids représentée par un dictionnaire dont les clés sont les arrêtes (couple) et la valeur, le poids de l'arrête correspondante.
On pourra aussi représenter un graphe valué par une liste d'arrêtes composée de triplet représentant les deux sommets de l'extrémité de l'arrête et le poids de l'arrête.

Définition (**Arbres couvrant**) . Soit $G = (X, E)$ un arbre graphe non orienté valué. Un arbre couvrant de G est un graphe partiel de G, $A = (X, E')$ qui soit un arbre.

Théorème (**Condition d'existence**) . Soit $G = (X, E)$ un graphe non orienté valué. G admet un arbre couvrant ssi il est connexe.

Algorithme de Kruskal

L'algorithme de Kruskal construit de **manière itérative** un arbre couvrant de poids minimal. Pour cela, on va considérer les arrêtes par ordre de poids croissant. A chaque itération, on va essayer d'ajouter **la plus petite arrête ne créant pas de cycle** à l'arbre couvrant.
On s'arrête lorsque l'on a ajouté $n - 1$ arrêtes.

Remarque (Condition d'arrêt et vérification)

- A chaque itération, il faut vérifier que le graphe construit de possède pas de cycle, il faut donc effectuer un **parcours en largeur**.
- Si à une itération, on ne trouve pas d'arrête qui convienne, c'est que le graphe n'était initialement pas connexe.

Algorithme de Prim

L'algorithme de Prim cherche, contrairement à Kruskal, à **conserver la connexité à moindre coût**.
On va donc considérer deux ensembles :

- C : l'ensemble des arrêtes de T (arbre couvrant)
- M : le complémentaire de C dans X

A chaque étape, on va donc chercher une arrête **joignant un sommet de C à un sommet de M dans G.** Un arrête joignant deux composantes connexe ne pouvant pas créer de cycle, le graphe T reste bien un arbre.
On s'arrête lorsque **T contient tous les sommets de G**.

1.2.5 Plus courts chemins dans un graphe valué

On considère ici un **graphe orienté valué**. L'objectif est de trouver un chemins entre deux sommets x et y dans G de poids minimal. On retrouve le même problème pour le routage de paquets dans un réseau. En effet, lors de l'établissement des tables de routage, on va chercher le chemin le plus court (au sens de la durée de transfert) entre deux noeuds.

Définition (**Cycle Absorbant**) . Dans le cas d'un graphe orienté, valué à **poids négatifs**, on appelle cycle absorbant un cycle de coût total négatif.

Remarque Le passage par un cycle absorbant dans un tel graphe fait diminuer le coût du chemin de poids minimal. On comprend alors vite qu'un graphe avec un circuit absorbant ne possède pas de chemin de poids minimal.

Théorème (**Condition d'existence**) . Soit $G = (X, E)$ un grapge orienté valué. Soient x et y deux sommets de G. Alors il existe un chemin de poids minimal entre x et y ssi

- y est accessible à partir de x
- G ne possède pas de circuit absorbant

Algorithme de Dijkstra

L'algorithme de Dijkstra 1 se base sur un **parcours en largeur** du graphe. On cherche tous les plus courts chemins d'un sommet s vers les autres sommets du graphe. Pour cela on dispose d'un **dictionnaire des poids**, et d'un **dictionnaire des pères**.
A l'initialisation, on commence par parcourir tous les sommets du graphe à la recherche d'arrêtes partant de s. On va construire pas à pas un plus court chemin vers chaque sommet. A chaque étape, on considère tous les "nouveaux" chemins vers les sommets du voisinnage du sommet traité. Si on trouve un chemin plus court, on remplace le chemin actuel et on remet le sommet dans la file. Dans le cas contraire, on ne fait rien.

```
1  M:={s};
2  {Initialisation}
3  Pour i de 1 a n faire
4      Si (s,i) est une arete alors
5          D[i]:=cout(s,i);
6          P[i]:=s;
7          ajouter(i,M);
8      Sinon D[i]:=infini;
9
10 {Algorithme}
```

```
Tant que M<>{} faire
    x:=enleveTete(M);
    Pour tout successeur y de x faire
        d:=D[x]+cout(x,y);
        si d<D[y] alors
            D[y]:=d;
            P[y]:=x;
            ajouter(y,M)
```

L'algorithme de Dijkstra possède quand même quelques inconvénients. Un même sommet peut se retrouver plusieurs fois dans la file et on visite donc tous ses voisins plusieurs fois. Dans le cas **d'arrêtes à poids tous positifs**, on peut éviter tous ces parcours inutiles choisissant à chaque étape le sommet dont le chemin depuis s est de coût minimal.

Dijkstra Opt

A chaque fois que l'on récupérer un sommet dans la file, on va **récupérer le sommet de poids minimal.** Pour cela, on va **garder la file triée selon le poids des sommets**. Il faut donc implémenter un fonction `insere` qui ajoute un sommet dans la file en fonction de son degré d'accessibilité.

```
{Initialisations}
M:={};
Pour i de 1 a n faire
    Si (s,i) est une arete alors
        D[i]:=cout(s,i);
        P[i]:=s;
        ajouter(i,M);
    Sinon D[i]:=infini;

{Algorithme}
Tant que M<>{} faire
    x:=choisir_min(M,d);
    enlever(x,M);
    Pour tout successeur y de x faire
        Si y est dans M alors
            d:=D[x]+cout(x,y);
            si d<D[y] alors
                D[y]:=d;
                P[y]:=x;
                ajouter(y,M)
```

La **complexité** de l'algorithme de Dijkstra est donc en

$$\boxed{\Theta(n^2) + \Theta(m) = \Theta(n^2)}$$

Algorithme de Floyd

Cet algorithme permet de calculer **tous les plus courts chemins** de tous les sommets vers tous les autres. Il a un **complexité** en $\Theta(n^3)$, donc il reste très efficace.
L'algorithme repose sur une **idée itérative**. On commence donc par calculer tous les plus courts chemins de i vers j (deux voisins) sans intermédiaires. Puis on calcule tous les plus courts chemins avec **un intermédiaire.**

...

Arrivé à la n-ième itération, on a tous les plus courts chemins du graphe.

Par **récurence**, pour passer de l'étape k à l'étape k+1 :

$$\forall (i,j) \in \{1,n\}^2, \quad d = pcc[(i,k)] + pcc[(k,j)]$$
$$\text{si } d < pcc[(i,j)] :$$
$$pcc[(i,j)] = d$$

On utilise une **matrice de poids** (pcc) et un **dictionnaire des pères** P.

Algorithme de Bellman (poids quelconques)

Ici, on cherche les plus courts chemins d'un graphe orienté valué avec potentiellement des poids négatifs.
L'idée est que si on connaît tous les plus courts chemins d'un sommet s vers tous les prédécesseurs d'un sommet y de G, alors, le plus court chemin de s vers y est :

$$min(\{d_i + p_i, \quad i \in \{i,n\}\}) =: d_{i0} + pi0$$

et le prédécesseur de y dans le plus court chemin sera x_{i0}.
A **l'initialisation**, on connaît tous les plus courts chemins de s vers lui-même. On considère donc G' le **sous-graphe** de G constitué des sommets dont one ne connaît pas encore de plus courts chemins. G' est un sous-graphe d'un graphe sans cycle donc il admet toujours des **sources**. Une source de G' est un sommets de G' dont on connaît tous les prédécesseurs. On peut donc calculer son plus courts chemin comme vu précédement et il sort de G'.
On va construire un **dictionnaire des distances** `Dist` et un **dictionnaire des pères** dans le plus court chemin `Pred`. Pour l'algorithme, on utilise un dictionnaire représentant le nombre de prédécesseurs inconnus d'un sommet.

```
# Sort de G'
Pour x dans G[y] faire :
    Deg[x] -= 1
    si Deg[x] == 0 :
        ajouter(x,S)
```

```
# calcul du pcc vers y
{Soit H la liste des predecesseurs dans G}
Pour x dans H[y] faire :
    si Dist[x] + P[(x,y)] < D[y] faire :
        D[y] = Dist[x] + P[(x,y)]
        Pred[y] = x
```

Pour représenter le graphe on a donc besoin :

- La liste d'adjacence de G.
- Le dictionnaire des pères.
- La matrice de poids représentée par un dictionnaire.

L'algorithme se base donc sur un **parcours en largeur**, d'où :

```
{Initialisation}
...
{Algorithme}
Tant que S<>[] faire :
    y = tete de S
    # calcul pcc vers y
    # y sort de G'
```

Cet algorithme a donc un **complexité** en :

$$\Theta(m+n)$$

Chapitre 2

Mots et Langages

Contents

Si on connaît plusieurs langages de programmation, on remarque que chaque langage, ou plutôt chaque paradigme de langage est spécialisé pour la résolution d'une carégorie dde problèmes. On pourrait se demander s'il est possible de créer un langage permettant de résoudre tous les problèmes. Pour cela, il nous faudrait d'abord être capable de définir formellement un langage, des mots, etc...

2.1 Alphabets et Mots

2.1.1 Premières Définitions

Commençons tout d'abord par redéfinir correctement la notion d'alphabet et de mot.

Définition (**Alphabet**) . Un alphabet est simplement un ensemble fini, noté Σ. On nomme "lettre" ou "symbole" les éléments d'un alphabet.

Exemple Quelques exemples d'alphabets :

- $\Sigma = \{a, b\}$
- $\Sigma = \{a, b, \ldots, \%, \$\}$

Définition (**Mot**) . Un mot est une suite finie de lettres d'un alphabet.

Proposition Le mot vide est noté ε. On note l'ensemble des mots d'un alphabet Σ^*.

Définition (**Longueur d'un mot**) . On note $|w|$ la longueur d'un mot $w \in \Sigma^*$ qui correspond au nombre de lettres, avec répétition du mot w.

Définition (**Egalité de mots**) . On dit que deux mots $u, v \in \Sigma^*$ son égaux ssi :

- $|v| = |u|$
- $\forall i \in [\![1, |v|]\!], \quad u[i] = v[i]$ où $u[i]$ est la ième lettre du mot u.

Deux mot sont égaux ssi ils sont de même longueur et sont composés des mêmes lettres dans le même ordre.

2.1.2 Opérations sur les mots

Sur les mots, on ne définit qu'une seule opération, la **concaténation**.

Définition (**Concaténation**) . Soient $u, v \in \Sigma^*$ deux mots définis sur un même alphabet. On appelle la concaténation l'application :

$$\begin{cases} \Sigma^* \times \Sigma^* \longrightarrow \Sigma^* \\ (u, v) \longmapsto w = u.v \end{cases}$$

Elle est définie telle que $\forall u, v \in \Sigma^*$ de longueur $n, p \in \mathbb{N}$, on ait :

- $|u.v| = |u| + |v| = n + p$
- $\forall i \in [\![1, n]\!] u.v[i] = u[i]$ et $\forall i \in [\![1, p]\!], u.v[n + i] = v[i]$

On parlera identiquement de concaténation ou de produit.

Remarque Cette définition reprend bien la caractérisation de deux mots égaux.

Proposition (Propriétés de la concaténation) La concaténation est une application :

- **Associative :** $\forall u, v, w \in \Sigma^*, \quad w.(u.v) = (w.u).v$
- **Pas commutative** pour un alphabet de plus d'une lettre.
- admet pour **élément neutre** le mot vide ε.

2.1.3 Puissance d'un mot

Une fois la concaténation définie pour un mot, on peut alors parler de puissance de mot. Définissons celle-ci par récurrence.

Définition (**Puissance d'un mot**) . Soit Σ un alphabet et $u \in \Sigma^*$, on a :

- $u^0 = \varepsilon$
- $u^1 = u$
- $\forall n \in \mathbb{N}, u^{n+1} = u^n.u$

Exemple $(ba)^3 = bababa$

Proposition Soient $u \in \Sigma^*$ on peut appliquer les règles "connues" des puissances d'où :

$$\forall n, p \in \mathbb{N}, \quad u^{n+p} = u^n.u^p = u^p.u^n$$

On remarque que l'on peut effectuer des simplifications sur les égalités de mots.

Propriété (**Simplifications**) . L'ensemble Σ^* est simplifiable à gauche et à droite.

- $\forall u, v, w \in \Sigma^*, \quad u.w = v.w \Longrightarrow u = v$
- $\forall u, v, w \in \Sigma^*, \quad w.u = w.v \Longrightarrow u = v$

Ici, pas besoin d'inverse, la démonstration repose sur la définition de l'égalité entre deux mots.

2.2 Relations d'Ordre

Dans l'alphabet dit "classique" on possède un ordre lexicographique des mots permettant de les classer en fonction de leurs lettres et de la position de leurs lettres. Ici, nous allons définir deux types de relations d'ordre sur les mots.
Commençons par rappeler la définition de relation d'ordre.

Définition (**Relation d'Ordre**) . Soit $\lhd$ une relation sur un ensemble E. On dit que $\lhd$ est une **relation d'ordre** ssi pour tout $x, y, z \in E$, $\lhd$ est :

- **Réflexive :** $x \lhd x$
- **Anti-Symétrique :** $x \lhd y$ et $y \lhd x \Longrightarrow x = y$
- **Transitive :** $x \lhd z$ et $z \lhd y \Longrightarrow x \lhd y$

2.2.1 Ordre Préfixe

Naturellement, on munitΣ^* d'un ordre préfixe permettant de classer les mots en fonction de leur préfixe. Cette relation peut être vue comme un forme d'inclusion de mots.

Définition (**Ordre Préfixe**) . Soient $u, w \in \Sigma^*$, on définit la relation d'ordre préfixe $\sqsubseteq$ telle que :

$$\boxed{u \sqsubseteq w \iff \exists v \in \Sigma^*, w = u.v}$$

Autrement dit, u est un préfixe de w ssi il existe un mot v tel que w soit composé de la concaténation de u et v.

Remarque L'ordre préfixe ne nécessite pas de relation d'ordre directement sur l'alphabet Σ.

Propriété (**Ordre Préfixe et égalité**) . Soient $u, v \in \Sigma^*$ on a :

$$u \sqsubseteq v \text{ et } v \sqsubseteq u \Longrightarrow u = v$$

Démonstration Soient $u, v \in \Sigma^*$ tels que $\exists x, y \in \Sigma^*$ tels que

$$u = v.x \quad \text{et} \quad v = u.y$$

On a alors que :

$$\begin{cases} v = v.y.x \\ yx = \varepsilon \end{cases} \Longrightarrow \begin{cases} y = \varepsilon \\ x = \varepsilon \end{cases} \Longrightarrow u = v$$

Remarque Attention : l'ordre préfixe est une relation d'ordre partielle. Autrement dit, tous les éléments d'un même alphabet de sont pas comparables.

2.2.2 Ordre Lexicographique

Définition (**Ordre Lexicographique**) . Soit Σ un alphabet que l'on muni d'une relation d'ordre $\leqslant$. L'ordre lexicographique $\leqslant$ est une relation d'ordre totale sur Σ^*.

Remarque Ici, nous avons bien besoin de définir au préalable un ordre sur notre alphabet Σ.

Propriété (**Compatibilité**) . L'ordre lexicographique est compatible avec l'ordre préfixe. Plus formellement,

$$\forall u, v \in \Sigma^*, \quad u \sqsubseteq v \Longrightarrow u \leqslant v$$

2.3 Langage

Maintenant que nous sommes au clair sur la définition de lettre et de mot, on peut enfin définir l'objet principal de ce chapitre, les langages.

2.3.1 Definition

Définition (**Langage**) . Soit Σ un alphabet, on appelle langage sur Σ toute partie de Σ^*.

Remarque L'ensemble de tous les langages d'un alphabet Σ est donc $\mathcal{P}(\Sigma^*)$, l'ensemble de toutes les parties de Σ^*.

Définition (**Complémentaire**) . Soit L un langage sur Σ. On définit le complémentaire de L dans Σ^* le langage :

$$\overline{L} = \{w,\ w \notin L\}$$

2.3.2 Opérations

De même que pour les alphabets et les mots, on peut définir des opérations sur les langages.

Définition (**Opérations sur les langages**) . Soit Σ un alphabet et $L_1, L_2 \subseteq \Sigma^*$ deux langages de Σ. On définit 4 principales opérations sur des langages :

- **Somme :** notée $+$, la somme de deux langages d'apparente à l'union des ensembles.

  $$\boxed{L_1 + L_2 = \{w,\ w \in L_1 \text{ ou } w \in L_2\}}$$

  C'est une opération :
  - Commutative
  - Associative
  - dont $\emptyset$ est le neutre.
- **Intersection :** de même que pour les ensembles :

  $$\boxed{L_1 \cap L_2 = \{w,\ w \in L_1 \text{ et } w \in L_2\}}$$

  C'est une opération :
  - Commutative
  - Associative
  - dont Σ^* est le neutre

- **Différence :** comme les ensembles, on définit la différence de langages :

$$\boxed{L_1/L_2 = \{w,\ w \in L_1 \text{ et } w \notin L_2\} = L_1 \cap \overline{L_2}}$$

- **Produit de concaténation :** de même que pour les mots, on peut généraliser le produit de concaténation aux langages :

$$\boxed{L_1.L_2 = \{u.v,\ u \in L_1, v \in L_2\}}$$

  C'est une opération :
  - Associative
  - Distributive par rapport à l'union
  - D'élément neutre $\{\varepsilon\}$.

Définition (**Puissance de langage**) . Soit L un langage, on définit **par récurrence** la puissance de L par :

- $L^0 = \{\varepsilon\}$
- $L^1 = L$
- $\forall n \in \mathbb{N}^*,\ L^n = L^{n-1}.L$

Une fois définies des opérations "simples" sur les langages, on peut en définir des plus complexes, permettant de "générer" un langage infini à partir d'un langage fini ou infini.

Définition (**Langage plus et étoile**) . Soit L un langage sur un alphabet Σ. On définit le langage plus de L comme le langage :

$$L^+ = L^1 + L^2 + \dots$$

De même le langage étoile de L est défini par :

$$L^* = \{\varepsilon\} + L^1 + L^2 + \dots$$

2.3.3 Propriétés

Voyons quelques propriétés des langages...

Proposition Soient L_1 et L_2 deux langages sur un alphabet Σ, on a les propriétés suivantes :

- $\forall p \in \mathbb{N}$, on a :
$$(L_1)^p.(L_2)^p \subseteq (L_1)^*.(L_2)^*$$
- L'opération étoile est idempotente :
$$(L^*)^* = L^*$$
- $L^* = \{\varepsilon\} + L^+$
- $\varepsilon \in L \iff L^+ = L^*$

2.3.4 Expressions Régulières

Lorsque l'on manipule des langages infinis, il serait appréciable d'avoir une expression pratique pour un langage permettant de directement voir la forme des mots qu'il contient. On définit ainsi les expressions régulières.

Définition (**Expression Régulière**) . On définit récurisevement une expression régulière sur un alphabet Σ :

- ε est une expression régulière.
- $\forall w \in \Sigma$ est une expression régulière.
- Si E est un expression régulière alors (E) l'est aussi.
- Si E_1 et E_2 sont des expressions régulières, alors $E_1 + E_2$ l'est aussi.
- Si E_1 et E_2 sont des expressions régulières, alors $E_1.E_2$ l'est aussi.
- Si E est une expression régulière, alors E^* l'est aussi.

Exemple Voyons quelques exemples d'expressions régulières sur un alphabet $\Sigma = \{a, b\}$:

$$a^*b, \quad (a+b)^*, \quad (a+b)^*ba(a+b)^*$$

Définition (**Langage Régulier**) . On dit qu'un langage est régulier si il peut s'écrire sous la forme d'une expression régulière.

Il sera donc préférable de travailler avec des langages réguliers.

2.4 Langage Décidable

L'objectif de ce cours est bien entendu de comprendre comment fonctionne un compilateur, pour pouvoir en créer un par nous même. Pour rappel, on doit d'abord bien comprendre les notions de langage et de mot pour pouvoir ensuite déterminer si un ensemble de mots est syntaxiquement corrects lors de la compilation.
Lors de la compilation, il faut d'abord commencer par savoir si un mot traité appartient au langage défini ou pas. Pour des langages finis, l'opération n'est pas compliquée, pour chaque mot il suffit de vérifier si il appartient à un ensemble fini. Pour des langages infinis, l'opération semble plus complexe, il va falloir trouver une manière systématique et efficace de définir si un mot appartient au langage ou pas.
On appelle ce genre de problème un problème de **décision**.

Définition (**Langage Décidable**) . Un langage L est dit décidable si il existe un algorithme permettant de dire si un mot w appartient ou pas au langage L.

Théorème (**Nombre de Langages Décidables**) . Il existe un nombre fini de langages décidables.

Autrement dit, il existe un nombre infini de langages non décidables...

Chapitre 3

Automates (AFD, AFN, AFε)

Contents

Comme abordé dans le chapitre précédent, on cherche une méthode pratique et efficace pour déterminer si un mot appartient à un langage ou pas. On veut donc un modèle qui soit d'une part très pratique mathématiquement pour nous permettre de démontrer des choses dessus mais aussi facilemet implémentable algorithmiquement.
Alerte Spoiler : de solides connaissances en théorie des graphes seront plus qu'utiles...

3.1 Automates fini déterministes

3.1.1 Définition et représentation

Définition (**Automate fini déterministe**) . Un Automate Fini Déterministe est un quintuplet :

$$\boxed{\mathcal{A} = (\Sigma, Q, T, q_0, A)}$$

où :

- Σ est un alphabet

- Q est un ensemble fini d'états (souvent une partie finie de $\mathbb{N}$)
- $T : Q \times \Sigma \longrightarrow Q$ est une application qui, à un état et une lettre associe un autre état.
- q_0 un état initial
- $A \subseteq Q$ les états acceptants

On représentera ainsi un automate fini déterministe de plusieurs façons en fonction de son utilisation :

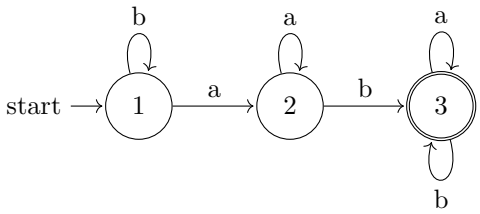
- **Mathématique :** $\mathcal{A} = (\Sigma, Q, T, q_0, A)$
- **Table de Transition :** Elle va permettre de trouver rapidement les différents types d'états.
- **Sagitale :** Sous forme de graphe
- **En Python :** Nous représenterons les automates finis déterministes sous la forme de quintuplet aussi.

Regardons en détail ces différentes représentations au travers d'un exemple.

Exemple Soit $\mathcal{A} = (\Sigma, Q, T, q_0, A)$ un automate fini déterministe.
Mathématiquement nous avons :

- $Q = \{1, 2, 3\}$
- $\Sigma = \{a, b\}$
- $q_0 = 1$
- $A = \{3\}$

La représentation **sagitale** de notre automate sera :



On représente de façon doublement cerclée les états acceptants. Les états sont les sommets du graphe. Les arcs valués sont les antécédants/images de la fonction T.
Enfin, la **table de transition** de l'automate est représentée par le tableau suivant :

Q/Σ	a	b
1	2	3
2	2	3
3	3	3

La première ligne présente les lettres de l'alphabet et la première colonne les différents états. Pour chaque état, le tableau donne l'état obtenu en fonction de la lettre suivante lue.

3.1.2 Mot et Langage Automatique

Définition (**Lecture d'un mot**) . Soit Σ un dictionnaire, $l \in \Sigma$ et $\mathcal{A}$ un automate. On lit la lettre l en dérivant d'un état $q \in Q$ vers un état $q' \in Q$ et si $T(q, l) = q'$. On notera

la lecture d'un mot de longueur p par la lecture successive de ses lettres :

$$q_1 \xrightarrow{l_1} q_2 \ \dots \ q_{p-1} \xrightarrow{l_p} q_p$$

Concrètement, pour la lecture d'un mot, on va partir de l'état initial et en fonction des valeurs de l'état courant et de la lettre lue, on va "bouger" d'un état (sommet) à un autre.

Définition (**Mot refusé**) . Un mot $w \in \Sigma^*$ est dit refusé par un automate $\mathcal{A}$ si sa lecture à partir de l'état initial se termine sur un état refusant ou ne se termine pas. Dans le cas contraire, w est dit accepté.

Définition (**Langage d'un Automate**) . Le langage d'un automate $\mathcal{A}$ est l'ensemble des mots acceptés par l'automate. On le note $L(\mathcal{A})$.

On parle de **langage automatique** si il est reconnaissable par un automate. Deux automates sont dits **équivalents** si ils reconnaissent le même langage.

Définition (**Automate Complet**) . Un automate $\mathcal{A}$ est dit complet si

$$\forall i \in Q, \forall l \in \Sigma, \quad T(i,l) \text{ est défini}$$

Autrement dit, un automate est dit complet si pour toute lettre et pour tout état fixés, il est possible de changer d'état dans l'automate.

Définition (**Puit**) . Un état $q \in Q$ est un puit ssi

$$\forall l \in \Sigma, \quad T(q,l) = q$$

Un puit est un état duquel on ne peut sortir.

On définit aussi la notion de piège comme un puit refusant (i.e un puit dont l'état est refusant). Puisque la notion de complémentaire existe pour les langages et que les automates semblent très étroitement liés aux langages, on peut se demander si un automate peut admettre un complémentaire...
Soit $\mathcal{A}$ un automate associé à un langage L. On cherche $\mathcal{A}' = \overline{\mathcal{A}}$.

$$w \in \overline{\mathcal{A}} \iff w \notin L \iff w \notin L(\mathcal{A})$$

Il semble falloir que $\mathcal{A}$ soit complet. Si c'est le cas, on pourrait inverser $\mathcal{A}$ en inversant les sommets accpetants/refusants.

Proposition Tout automate fini peut être complété par des puits refusants en un automate complet.

Théorème (**Complémentarité**) . L'ensemble des automates est stable par complémentarité.

3.2 Automates fini non déterministes

Dans la section précédente nous avons vu un modèle très efficace pour vérifier l'appartenance d'un mot à un langage. En plus d'être facilement représentable en mémoire (i.e Python), il est

facile à utiliser à la main et hérite de toute la théorie des graphes vue précédement ce qui en fait un très beau modèle mathématiquement parlant.
Malgré tout cela, nous ne savons pas comment, à partir de plusieurs langages simples, constituer un automate reconnaissant la somme de ces langages. Nous n'avons pas défini de somme/union d'automate et celles-ci semblent assez difficiles vu la rigidité de notre modèle.
Nous allons donc construire un modèle d'automate, appelé non déterministe, nous permettant de faire ces opérations d'union (que nous appelerons juxtaposition). Elles nous permettrons de construire des automates complexes à partir de somme de langages.

3.2.1 Généralités

Définition (**AFN**) . Un automate fini non déterministe $\mathcal{A}$ est un quintuplet :

$$\mathcal{A} := (Q, \Sigma, T, I, A)$$

tel que :

- Q est l'ensemble des états de l'automate
- Σ est un alphabet
- $T : Q \times \Sigma \longrightarrow \mathcal{P}(Q)$ est une application
- $I \subseteq Q$ est l'ensemble des états initiaux
- $A \subseteq Q$ est l'ensemble des états acceptants.

Remarque Plusieurs remarques concernant ce nouveau modèle. Premièrement, on remarque que l'on peut maintenant définir des transitions multiples entre les états de l'automate. Deuxièmement, il existe plusieurs états initiaux.

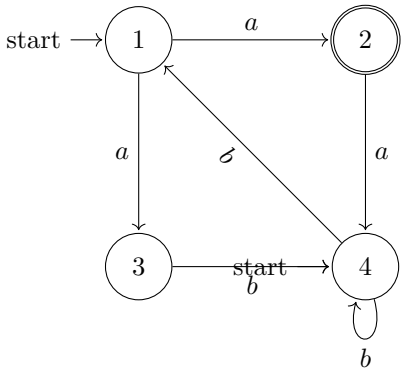
Définition (**Arbre de lecture**) . Soient $w \in \Sigma^*$, L un langage sur Σ et $\mathcal{A}$ un automate reconnaissant L. L'arbre de lecture de w par $\mathcal{A}$ est l'arbre résultat du parcours de $\mathcal{A}$ en fonction des lettres de w.
Autrement dit dans l'arbre de lecture G de w, un noeud a est le fils d'un noeud b si il existe une lettre l de w telle que $T(b, l) = a$. Une feuille de cet arbre est un état acceptant ou refusant de l'automate.

Définition (**Lecture acceptante**) . Soient $w \in \Sigma^*$, L un langage sur Σ et $\mathcal{A}$ un automate reconnaissant L. Une lecture de w par $\mathcal{A}$ est dite acceptante si il existe un chemin d'un état initial vers un état acceptant dans l'arbre de lecture de w par $\mathcal{A}$.

Proposition On peut dire plusieurs choses de la lecture d'un mot $w \in \Sigma^*$ par un automate $\mathcal{A}$:

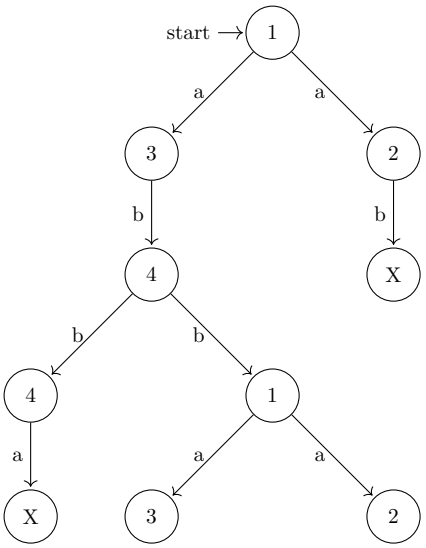
- Si l'automate possède plusieurs états initiaux, la lecture produit un arbre de lecture pour chaque état initial, nous auront donc une forêt d'arbres de lecture.
- Une lecture sera donc acceptante ssi il existe un arbre de la forêt dont au moins une des feuilles est un état acceptant.

Exemple Voyons tout cela sur un exemple. Soit $\mathcal{A} := (Q, \Sigma, T, I, A)$ un automate fini non déterministe $\mathcal{A}$ sur l'alphabet $\Sigma : \{a, b\}$. Représentons notre automate sous forme de graphe orienté valué et sa table de transition :



T	a	b
1	{2,3}	X
②	4	X
3	X	4
4	X	{1,4}

C'est un automate non déterministe puisqu'il contient deux transitions mutliples et deux états initiaux.
Posons $w := abba$, déterminons si ce mot appartient au langage $L(\mathcal{A})$. Nous allons construire un seul arbre permettant d'avoir une condition suffisante de validation du mot.



L'arbre de lecture du mot *abba* contient un état acceptant comme feuille.

Autrement dit, il existe un chemin menant d'un état initial à un état acceptant dans la forêt de lecture de *abba*. Donc $abba \in L(\mathcal{A})$.

3.2.2 Juxtaposition et construction d'un AFD

Juxtaposition

Rappelons la problématique principale du chapitre. On cherche un modèle dérivant des AFD nous permettant de définir des opérations dessus et qui puisse être convertit algorithmiquement vers un AFD pour construire des automates d'un langage complexe à partir de langages plus simple.
Autrement dit, on veut pouvoir appliquer l'opération de somme de langages sur les automates fini déterministes.

Définition (**Juxtaposition d'AFN**) . Soient L_1 et L_2 deux langages reconnus par deux automates $\mathcal{A}_1$ et $\mathcal{A}_2$. Le langage $L_1 + L_2$ est reconnu par la **juxtaposition disjointe** de $\mathcal{A}_1$ et $\mathcal{A}_2$.

Théorème (**Langages automatiques et stabilité**) . L'ensemble des langages automatiques est stable par somme.

On peut maintenant, à partir de deux langages automatiques, définir le nouveau langage résultant de la somme des deux qui sera lui aussi automatique. Il suffit de faire la juxtaposition disjointe des deux automates de départ.

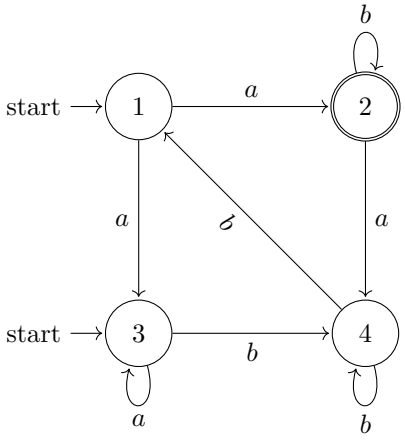
Déterminisation

Théorème (**Existence et équivalence**) . Pour tout automate fini non déterministe, il existe un automate déterministe équivalent.

Ce théorème est peut être un peu obscur mais permet de dire qu'il est toujours possible de passer d'un automate fini non déterministe (obtenu par exemple par juxtaposition) à un automate fini déterministe qui reconnaisse le même langage. En tout cas, il nous dit qu'il en existe un...
L'intérêt de vouloir repasser chez les automates fini déterministes vient du fait que la lecture d'un mot par un AFN est de complexité exponentielle alors que la lecture d'un mot par un AFD est polynômiale... Lors de la vérification syntaxique de très long mots pour des langages très complexes, cela fait une différence cruciale pour la compilation.
Ce processus est appelé **déterminisation** d'un AFN.

Proposition Soit $\mathcal{A} = (Q, \Sigma, T, I, A)$ un AFN. On cherche à construire un AFD $\mathcal{A}'$ équivalent à $\mathcal{A}$. L'idée est de raisonner sur l'application $T : Q \times \Sigma \longrightarrow \mathbb{Q}$. Dans un AFN, cette application n'est pas injective, on va donc poser une nouvelle application dans l'espace quotient de $Q \times \Sigma$ par le noyau de T. Nous obtiendrons donc une application injective et donc un AFD.

Exemple (Déterminisation) Soit $\mathcal{A}$ l'automate défini sur l'alphabet $\Sigma = \{a, b\}$ non déterministe et sa table de transition suivants :



T	a	b
1	{2, 3}	X
②	4	2
3	3	4
4	X	{1, 4}

Déterminisons cet automate. Pour cela, nous allons renommer tous les états de l'automate en prenant en compte les ensembles. L'algorithme consiste donc à construire la table de transition du nouvel automate. Pour chaque itération (i.e ajout d'une ligne dans la table), on effectue un parcours en largeur du nouvel automate pour "découvrir" de nouveau état. On créé ainsi un "automate des parties".

3.3 Automates fini à ε-transitions

Définissons un nouveau type d'automates non déterministes. Les automates non déterministes à ε-transition. Il diffèrent des premiers puisque l'on va permettre le changement d'état sans lecture

	a	b
$I = \{1, 3\}$	II	III
$II = \{2, 3\}$	IV	V
$III = \{4\}$	-	VI
$IV = \{3, 4\}$	VII	VI
$V = \{2, 4\}$	III	$VIII$
$VI = \{1, 4\}$	II	VI
$VII = \{3\}$	VII	III
$VIII = \{1, 2, 4\}$	IX	$VIII$
$IX = \{2, 3, 4\}$	IV	$VIII$

FIGURE 3.1 – Table de l'AFD

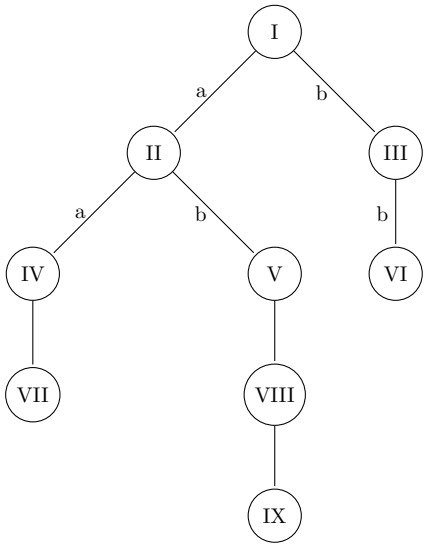


FIGURE 3.2 – Automate des parties

de lettres lors de la lecture d'un mot. Pour cela, nous allons définir une transition ε. Cela peut se voir comme une transition via le mot vide.

3.3.1 Généralités

Définition (**Automate Fini à ε-transitions (AFNε)**) . Un automate fini à ε-transitions est un quintuplet :

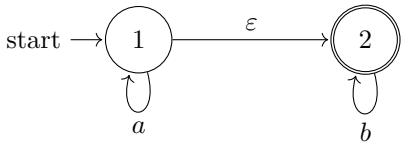
$$\mathcal{A} = (Q, \Sigma, T, I, A)$$

définit de la même façon que les automates précédents mais où :

$$T : Q \times \Sigma \cup \{\varepsilon\} \longrightarrow \mathcal{P}(Q)$$

Ici, le changement spontané d'état sans lecture de lettre sera donc caractérisé par une nouvelle entrée dans la table de transition ε.

Exemple Soit le langage $L := a^*b^*$. Un automate reconnaissant ce langage peut être écrit avec une ε-transition. Ecrivons aussi sa table de transition :



T	a	b	ε
1	1	-	2
②	-	2	-

Lors de la lecture d'un mot, les transitions peuvent être très aléatoires en fonction du nombre d'ε-transitions possibles de l'état courant. Un tel automate est donc hautement non déterministe.

Définition (**Lecture d'un mot**) . Soit $w = l_1 l_2 \dots l_n$ un mot sur Σ. Soit $w' = a_1 a_2 \dots a_p$ le mot w ε-complété (rembourré par des ε) tel que :

- $p \geqslant n$
- $\forall i \in [\![1, n]\!],\ a_i \in \{l_1, \dots, l_n\} \cup \{\varepsilon\}$

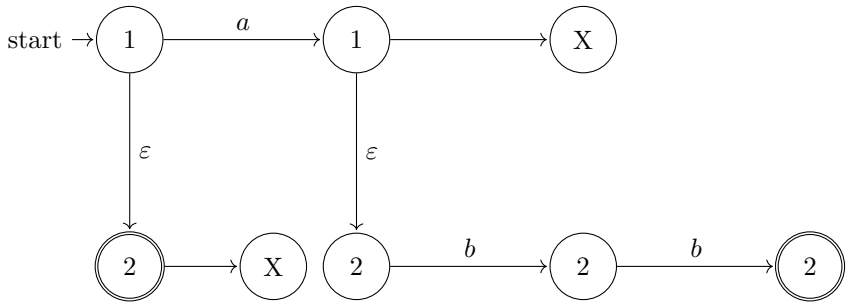
- $l_1 l_2 \dots l_n = a_1 a_2 \dots a_p$ (du point de vue du produit de concaténation)

Une lecture du mot w par $\mathcal{A}$ est une lecture par $\mathcal{A}$ de n'importe quel w', un ε-complété de w.

Remarque Tout comme pour les automates précédants, un mot appartient au langage d'un automate ssi la lecture de ce mot par celui-ci se finit sur au moins un état acceptant de l'automate.

La lecture d'un mot par un AFNε conduira donc à la construction d'une forêt de lecture de ce mot par l'automate.

Exemple (Lecture d'un mot) Soit l'automate fini non déterministe à ε-transitions précédent. Soit le mot *abb*. Construisons la forêt de lecture de *abb* par $\mathcal{A}$.



Il existe un chemin d'un racine vers une feuille acceptante dans la forêt de lecture :

$$1 \xrightarrow{a} 1 \xrightarrow{\varepsilon} 2 \xrightarrow{b} 2 \xrightarrow{b} 2$$

Donc par définition, $abb \in L(\mathcal{A})$.

3.3.2 Déterministation

Dans le chapitre précédent, un théorème nous permet de dire que pour tout automate fini non déterministe, il existe un automate fini déterministe équivalent. Ainsi, à chaque fois que l'on considère un AFNε, on est sûr qu'il existe un AFD équivalent.
Pour la déterminisation d'un AFNε, nous allons avoir besoin d'une définition supplémentaire...

Définition (**Clôture**) . Soit $\mathcal{A} = (Q, \Sigma, T, I, A)$ un AFNε. Pour tout $q \in Q$, on appelle clôture de q l'ensemble des états accessibles à partir de q sans lecture de lettre lors de la lecture d'un mot dans l'automate.
Autrement dit, la clôture de q est l'ensemble des états accessibles depuis q dans le sous-graphe de $\mathcal{A}$ restreint aux ε-transitions.
On note $cl(q)$ la clôture de q.

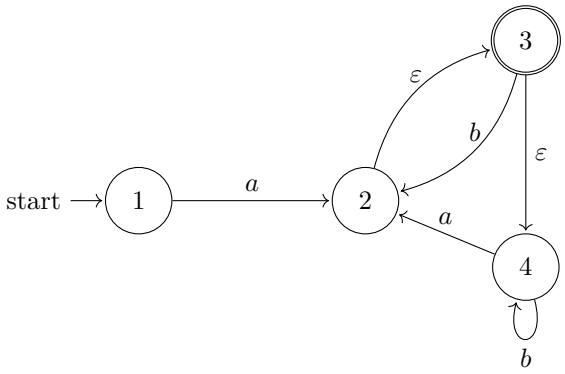
L'idée de la déterministation d'un AFNε est, non plus de regrouper des états, mais d'étendre les transitions de l'automate à tous les états accessibles par ε-transitions.
De plus, si un état acceptant est accessible uniquement par ε-transtion depuis un état, alors celui-ci hérite du caractère acceptant de l'état atteint.

Proposition (Algorithme de Déterministation) Soit $\mathcal{A} = (Q, \Sigma, T, I, A)$ un AFNε. On construit l'automate fini déterministe $\mathcal{A}'$ équivalent à $\mathcal{A}$ en :

1. Calculer les clôtures de $\mathcal{A}$
2. Héritage : Tous les états dont la clôture contient un état acceptant sont acceptants.

3. Calculer les transitions étendues
4. Déterminisation de $\mathcal{A}'$ par l'automate des parties (voir chap précédent)

Exemple (Déterminisation) Soit $\mathcal{A}$ l'AFNε suivant :



T	a	b	ε
1	2	-	-
2	-	-	3
③	-	2	4
4	2	4	-

1. **Calcul des clôtures et héritage**

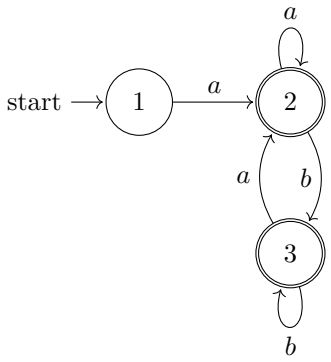
T	a	b	ε	$cl(q)$
1	2	-	-	$\{1\}$
②	-	-	3	$\{2,3,4\}$
③	-	2	4	$\{3,4\}$
4	2	4	-	$\{4\}$

2. **Calcul des transitions étendues**

$\widetilde{T}$	a	b	$cl(q)$
1	2	-	
②	2	$\{2,4\}$	$\{2,3,4\}$
③	2	$\{2,4\}$	$\{3,4\}$
4	2	4	

3. **Déterminisation par l'automate des parties**

	a	b
I = $\{1\}$	II	X
II = $\{2\}$	II	III
III = $\{2,4\}$	II	III



La déterminisation d'un AFNε nous permet donc de passer de la lecture d'un mot de complexité exponentielle (voire infinie) à un automate permettant de lire tous les mots avec une complexité linéaire.

==========

3.4 Opérations entre automates

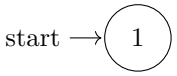
L'objectif de cette section est de nous permettre de construire un automate reconnaissant le langage $L_1 \ T \ L_2$ où T est une opération entre langages à partir des automates $\mathcal{A}_1$ et $\mathcal{A}_1$ reconnaissant respectivement les langages L_1 et L_2. Pour cela, nous aurons besoin de définir formellement les opérations entre langages et les processus nous permettant de "passer aux automates".

3.4.1 Langages Elémentaires

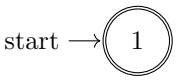
Toutes les opérations entre langages (par extension entre automates) se feront à partir de langages de élémentaires. Ces langages seront reconnus par des automates fixés, que nous connaissons d'avance. Nous en choisissons un nombre fini pour pouvoir les stocker en mémoire. Ils vont représenter les briques de base nous permettant de construire des automates plus complexes par la suite.

Proposition (Langages Elémentaires) Soit $\Sigma = \{a, b\}$ un alphabet. On définit les langages élémentaires de Σ comme :

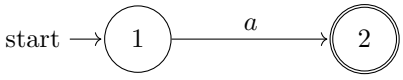
- $L = \emptyset$ reconnu par l'automate :



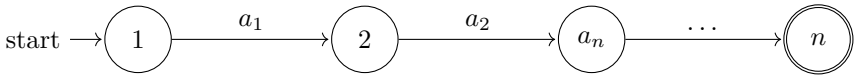
- $L = \{\varepsilon\}$ reconnu par l'automate :



- $L = \{a\}$ reconnu par le langage :



- $L = \{a_1 \dots a_n\}$ reconnu par le langage :



3.4.2 Automate Complémentaire

A partir d'un langage, on peut définir son langage complémentaire. De même, on peut définir l'automate complémantaire reconnaissant ce langage.

Définition (**Automate Complémentaire**) . Soit L un langage reconnu un automate $\mathcal{A} = (Q, \Sigma, T, q_0, A)$ **complet**. L'automate reconnaissant le langage complémentaire de L est $\overline{\mathcal{A}} = (Q, \Sigma, T, q_0, Q \backslash A)$.

Remarque Attention, pouvoir "passer au complémentaire" pour un automate, il faut que celuici soit complet.

3.4.3 Somme d'automates

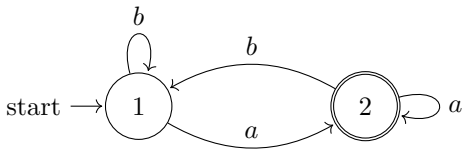
Définition (**Somme d'automates**) . Soient L_1 et L_2 deux langages respectivement reconnus par $\mathcal{A}_1$ et $\mathcal{A}_2$. L'automate reconnaissant L_1+L_2 est l'automate fini non déterministe construit par **l'union disjointe** de $\mathcal{A}_1$ et $\mathcal{A}_2$. On l'appelle **automate somme** des automates $\mathcal{A}_1$ et $\mathcal{A}_2$.

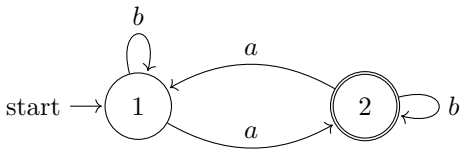
Cet automate n'est pas déterministe puisqu'il contient deux états initiaux mais que l'on peut déterminiser.

Exemple Soient les langages suivants sur $\Sigma = \{a, b\}$:

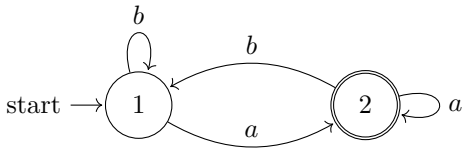
$$L_1 = \{\text{mots de } \Sigma \text{ terminant par } a\}$$
$$L_2 = \{\text{mots de } \Sigma\text{contenant un nombre pair de } a\}$$

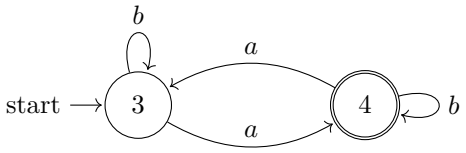
Ces langages sont reconnus par les automates suivants :





On construit l'automate $\mathcal{A}$ comme la juxtaposition disjointe des deux automates :





Que l'on doit ensuite déterminiser...

Ces définitions de langages élémentaires nous permettent de savoir que tout langage réduit à un mot est automatique. On en déduit le théorème suivant :

Théorème (**Langage Fini**) . Tout langage fini est automatique.

3.4.4 Intersection d'Automates

Proposition Soient L_1 et L_2 reconnus par les automates fini déterministes suivants $\mathcal{A}_1$ et $\mathcal{A}_2$. On a alors :

$$\overline{L_1 \cap L_2} = \overline{L_1} + \overline{L_2}$$

On peut donc en conclure que :

$$\boxed{L_1 \cap L_2 = \overline{\overline{L_1} + \overline{L_2}}}$$

Ainsi, l'intersection de deux automates peut être construite par somme et complémentarité.

En pratique nous utuliseront plutôt l'automate des couples :

Définition (**Automate des Couples**) . Soient $\mathcal{A}_1 = \{Q_1, \Sigma, T_1, Q_0, A_1\}$ reconnaissant le langage L_1 et $\mathcal{A}_2 = \{Q_2, \Sigma, T_2, q'_0, A_2\}$ reconnaissant le langage L_2. On définit l'automate des couples :

$$A = \{Q_1 \times Q_2, \Sigma, T, (q_0, q'_0), A_1 \times A_2\}$$

reconnaissant le langage $L_1 \cap L_2$ où

$$\forall i \in \Sigma, \forall q_1, q_2 \in Q_1, \forall q'_1, q'_2 \in Q_2 \text{ tels que } q_2 = T_1(q_1, j) \text{ et } q'_2 = T(q'_1, i)$$

$$\text{alors } T'((q_1, q_1'), i) = (q_2, q_2')$$

Proposition Si $\mathcal{A}_1$ possède $n \in \mathbb{N}$ états et que $\mathcal{A}_2$ possède $p \in \mathbb{N}$ états, alors $\mathcal{A}_1 \cap \mathcal{A}_2$ possède $n \times p$ états. Pour de gros automates, cette méthode peut donc engendrer des très gros. Même si la façon de les construire est assez simple et ressemble beaucoup à la déterminisation.
L'automate obtenu est, de plus, déterministe.

3.4.5 Différence d'Automates

Proposition Soient $\mathcal{A}_1$ et $\mathcal{A}_2$ deux automates reconnaissant respectivement les langages L_1 et L_2. Pour reconnaître le langages $L_1 \backslash L_2$, on peut simplement construire l'automate reconnaissant :

$$L_1 \backslash L_2 = L_1 \cap \overline{L_2}$$

3.4.6 Langages Automatiques

Essayons maintenant de déduire des conditions nécessaire pour qu'un langage soit automatique. D'après ce que l'on a vu grâce aux opérations, on peut déjà énoncer la proposition suivante :

Proposition Les langages réguliers sont tous automatiques.

Proposition que nous élargirons plus tard grâce au *théorème de Kleene*.

Théorème (**Pompage (faible)**) . Soit L un langage. Supposons L automatique. Soit $N \in \mathbb{N}$, alors pour tout $w \in L$ tel que $|w| \leqslant N$, w admet une décomposition de la forme :

$$\exists w_1, w_2, w_3 \in L, w = w_1 w_2 w_3 \quad \text{avec } w_2 \neq \varepsilon$$

telle que cette décomposition soit gonflable :

$$\text{i.e } \forall k \in \mathbb{N}, w_1 w_2^k w_3 \in L$$

Ce théorème n'est pas idéal pour montrer qu'un langage est automatique. En revanche, sa contraposé de la forme :

$$\boxed{\text{Non Pompable} \implies \text{Non Automatique}}$$

Est en pratique très utilisée pour montrer qu'un langage n'est pas automatique.

Proposition Ainsi, soit L un langage. Pour montrer que L n'est pas pompable, on montrera que pour tout $N \in \mathbb{N}$, il existe $w \in L$ tel que $|w| \geqslant N$ qui admette une décomposition de la forme :

$$w = w_1 w_2 w_3 \qquad \text{avec } w_2 \neq \varepsilon$$

telle que :

$$\exists k \in \mathbb{N}, w_1 w_2^k w_3 \notin L$$

Chapitre 4

Lemme d'Arden et Systèmes d'équations aux langages

Contents

Ce chapitre est dédié à l'étude du *Lemme d'Arden*. Ce lemme nous permettra de construire un automate déterministe à partir d'une expression régulière mais aussi à déterminer le langage d'un automate déterministe. Nous finirons par énoncer le théorème de Kleene caractérisant les langages automatiques. Il nous permettra de faire le lien entre langages automatiques et langages réguliers.

4.1 Lemme d'Arden

Le lemme d'Arden est présenté en 1961 par Dean N. Arden. En voici une de ces formes :

Lemme (Arden) Soient A et B deux langages. Le langage $L = A^*B$ est le plus petit langage (pour l'inclusion ensembliste) solution de l'équation :

$$(E) : X = (AX) \cup B$$

De plus, si A ne contient pas le mot vide ε, A^*B est l'unique solution de cette équation. Que l'on peut dériver en deux formes différentes : pour tout langage $L \in \Sigma^*$ et $a, b \in \Sigma$ tel que $\varepsilon \notin a$ on a :

Version Gauche :

$$\boxed{L = aL + b \iff L = a^*b}$$

Version Droite :

$$\boxed{L = La + b \iff L = ba^*}$$

4.2 Applications

4.2.1 Langage d'un automate fini

Maintenant que nous savons bien manipuler les automates fini déterministes et non déterministes, il serait utile, à partir d'un automate de pouvoir déterminer le langage qu'il reconnaît.

Pour cela nous avons besoin de définir les systèmes d'équations au langages.

Définition (**Langage d'arrivée**) . Soit $\mathcal{A} = \{Q, \Sigma, T, q_0, A\}$ un automate fini. On définit le langage d'arrivée à l'état $q \in Q$, noté L_q l'ensemble des mots de Σ^* dont la lecture par $\mathcal{A}$ débute par q_0 et finit en q.

Proposition Soit $\mathcal{A} = \{Q, \Sigma, T, q_0, A\}$ un automate fini. Soit $\{L_q \mid i \in A\}$ l'ensemble des langages d'arrivés aux états acceptants de l'automate $\mathcal{A}$. On a alors l'égalité suivante :

$$L(\mathcal{A}) = \sum_{q \in A} L_q$$

Autrement dit, le langage de $\mathcal{A}$ est la somme de tous ses langages d'arrivé aux états acceptants.

Définition (**Système d'équations aux langages**) . Soient un ensemble $X_1, \ldots, X_n$ de langages sur un même alphabet Σ. Un système d'équations aux langages est un ensemble d'équations de la forme :

$$X_i = \sum_{j=1}^{n} a_{ij} X_j + b_j \quad \forall i \in [\![1, n]\!]$$

où $\forall i, j \in [\![1, n]\!], a_{ij} \in \Sigma, b_i \in \Sigma^*$

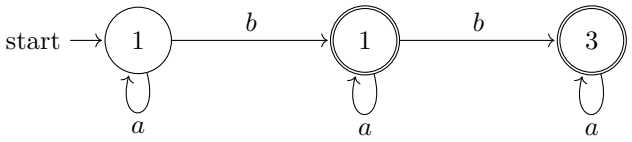
Proposition Soit $\mathcal{A} = \{Q, \Sigma, T, q_0, A\}$ un automate fini. À partir des définitions, on peut donc représenter le langage d'un automate par un système d'équations aux langages. Elles associent à chaque L_q une équation de langages dérivant de l'automate.

Ainsi si un état q a des transitions vers d'autres états selon les lettres $a \in \Sigma$ et si q est un état acceptant alors l'équation associée à L_q est de la forme

$$\begin{cases} L_q = \sum_{(q,a,q') \in T} a L_{q'} + \{\varepsilon\} & \text{si q est un état initial} \\ L_q = \sum_{(q,a,q') \in T} a L_{q'} & \text{sinon} \end{cases}$$

Il nous faut maintenant être capable de résoudre ces équations pour déterminer les L_q et ainsi le langage de l'automate. Équations que l'on peut maintenant résoudre grâce au lemme d'Arden.

Exemple Déterminons le langage de l'automate suivant :



Les langages d'arrivés vérifient donc le système d'équations aux langages suivant :

$$\begin{cases} L_1 = L_1 a + \varepsilon \\ L_2 = L_1 b + L_2 a \\ L_3 = L_2 b + L_3 b \end{cases}$$

Que l'on résout progressivement grâce au Lemme d'Arden :

$$\begin{cases} L_1 = \varepsilon a^* = a^* \\ L_2 = L_2 a + a^* b \\ L_3 = L_3 a + L_2 b \end{cases} \iff \begin{cases} L_1 = a^* \\ L_2 = a^* b a^* \\ L_3 = L_3 a + a^* b a^* \end{cases} \iff \begin{cases} L_1 = a^* \\ L_2 = a^* b a^* \\ L_3 = a^* b a^* b a^* \end{cases}$$

D'où :

$$\begin{aligned} L(\mathcal{A}) &= a^*ba^* + a^*ba^*ba^* \\ &= a^*ba^*(\varepsilon + ba^*) \end{aligned}$$

Donc $\mathcal{A}$ reconnaît les mots qui contiennent 1 ou 2 b.

Cet algorithme de résolution est très utile en terme pratique mais il apporte aussi une précisions supplémentaire sur les langages automatiques. En effet, à partir d'un automate (i.e langage automatique), on peut écrire le langage reconnu comme une expression régulière. D'où le résultat suivant :

Propriété (**Langages Automatiques**) . Tout langage automatique peut être décrit par une expression régulière.

On en déduit donc le théorème de Kleene établissant définitivement le lien entre les langages réguliers et automatique :

Théorème (**Kleene**) . Les langages réguliers sont les mêmes que les langages automatiques.

4.2.2 Construction d'un automate déterministe

Le lemme d'Arden et le théorème de Kleene nous permettent maintetant d'affirmer que toute expression régulière peut être exprimée comme un automate déterministe équivalent. Nous allons ici voir, autour de plusieurs exemples, comment construire un tel automate. Pour cela, nous aurons besoin de la version gauche du lemme d'Arden.

Remarque (Principe) L'idée de l'algorithme est de partir d'une expression régulière E et de la développer sous la forme :

$$E = x_1L_1 + x_2L_2 + \cdots + x_nL_n$$

où $x_1, \ldots, x_n \in \Sigma$ et $L_1, \ldots, L_n \in \Sigma^*$. Ensuite, on appelique récursivement cette opération sur tout les L_i pour obtenir une forme propice à appliquer Arden :

$$L_i = A^*B \iff L_i = AL_i + B$$

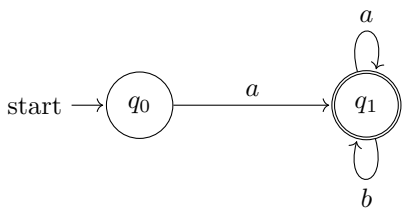
Enfin, chaque L_i représente un état de l'automate et tous les L_i composés d'un ε correspondent à un état acceptant.

Exemple Soit $\Sigma = \{a, b\}$ et le langage L décrit par l'expression régulière $a.(a+b)^*$. Déterminons l'automate de $\mathcal{A}$ reconnaîssant L :

$$\begin{aligned} L_0 &= a.(a+b)^* \\ &= a.L_1 \end{aligned}$$

$$\begin{aligned} L_1 &= (a+b)^*.\lambda \\ &= (a+b)L_1 + \lambda \\ &= a.L_1 + bL_1 + \lambda \end{aligned}$$

On a donc l'automate suivant :



Exemple Soit $\Sigma = \{a, b\}$ et le langage L décrit par l'expression régulière $ab^* + (a + b)c^*$. Déterminons l'automate de $\mathcal{A}$ reconnaîssant L :

$$\begin{aligned} L_0 &= ab^* + (a+b)c^* \\ &= ab^* + ac^* + bc^* \\ &= a(b^* + c^*) + bc^* \\ &= aL_1 + bL_2 \end{aligned}$$

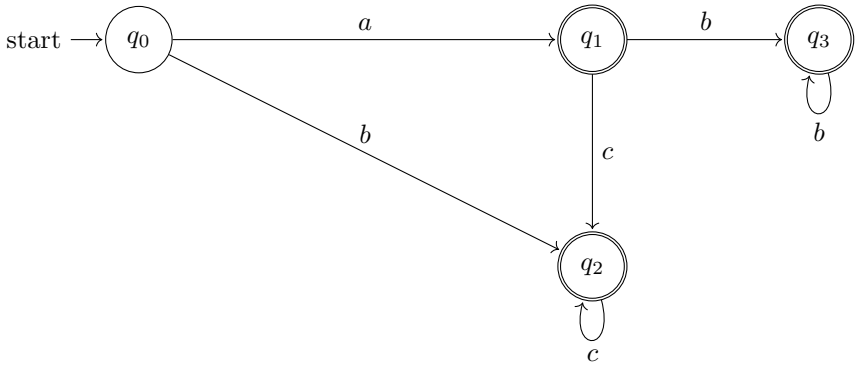
$$\begin{aligned} L_1 &= bb^* + \lambda + cc^* + \lambda \\ &= bL_3 + c_L 2 \end{aligned}$$

$$L_2 = c^* \iff L_2 = cL_2 + \lambda$$

$$L_3 = b^* if L_3 = bL_3 + \lambda$$

$$\begin{cases} L_0 = aL_1 + bL_2 \\ L_1 = bL_3 + cL_2 + \lambda \\ L_2 = cL_2 + \lambda \\ L_3 = bL_3 + \lambda \end{cases}$$

On a donc l'automate suivant :



Chapitre 5

Langages Algébriques et Automates à Piles

Contents

Précédement, nous avons vu que les langages automatiques sont de très bonnes propriétés. Ils sont stables pour la plupart des opérations définies sur les langages. De plus, le théorème de Kleene nous a permis d'établir le lien direct entre langages automatiques et langages réguliers. La simplicité de la représentation sagitale des automates permet de les implémenter facilement algorithmiquement. Il sont, de plus, facile à manipuler à la main et permettent de rapidement "voir" les langages reconnus.
Cependant, cette simplicité a un certain coût, celui de ne pas pouvoir reconnaître des langages "compliqués", notamment ceux où il faut "compter" les lettres. Un automate fini déterministe ne peut donc pas reconnaître le langage composé d'autant de a que de b. On va donc chercher à indroduire une nouvelle théorie, celle des **grammaires formelles** qui nous permettra de reconnaître de tels langages.

5.1 Grammaires Formelles

5.1.1 Contexte et définition

Les grammaires formelles ont initialement été développés par des linguistes, notamment Noam Chomsky en 1955. L'objectif était de développer un méthode systématique de traduction entre différentes langues. Ils se sont alors heurtés au problème des mêmes mots qui admettent plusieurs traductions en fonction du contexte de la phrase et n'ont pas pu aboutir leur oeuvre.

Or en informatique, pour l'étude de la syntaxe de langages de programmation, le problème du contexte ne se pose pas. Leur théorie a donc été récupérée pour la vérification syntaxique.

L'idée est donc de représenter un langage **récursivement** par un ensemble de règles de production composées d'un axiome de départ et de différentes règles de productions ou de réécriture. Nous utilisons souvent cette approche pour la gestion de types en Caml en définissant tes types récurisivement.

Définition (**Grammaire Formelle**) . Une grammaire formelle est un quadruplet

$$G = (\Sigma, V, S, P)$$

où

- Σ est un **alphabet terminal** dont chaque élément ne peut se réécrire plus simplement.
- V est **l'alphabet auxiliaire** (disjoint de Σ) composé de variables, qui ne peuvent pas non plus se réécrire.
- S est la variable de départ, appelé axiome.
- P est un ensemble de règles dites **de production** ou de réécriture du type

$$X \longrightarrow w \quad X \in V \text{ et } w \in (V \cup \Sigma)^*$$

Par convention, on notera toujours les variables en majuscule et les éléments terminaux en minuscules. En pratique, on regroupera plusieurs réécritures d'une même variable sur la même ligne en les séparant par des barres verticales de la forme :

$$X \longrightarrow w_1|w_2|\ldots|w_p \iff \begin{cases} X \longrightarrow w_1 \\ X \longrightarrow w_2 \\ \vdots \\ W \longrightarrow w_p \end{cases}$$

5.1.2 Réécriture d'un mot et langages algébriques

L'obectif d'une grammaire formelle, vous l'aure compris, est de réécrire un mot récursivement jusqu'à arriver à des éléments terminaux.

Définition (**Réécriture d'un mot**) . Soit $G = (\Sigma, V, S, P)$ une grammaire formelle. Soient $u, v \in (V \cup \Sigma)^*$ deux mots. On dit que u **peut se réécrire en** v **en une étape** et on note :

$$u \vdash v$$

si il existe des décompositions de u et v en

$$u = u_1 X u_2 \text{ et } v = u_1 w u_2$$

et que G contient la règle de production :

$$X \longrightarrow w$$

Plus généralement, on peut définir la réécriture en plusieurs étapes de la forme :

Définition (**Réécriture (2)**) . Soit $G = (\Sigma, V, S, P)$ une grammaire formelle. Soient $u, v \in (V \cup \Sigma)^*$ deux mots. On dit que u **peut se réécrire en** v ou que v **dérive en** u en un nombre quelconque de fois si il existe $u_1, \ldots, u_p \in (V \cup \Sigma)^*$ tels que

$$u \vdash u_1 \vdash u_2 \vdash \cdots \vdash u_p \vdash v$$

On note alors

$$u \overset{*}{\vdash} v$$

On peut maintetant définir les langages engendrés par des grammaires formelles et les langages algébriques, le coeur de ce chapitre.

Définition (**Langage Engendré**) . La **langage engendré** par une grammaire formelle $G = (\Sigma, V, S, P)$ est l'ensemble des mots de Σ^* qui dérivent de l'axiome S en un nombre quelconque d'étapes. On le note, comme pour les automates, en $L(G)$.

Définition (**Langage Algébrique**) . Un langage engendré par une grammaire est appelé **langage algébrique**.

5.1.3 Arbre de dérivation d'un mot

Définition (**Arbre de dérivation d'un mot**) . Soit $G = (\Sigma, V, S, P)$ une grammaire formelle. On appelle l'arbre de dérivation de $w \in Sigma^*$ l'arbre dont :

- La racine est S
- Tous les sommets intérieurs appartiennent à V
- Toutes les feuilles appartiennent à $\Sigma \cup \{\varepsilon\}$
- Si un sommet intérieur X a pour fils $X_1, \ldots, X_p$ alors la règle

$$X \longrightarrow X_1 | \ldots | X_p \in P$$

- Le mot obtenu en visitant les feuilles de l'arbre par un parcours profondeur préfixe de l'arbre est un mot de $L(G)$

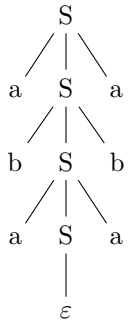
Définition (**Grammaire Ambiguë**) . Soit G une grammaire. On dit que G est ambiguë s'il existe un mot $w_{\in} L(G)$ possédant deux arbres de dérivation différents.

En pratique une telle grammaire est pas très utilisée. En effet, en informatique, il ne serait pas très pratique de pouvoir compiler un code en deux expressions différentes d'un autre langage. On ne saurait pas laquelle choisir. Il faut que la dérivation puisse se faire de façon unique.

Exemple (Grammaire Formelle et Arbre de dérivation) Soit $\Sigma = \{a, b\}$. Soit la grammaire formelle G définie par les règles suivantes telles que $V = \{S\}$:

$$P : \begin{cases} S \longrightarrow aSa \\ S \longrightarrow SbS \\ S \longrightarrow \varepsilon \end{cases} \iff \Big\{ S \longrightarrow aSa \mid bSb \mid \varepsilon$$

Soit $abaaba \in G$ on a alors l'arbre de dérivation suivant pour ce mot :



Cette grammaire reconnaît bien les palindrômes pairs.

Remarque Lors de la dérivations de mots par une grammaire, on remarque qu'il est plus facile que les règles de dérivation possèdent des traces initiales ou finales uniques telles que les a et les b. Elles permettent d'identifier plus facilement les règles à utiliser pour les dérivations.

5.1.4 Grammaires Régulières

Nous allons ici faire le lien entre les deux modèles présentés précedement, les automates fini et les grammaires formelles. Nous allons ainsi définir les grammaires régulières qui permettent de représenter les automates fini déterministes sous la forme que nous venons d'introduire.

Propriété (**Représentation d'un langage automatique**) . Soit $\mathcal{A} = (Q, \Sigma, T, q_0, A)$ un automate fini déterministe. Le langage L reconnu par cet automate peur être engendré par la grammaire :

$$G = (Q, \Sigma, q_0, P)$$

dont les variables auxiliaires sont les états de l'automate et où P est l'ensemble des productions de la forme :

$$q \longrightarrow x.T(q,x) \quad \text{où } q \in Q \text{ et } x \in \Sigma$$

$$q \longrightarrow \varepsilon \quad \text{si } q \in A$$

On peut donc représenter facilement n'importe quel langage automatique par une grammaire formelle. D'où le théorème suivant.

Théorème (**Langage Automatique et Grammaire Formelle**) . Tout langage automatique (reconnaissable par un automate fini) est algébrique (reconnaissable par une grammaire formelle).

L'ensemble des langages automatiques est même strictement inclus dans l'ensemble des langages algébriques. Autrement dit, certains langages sont reconnaissables par une grammaire formelle mais pas par un automate.

On définit ainsi les grammaires régulières.

Définition (**Grammaire Régulière**) . Une grammaire régulière est une grammaire formelle dont toutes les règles de production de P sont de la forme :

$$X \longrightarrow a.Y \quad \text{ou } X \longrightarrow \varepsilon$$

où $X, Y \in V$ et $a \in \Sigma$.

Une grammaire régulière est donc conçue de façon à "laisser des traces" explicites de la structure des mots pour faciliter les dérivations. De même que précédement, on peut passer d'une grammaire régulière à un automate fini déterministe.

Proposition (Représentation d'une grammaire régulière) Soit G une grammaire régulière. Soit L le langage reconnu par G. L'automate fini déterministe reconnaissante aussi L est :

$$\mathcal{A} = (V, \Sigma, T, q_0 = S, A)$$

dont les états sont les variables auxiliaires de G et dont les transitions sont définies par :

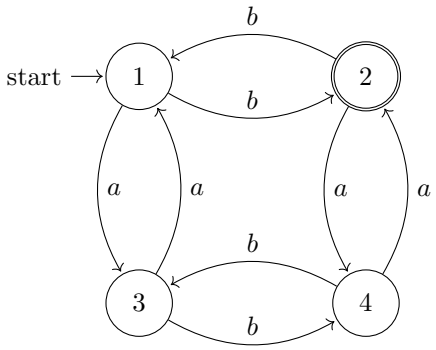
$$q' = T(q, x) \quad \text{si} \quad q \longrightarrow xq' \in P$$

et dont les états acceptants sont définis par :

$$q \in A \quad \text{si} \quad q \longrightarrow \varepsilon \in P$$

Remarque Grâce à cette propriété, les langages automatiques (réguliers) sont donc exactement les langages reconnus par des grammaires régulières. D'où le nom...

Exemple (Construction d'une grammaire régulière) Définissons le langage L reconnaissant les mots contenant un nombre pair de a et impair de b. Alors ce langage est reconnu par l'AFn suivant :



D'après la propriété précédente, L est reconnu par la grammaire régulière $G = (\Sigma, V, S, P)$ où $V = \{S, A, B, C\}$ et les états sont représentés par :

$$\begin{cases} 1 \longrightarrow S \\ 2 \longrightarrow A \\ 3 \longrightarrow B \\ 4 \longrightarrow C \end{cases}$$

On peut ensuite déterminer les règles de production à partir du voisinnage sortant de chaque état de l'automate. De plus, puisque A est un état acceptant, on y rajouter ε.

$$P : \begin{cases} S \longrightarrow bA \mid aB \\ A \longrightarrow bS \mid aC \mid \varepsilon \\ B \longrightarrow aS \mid cB \\ C \longrightarrow aA \mid bB \end{cases}$$

5.2 Simplification de Grammaires

Tout comme les automates, on va chercher à simplifier les grammaires. Or ici, pour un langage donné il n'existe pas de forme minimale de grammaire qui l'engendre.

On va donc chercher à simplifier les grammaires dans le but d'obtenir des formes dites normales pour réduire le nombre de dérivations à faire pour un mot donné. L'idée est de ramener les arbres de dérivation à des arbres binaires donc ekes dérivations sont seulement de deux formes. On pourra donc calculer directement la profondeur de l'arbre de dérivation de n'importe quel mot du langage engendré en fonction de son nombre de caractères.
Pour cela, il faut définir un certain nombre de règles qui serviront à cette simplification. Commençons par un règle très simple :

Définition (**Règle 0**) . On peut toujours éliminer une règle de la forme $X \longrightarrow X$.

5.2.1 Règle 1 : Suppression des epsilon-productions

On va chercher ici à supprimer toutes les ε productions qui ne produisent rien dans la dérivation d'un mot et prennent beaucoup de temps et de place à exécuter.

Définition (**Règle 1 : Suppression des espilon-productions**) . Soit G une grammaire formelle. On définit l'algorithme suivant pour supprimer toutes les ε-productions de G en une grammaire équivalente :

1. On cherche récursivement toutes les variables dont ε dérive (i.e toutes les variables qui peuvent nous donner ε à la fin).
2. On supprimer toutes les règles de la forme $X \longrightarrow \varepsilon$.
3. Pour toutes les variables X de la forme $X \longrightarrow w$ on rajoute toutes les productions $X \longrightarrow u$ avec $u \neq u$ et u est obtenu à partir de w en remplaçant une ou plusieurs variables indentifiées en 1.

Exemple Soit G une grammaire d'alphabet $\Sigma = \{a, b\}$ et $V = \{S, A, B\}$ tel que :

$$P : \begin{cases} S \longrightarrow AB|aS|A \\ A \longrightarrow Ab|\varepsilon \\ B \longrightarrow B|AS \end{cases}$$

On cherche G' telle sans ε-productions telle que :

$$L(G') \cup \{\varepsilon\} = L(G)$$

D'après la règle 1, toutes les variables produisent une ε-production. On obtient donc la grammaire équivalente à ε-production près :

$$P' : \begin{cases} S \longrightarrow AB|A|B|aS|a \\ A \longrightarrow Ab|b \\ B \longrightarrow AS|A|S \end{cases}$$

5.2.2 Règle 2 : Élimination des cycles

Dans les arbres de dérivation, les cycles peuveut conduire à des dérivations infinies. On va donc chercher à les supprimer.

Définition (**Règle 2 : Élimination des cycles**) . Soit G une grammaire formelle. On définit l'algorithme suivant pour supprimer tous les cycles de G en une grammaire équivalente. Soit un cycle de la forme :

$$X_1 \longrightarrow X_{n-1} \longrightarrow X_1$$

Alors on remplace dans P toutes les variables $X_i \forall i \in [\![1, n-1]\!]$ par X_1.

Exemple En reprennant l'exemple précédent :

$$P' : \begin{cases} S \longrightarrow AB|A|B|aS|a \\ A \longrightarrow Ab|b \\ B \longrightarrow AS|A|S \end{cases}$$

On détecte un seul cycle : $S \longrightarrow B \longrightarrow S$. On applique donc l'algorithme pour obtenir :

$$P'' : \begin{cases} S \longrightarrow AS|A|aS|a \\ A \longrightarrow Ab|b \end{cases}$$

5.2.3 Règle 3 : Suppression des changements de variable

Les changement de variable dans les arbres de dérivation font perdre du temps. En effet, ils augmentent la profondeur de l'arbre de dérivation sans produire de lettre. On va donc chercher à les supprimer avec la règle 3.

Définition (**Règle 3 : Suppression des changements de variable**) . Soit G une grammaire formelle. Soit une dérivation de la forme $A \longrightarrow B \longrightarrow C$. Alors on peut la remplacer en $A \longrightarrow C$.

Exemple En reprenant l'exemple précédent, on peut supprimer les changements de variable :

$$P'' : \begin{cases} S \longrightarrow AS|A|aS|a \\ A \longrightarrow Ab|b \end{cases}$$

On a donc les règles de productions suivantes en remplaçant :

$$P''' : \begin{cases} S \longrightarrow AS|Ab|b|aS|a \\ A \longrightarrow Ab|b \end{cases}$$

5.2.4 Forme Normale de Chomsky

Pour l'instant, on ne peut pas encore déterminer la profondeur de l'arbre de dérivation d'un mot. Il faut donc définir une forme, dite Normale de Chomsky, qui va permettre cette estimation.

Définition (**Forme Normale de Chomsky**) . Soit $G = (\Sigma, V, S, P)$ une grammaire formelle. Supposons que G ne contienne ni ε-production, si changements de variables, ni cycles. On définit alors la forme normale de Chomsky de ses règles de production P comme ses mêmes règles de production où seules deux formes sont autorisées :

- Les productions de lettres de la forme $X \longrightarrow a$
- Les dédoublements de variables de la forme $X \longrightarrow AB$

Exemple En reprenant l'exemplement précédent, on obtient la forme normale de Chomsky suivante :

$$\widetilde{P} : \begin{cases} S \longrightarrow AS|AY|b|XS|a \\ A \longrightarrow AY|b \\ X \longrightarrow a \\ Y \longrightarrow b \end{cases}$$

On en déduit donc le théorème suivant :

Théorème (**Majoration de la profondeur**) . Soit $w \in L(G)$ où G est une grammaire formelle sous forme normale de Chomsky. Notons p la profondeur de l'arbre de dérivation de w dans cette grammaire g. On a alors l'inégalité suivante :

$$\boxed{p \leqslant 2|w| - 1}$$