Université Libre de Bruxelles

INFO-F302
Informatique Fondamentale

Synthèse

Étudiants :
Hugo Callens

Enseignants :
E. Filiot

4 janvier 2024

Contents

1 Logique propositionnelle

1.1 Construction de formules

Le vocabulaire du langage de la logique propositionnelle est composé de :

1. de propositions x, y, z, ... ; ou X, Y, Z, ... ;
2. de deux constantes vrai ($\top$ ou 1) et faux ($\bot$ ou 0) ;
3. d'un ensemble de connecteurs logiques : $\neg$, $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$.
4. de paranthèses ( ).

1.2 Sémantique

Définition 1.1. Sémantique

La sémantique d'une formule est la valeur de vérité de cette formule. La valeur de vérité d'une formule Φ formée àpd propositions d'un ensemble X, évaluée avec la fonction d'interprétation V, est notée $[\![\Phi]\!]_V$. La fonction $[\![\Phi]\!]_V$ est définie par induction sur la syntaxe de Φ de la façon suivante :

- $[\![\top]\!]_V = 1$; $[\![\bot]\!]_V = 0$; $[\![x]\!]_V = V(x)$
- $[\![\neg\Phi]\!]_V = 1 - [\![\Phi]\!]_V$
- $[\![\Phi_1 \vee \Phi_2]\!]_V = \max([\![\Phi_1]\!]_V, [\![\Phi_2]\!]_V)$
- $[\![\Phi_1 \wedge \Phi_2]\!]_V = \min([\![\Phi_1]\!]_V, [\![\Phi_2]\!]_V)$
- $[\![\Phi_1 \rightarrow \Phi_2]\!]_V = \max(1 - [\![\Phi_1]\!]_V, [\![\Phi_2]\!]_V)$
- $[\![\Phi_1 \leftrightarrow \Phi_2]\!]_V = \min([\![\Phi_1 \rightarrow \Phi_2]\!]_V, [\![\Phi_2 \rightarrow \Phi_1]\!]_V)$

Nous notons $V \models \Phi \Leftrightarrow [\![\Phi]\!]_V = 1$ soit "V satisfait Φ."

L'information contenue dans la définition est souvent représentée sous forme de table de verité :

Φ_1	Φ_2	$\Phi_1 \vee \Phi_2$	$\Phi_1 \wedge \Phi_2$	$\Phi_1 \rightarrow \Phi_2$	$\Phi_1 \leftrightarrow \Phi_2$
0	0	0	0	1	1
0	1	1	0	1	0
1	0	1	0	0	0
1	1	1	1	1	1

Dans l'implication suivante : $\Phi_1 \rightarrow \Phi_2$, la cas où Φ_1 est faux ne nous intéresse pas. Dans ce cas, l'implication est toujours vraie. Une hypothèse vraie ne peut pas mener à une conclusion fausse.

Exemple: Prenons Φ la formule suivante : $\Phi = (x \vee y) \wedge (\neg y \vee z)$ Considérons alors la fonction d'interprétation V_1 telle que $V_1(x) = 1$, $V_1(y) = 0$ et $V_1(z) = 1$. On a alors :

$$\begin{aligned} [\![\Phi]\!]_{V_1} &= [\![(x \vee y) \wedge (\neg y \vee z)]\!]_{V_1} \\ &= (x \vee y) \wedge (\neg y \vee z) \\ &= (1 \vee 0) \wedge (\neg 0 \vee 1) \\ &= 1 \wedge 1 \\ &= 1 \end{aligned}$$

On a donc $V_1 \models \Phi$.

Si on considère maintenant la fonction d'interprétation V_2 telle que $V_2(x) = 0$, $V_2(y) = 0$ et $V_2(z) = 1$, on a :

$$\begin{aligned} [\![\Phi]\!]_{V_2} &= [\![(x \vee y) \wedge (\neg y \vee z)]\!]_{V_2} \\ &= (x \vee y) \wedge (\neg y \vee z) \\ &= (0 \vee 0) \wedge (\neg 0 \vee 1) \\ &= 0 \wedge 1 \\ &= 0 \end{aligned}$$

On a donc $V_2 \nvDash \Phi$.

1.3 Validité et Stabilité

1.3.1 Définitions

Définition 1.2. Formule propositionnelle satisfaisable

Une formule propositionnelle Φ est **satisfaisable** $\Leftrightarrow$ **il existe** une fonction d'interprétation V pour les propositions de Φ, telle que $V \vDash \Phi$.

Définition 1.3. Formule propositionnelle valide

Une formule propositionnelle Φ est **valide** $\Leftrightarrow$ **pour toute** fonction d'interprétation V pour les propositions de Φ, $V \vDash \Phi$.

Exemple: En reprenant l'exemple précédent, on a que Φ est satisfaisable car $V_1 \vDash \Phi$. Cependant, Φ n'est pas valide car $V_2 \nvDash \Phi$.
Notons alors Φ_1 la formule $\Phi_1 = \neg(x \vee y) \Leftrightarrow (\neg x \wedge \neg y)$. On a alors que Φ_1 est valide car Φ_1 est vraie pour toutes les fonctions d'interprétation. En effet, c'est un loi de Morgan.

1.3.2 Conséquence logique

Définition 1.4. Conséquence Logique

Soit $\Phi_1, ..., \Phi_n, \Phi$ des formules. On dira que Φ est une **conséquence logique** de $\Phi_1, ..., \Phi_n$, noté $\Phi_1, ..., \Phi_n \vDash \Phi$, si $(\Phi_1 \wedge ... \wedge \Phi_n) \rightarrow \Phi$ est valide.

Exemple: Prenons $p, \neg p \vDash \perp$. En effet, on a que $p \wedge \neg p$ est toujours faux.

1.3.3 Equivalence

Définition 1.5. Formules équivalentes

Deux formules, Φ et Ψ, sont **équivalentes** si la formule $\Phi \leftrightarrow \Psi$ est valide. On notera $\Phi \equiv \Psi$.

Pour toutes formules Φ_1, Φ_2, Φ_3 :

- $\neg\neg\Phi_1 \equiv \Phi_1$
- $\neg(\Phi_1 \wedge \Phi_2) \equiv (\neg\Phi_1 \vee \neg\Phi_2)$
- $\neg(\Phi_1 \vee \Phi_2) \equiv (\neg\Phi_1 \wedge \neg\Phi_2)$
- $\Phi_1 \wedge (\Phi_2 \vee \Phi_3) \equiv (\Phi_1 \wedge \Phi_2) \vee (\Phi_1 \wedge \Phi_3)$
- $\Phi_1 \vee (\Phi_2 \wedge \Phi_3) \equiv (\Phi_1 \vee \Phi_2) \wedge (\Phi_1 \vee \Phi_3)$
- $\Phi_1 \rightarrow \Phi_2 \equiv (\neg\Phi_1 \vee \Phi_2)$
- $\Phi_1 \leftrightarrow \Phi_2 \equiv (\Phi_1 \rightarrow \Phi_2) \wedge (\Phi_2 \rightarrow \Phi_1)$

Les relations d'équivalence nous permettent de susbstituer des formules par d'autres équivalentes dans une formule. Cela nous permet de simplifier des formules.

Pour démontrer que deux formules sont équivalentes, on peut utiliser les tables de vérité.

Exemple: Prenons $\Phi_1 \vee (\Phi_2 \wedge \Phi_3) \equiv \Phi_1 \vee \Phi_2) \wedge (\Phi_1 \vee \Phi_3)$. Il nous suffit de vérifier que $\Phi_1 \vee (\Phi_2 \wedge \Phi_3) \Leftrightarrow \Phi_1 \vee \Phi_2) \wedge (\Phi_1 \vee \Phi_3)$ est valide pour toutes les fonctions d'interprétation (Φ_1, Φ_2, Φ_3). Il faut donc vérifier que les 2 formules ont la même table de vérité (même valeur de vérité pour toutes les fonctions d'interprétation). Voici la table de vérité :

Φ_1	Φ_2	Φ_3	$\Phi_1 \vee (\Phi_2 \wedge \Phi_3)$	$(\Phi_1 \vee \Phi_2) \wedge (\Phi_1 \vee \Phi_3)$
0	0	0	0	0
0	0	1	0	0
0	1	0	0	0
0	1	1	1	1
1	0	0	1	1
1	0	1	1	1
1	1	0	1	1
1	1	1	1	1

On voit que les deux formules ont la même table de vérité. Elles sont donc équivalentes.

1.3.4 Lien entre satisfaisabilité et validité

Théorème 1.1. Lien entre satisfaisabilité et validité

Une formule Φ est valide $\Leftrightarrow \neg\Phi$ est insatisfaisable.

En effet, ça signifie que pour toute fonciton d'interprétation, il n'existe pas de fonction d'interprétation qui satisfait $\neg\Phi$. S'il y en avait une, alors Φ ne serait pas valide car il y aurait une entrée pour laquelle Φ est fausse.

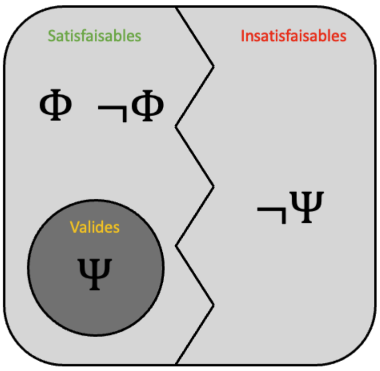


FIGURE 1.1 – Lien entre satisfaisabilité et validité

1.3.5 Tableaux sémantiques

Définition 1.6. Littéral

Un littéral est une proposition x ou la négation d'une proposition $\neg x$.

Théorème 1.2. Satisfaisabilité des littéraux
Un ensemble S de littéraux est satisfaisable **ssi** il ne contient pas de littéraux et leur négation, (pair de littéraux **complémentaire**, x et $\neg x$).

Exemple: $\{x, \neg x\}$ est insatisfaisable. alors que $\{x, y, \neg z\}$ est satisfaisable.

La méthode des tableaux sémantiques est un algorithme pour tester la satisfaisabilité d'une formule. Elle consiste à construire un arbre dont les noeuds sont des formules et les feuilles sont des littéraux. On construit l'arbre de la façon suivante :

- On place la formule à tester à la racine de l'arbre.
- On applique les règles suivantes jusqu'à ce que l'arbre soit complet :
  - Si la formule à tester est une constante, on arrête.
  - Si la formule à tester est une conjonction, on ajoute les deux conjoncteurs comme fils de la formule à tester.
  - Si la formule à tester est une disjonction, on ajoute un fils avec le premier disjoncteur et un autre fils avec le deuxième disjoncteur.
  - Si la formule à tester est une implication, on ajoute un fils avec la négation de l'antécédent et un autre fils avec le conséquent.
  - Si la formule à tester est une équivalence, on ajoute un fils avec la négation de la première formule et un autre fils avec la deuxième formule.
  - Si la formule à tester est une négation, on ajoute un fils avec la négation de la formule à tester.

Remarque: TODO –> Vérifier l'algorithme

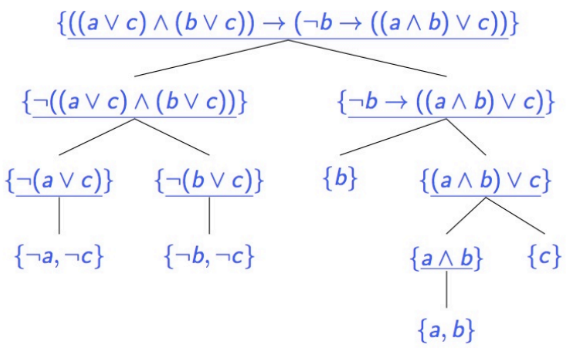


FIGURE 1.2 – Exemple de l'arbre créé avec l'algorithme

2 Déduction naturelle

Définition 2.1. Déduction naturelle

La déduction naturelle est un système de preuve pour la logique propositionnelle. Il est composé de règles d'inférence qui permettent de déduire de nouvelles formules à partir de formules existantes. Une preuve est un arbre dont les noeuds sont des formules et les feuilles sont des axiomes. Une preuve est correcte si elle respecte les règles d'inférence. Une preuve est complète si elle contient toutes les formules qui sont des conséquences logiques des axiomes.

2.1 Règles pour la conjonction

- Règle d'introduction :

$$\frac{\Phi \quad \Psi}{\Phi \wedge \Psi} \bigwedge i$$

- Règle d'élimination :

$$\frac{\Phi \wedge \Psi}{\Phi} \bigwedge e_1 \qquad \frac{\Phi \wedge \Psi}{\Psi} \bigwedge e_2$$

Exemple: La règle d'introduction se lit : si j'ai une preuve de Φ et une preuve de Ψ, alors j'ai une preuve de $\Phi \wedge \Psi$.

2.2 Règles pour la double négation

- Règle d'introduction :

$$\frac{\Phi}{\neg\neg\Phi} \neg\neg i$$

- Règle d'élimination :

$$\frac{\neg\neg\Phi}{\Phi} \neg\neg e$$

2.3 Elimination de l'implication : Modus Ponens

Règle d'élimination :

$$\frac{\Phi \quad \Phi \rightarrow \Psi}{\Psi} \rightarrow_{\text{MP}} (\text{ou} \rightarrow_e)$$

Exemple:

1. Φ : "Il pleut"
2. Ψ : "S'il pleut, "je prends mon parapluie"
3. Alors on en déduit que "je prends mon parapluie"

En contraposition, nous avons le Modus Tollens :

$$\frac{\Phi \rightarrow \Psi \quad \neg\Psi}{\neg\Phi} \rightarrow_{\text{MT}}$$

Exemple:

1. Φ : "s'il pleut, alors la route est mouillée"
2. Ψ : "la route n'est pas mouillée"
3. Alors on en déduit que "il ne pleut pas"

2.4 Règle pour l'introduction de l'implication

$$\frac{\begin{array}{l}\Phi \text{hyp.} \\ \dots \\ \Psi \text{fin hyp.}\end{array}}{\Phi \to \Psi} \to_i$$

Lorsqu'on posera une hypoèse, on indentera l'hypothèse et toutes les lignes de la sous-preuve, jusqu'à la fermeture d'hypothèse.

Exemple: Ici, on va voir un exemple de ce qu'on a vu jusque maintenant :
On veut démontrer : $t \vdash (t \to p) \to (q \to (s \to p))$
Ici le prémisse est "t est vrai", le prémisse sera toujours ce qui se trouve à gauche de la déduction.

1.	t	prémisse.
2.	$t \to p$	hyp$_1$.
3.	q	hyp$_2$.
4.	$s \to p$	hyp$_3$.
5.	p	MP(1,2),fin hyp$_3$.
6.	$s \to p$	$\to_i$ (4,5),fin hyp$_2$.
7.	$q \to (s \to p)$	$\to_i$ (3,6),fin hyp$_1$.
8.	$(t \to p) \to (q \to (s \to p))$	$\to_i$ (2,7).

Nous pouvons également prouver des formules sans prémisse comme suit :

1.	p	hyp
2.	$\neg\neg p$	$\neg\neg_i(1)$, fin hyp.
3.	$p \to \neg\neg p$	$\to_i (1, 2)$.

On a établit que $\vdash p \to \neg\neg p$.

Remarque: Les formules Φ telles que : $\vdash \Phi$ sont appelées des théorèmes.

2.5 Règle pour l'ouverture et la fermeture d'hypothèses

- toute hypothèse introduite doit être fermée.
- on ne peut jamais fermer deux hypothèses en même temps.
- Une fois une hypothèse fermée, on ne peut pas utilsier les formules déduites entre l'ouverture et la fermeture de cette hypothèse.

2.6 Règle pour l'introduction de la disjonction

$$\frac{\Phi}{\Phi \vee \Psi} \vee_{i_2} \qquad \frac{\Psi}{\Phi \vee \Psi} \vee_{i_2}$$

2.7 Elimination de la disjonction

$$\frac{\Psi_1 \vee \Psi_2 \quad \begin{array}{l}\Psi_1\text{hyp.}\\ \dots\\ \Phi\text{fin hyp.}\end{array} \quad \begin{array}{l}\Psi_2\text{hyp.}\\ \dots\\ \Phi\text{fin hyp.}\end{array}}{\Phi}\vee_e$$

Exemple: Supposons que les faits suivants soient vrais :

1. si ma note d'examen est excellente, j'irai boire un verre.
2. si ma note d'examen est bonne, j'irai boire un verre.
3. ma note sera excellente ou bonne.

Alors je peux en déduire que j'irai boire un verre.

Ici aussi, on ne peut pas utiliser l'hypothèse temporaire faite pour l'autre cas. (sauf si elle a été établie avant)

2.8 Règle de copie

$$\frac{\Phi}{\Phi}\text{copie}$$

2.9 Règle pour la négation

Les contradictions sont des formules de la forme :

$$\neg\Phi \wedge \Phi \quad \text{ou} \quad \neg\Phi \wedge \neg\Phi$$

Toutes les contradictions sont logiquement équivalentes à la formule $\perp$. (rappel : $\perp$ est une formule qui est toujours fausse)

Le fait que l'on peut tout déduire à partir d'une contradiction est formalisé par la règle suivante :

$$\frac{\perp}{\Phi}\perp_e$$

Le fait que $\perp$ représente une contradiction est formalisé par la règle suivante :

$$\frac{\Phi \quad \neg\Phi}{\perp}\neg_e$$

Afin d'introduire une négation, supposons que l'on fasse une hypothèse et que l'on arrive à déduire une contradiction, dans ce cas, l'hypothèse est fausse. Ceci est formaisé par la règle suivante :

$$\frac{\begin{array}{l}\Phi \quad \text{hyp.}\\ \dots\\ \perp\text{fin hyp.}\end{array}}{\neg\Phi}\neg_i$$

2.10 Règles pour l'équivalence

$$\frac{\Phi_1 \to \Phi_2 \quad \Phi_2 \to \Phi_1}{\Phi_1 \leftrightarrow \Phi_2}\leftrightarrow_i$$

$$\frac{\Phi_1 \leftrightarrow \Phi_2}{\Phi_1 \to \Phi_2}\leftrightarrow_{e_1} \qquad \frac{\Phi_1 \leftrightarrow \Phi_2}{\Phi_2 \to \Phi_1}\leftrightarrow_{e_2}$$

2.11 Règles dérivées

Il existe de nombreuses formules dérivées qui peuvent s'obtenir à partir des autres règles vues plus haut. (à voir si beaucoup utilisée au TP, si oui, ajouter : MT,RAA,LEM)

2.12 Théorèmes

Théorème 2.1. Cdéquation
Pour tout $\Psi_1, ..., \Psi_n, \Psi$, si $\Psi_1, ..., \Psi_n \vdash \Psi$ alors $\Psi_1, ..., \Psi_n \vDash \Psi$.

Théorème 2.2. Complétude
Pour tout $\Psi_1, ..., \Psi_n, \Psi$, si $\Psi_1, ..., \Psi_n \vDash \Psi$ alors $\Psi_1, ..., \Psi_n \vdash \Psi$.

2.13 Démontrer une implication

Il existe deux méthodes pour démontrer une implication $(A \rightarrow B)$:

1. On suppose A et on en déduit B.
2. On suppose non B et on en déduit non A.

2.14 Démontrer une équivalence

Il existe deux méthodes pour démontrer une équivalence $(A \leftrightarrow B)$:

1. On suppose A et on en déduit B et réciproquement on suppose B et on en déduit A.
2. On prouve une chaîne d'équivalences.

2.15 Preuve par cas

Ce type de preuve repose sur une généralisation de la règle $\vee_e$: si on sait qu'on est soit dans le cas A_1 soit dans le cas A_n, et que pout tout $i \in \{1, ..., n\}$, on peut démontrer une propriété P, alors c'est que P est vraie.

2.16 Preuve par contradiction

On veut démontrer une propriété P. On suppose son contraire $\neg P$ et on en déduit une contradiction. On en déduit que P est vraie.

> **Remarque:** Cette partie du cours nécessite de prendre le temps de bien comprendre les exemples donnés dans le cours ainsi que les exercices vus au TP.

3 Le Problème SAT

Remarque: Soit P un ensemble de propositions. Une formule Φ de la logique propositionnelle sur P est satisfaisable s'il exite une interprétation

$$V : X \to \{1, 0\}, \text{telle que } V \models \Phi$$

3.1 Littéraux et Clauses

Définition 3.1. Littéral

Un **littéral** est une variable x ou sa négation $\neg x$.

Définition 3.2. Clause

Une **clause** est une disjonction de littéraux $l_1 \vee l_2 \vee \cdots \vee l_n$. Elle est satisfaite par une valuation V s'il existe i tel que $V(l_i) = 1$. Par extension, on appelle $\perp$ la clause *vide*, qui est insatisfaisable.

Exemple: p et $\neg p$ sont des littéraux. $x \vee \neg y \vee r$ est une clause mais pas $x \wedge y$.

Théorème 3.1. Satisfaction d'ensemble de clauses

Un ensemble de clauses $A = \{C_1, \ldots, C_n\}$ est satisfait par une valuation V, notée $V \models A$, si pour tout i, $V \models C_i$. En particulier, tout valuation satisfait l'ensemble vide $A = \emptyset$.

3.1.1 Lien avec les formes normales

- Une formule est en forme normale conjonctive (**FNC**) si et seulement si c'est une conjonction de disjonctions de littéraux, de la forme :
$$\bigwedge_i (\bigvee_j (\neg) x_{i,j})$$
- Une formule est en forme normale disjonctive (**FND**) si et seulement si c'est une disjonction de conjonctions de littéraux, de la forme :
$$\bigvee_i (\bigwedge_j (\neg) x_{i,j})$$

Lemme 3.1. Equivalence formule FNC Tout ensemble non-vide de clauses $A = \{C_1, \ldots, C_n\}$ est équivalent à la formule en FNC $\Phi_A = \bigwedge_{i=1}^n C_i$, au sens où pour toute valuation, $V \models A \Leftrightarrow V \models \Phi_A$.

Remarque: Pour mettre une formule sous forme de clauses, il suffit de la mettre en FNC.

3.1.2 Mise sous FNC et FND

Afin de parvenir à une FNC ou une FND, on utilise les transformations successives suivantes pour obtenir les formes normales :

1. élimination des connecteurs $\to$ et $\leftrightarrow$ grâce aux équivalences suivantes :
$$(\Phi \to \Psi) \equiv (\neg\Phi \vee \Psi)$$
$$(\Phi \leftrightarrow \Psi) \equiv (\neg\Psi \vee \Psi) \wedge (\Phi \vee \neg\Psi)$$
2. entrer les négations le plus à l'intérieur possible :
$$\neg(\Phi \wedge \Psi) \equiv (\neg\Phi \vee \neg\Psi) \qquad \neg\neg\Phi \equiv \Psi$$
$$\neg(\Psi \vee \Psi) \equiv (\neg\Phi \wedge \neg\Psi)$$

3. utilisation des distributivité de $\vee$ et $\wedge$:

$$(\Phi \vee (\Psi \wedge \chi)) \equiv (\Phi \vee \Psi) \wedge (\Phi \vee \chi)(\text{mise sous FNC})$$
$$(\Phi \wedge (\Psi \vee \chi)) \equiv (\Phi \wedge \Psi) \vee (\Phi \wedge \chi)(\text{mise sous FND})$$

Exemple: Illustrons cela à travers un exemple, mettons la formule $\neg(x \leftrightarrow (y \rightarrow r))$

1. on retire les équivalences et implications :
$$\neg((\neg x \vee \neg y \vee r) \wedge (\neg(\neg y \vee r) \vee x))$$
2. on pousse les négations à l'intérieur :
$$(x \wedge y \wedge \neg r) \vee ((\neg y \vee r) \wedge \neg x)$$
3. on distribue :
$$(x \wedge y \wedge \neg r) \vee ((\neg y \vee r) \wedge \neg x)$$
4. et encore :
$$(x \vee \neg y \vee r) \wedge (y \vee \neg y \vee r) \wedge (\neg r \vee \neg y \vee r) \wedge (x \vee \neg x) \wedge (y \vee \neg x) \wedge (\neg r \vee \neg x)$$
5. on retire les formules équivalentes à $\top$:
$$(x \vee \neg y \vee r) \wedge (y \vee \neg x) \wedge (\neg r \vee \neg x)$$

Invitation à prendre le temps de faire cette démonstrations sur une feuille étape par étape pour voir si tout a bine été compris.

3.2 Problème SAT

Définition 3.3. Problème SAT

Une **entrée** est un ensemble de clauses S.
Une **sortie** est de la forme : Est-ce que S est **sat**isfaisable ?

Si S est satisfaisable, on aimerait que l'algorithme nous retourne une valuation V qui satisfait S.

Remarque: Actuellement, on ne sait toujours pas s'il existe un algorithme pour ce problème dont la complexité en temps est polynomiale. L'intuition veut que ce ne soit pas le cas mais aucune preuve n'a pu être fournie jusqu'à présent. Si on parvenait à le prouver, on pourrait gagner un prix d'un million de dollars ;). C'est un problème qui appartient à la classe NP. (cf. 5)

3.3 Introduction aux solveurs SAT

Définition 3.4. Solveur SAT

Un solveur est un programme qui décidé le problème SAT. Si la formule est satisfaisable, une interprétation qui la satifsait est retournée. Ils ont une complexité au pire cas exponentielle.

3.3.1 Notations pour les grandes disjonctions et conjonctions

Définition 3.5. disjonction et conjonction

- On appelera *disjonction* une formule de la forme $\Phi_1 \vee \Phi_2 \vee \cdots \vee \Phi_n$ où $n \geq 1$. Attention, lorsque $n = 1$, la disjonction est réduite à la seule forme Φ_1. On utilisera l'abbréviation suivante :
$$\bigvee_{i=1}^{n} \Phi_i$$

- On appelera *conjonction* une formule de la forme $\Phi_1 \wedge \Phi_2 \wedge \cdots \wedge \Phi_n$ où $n \geq 1$. Attention, lorsque $n = 1$, la conjonction est réduite à la seule forme Φ_1. On utilisera l'abbréviation suivante :

$$\bigwedge_{i=1}^{n} \Phi_i$$

3.4 Modélisation

Dans cette section, nous feront la modélisation du problème des 8 reines, d'autres exemples sont présents dans le cours mais le raisonnement demeure identique.

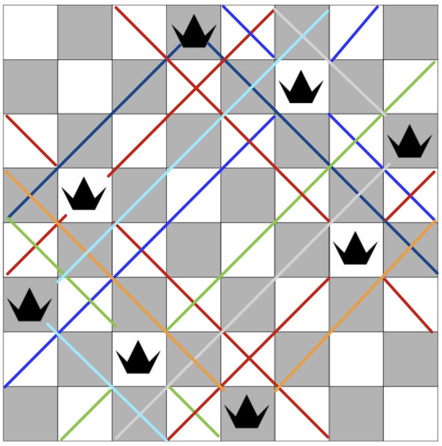


FIGURE 3.1 – Problème des 8 reines

3.4.1 Choix des variables

$$X = \{x_{i,j} | i, j \in \{1, \ldots, 8\}\}$$

La sémantique est la suivante : "$x_{i,j}$ est vraie si et seulement s'il y a une reine en case (i, j)".

3.4.2 Expression des contraintes

1. Pas d'attaque horizontale
2. Pas d'attaque verticale
3. Pas d'attaque en diagonale
4. Au moins une reine par ligne

Pas d'attaque horizontale

"Pour toute ligne, il n'existe pas deux reines sur cette ligne."

$$\bigwedge_{i \in \{1,\ldots,8\}} \neg (\bigvee_{j,j' \in \{1,\ldots,8\} | j \neq j'} x_{i,j} \wedge x_{i,j'})$$
$$\equiv \bigwedge_{i,j,j' \in \{1,\ldots,8\} | j \neq j'} (\neg x_{i,j} \vee \neg x_{i,j'})$$

Pas d'attaque verticale

"Pour toute colonne, il n'existe pas deux reines sur cette colonne."

$$\bigwedge_{j \in \{1,\ldots,8\}} \neg (\bigvee_{i,i' \in \{1,\ldots,8\} | i \neq i'} x_{i,j} \wedge x_{i',j})$$

$$\equiv \bigwedge_{i,i',j \in \{1,\ldots,8\} | i \neq i'} (\neg x_{i,j} \vee \neg x_{i',j})$$

Pas d'attaque en diagonale

"Pour toute diagonale, il n'existe pas deux reines sur cette diagonale."

$$\bigwedge_{i,i',j,j' \in \{1,\ldots,8\} | i \neq i' \wedge j \neq j'} \neg(x_{i,j} \wedge x_{i',j'} \wedge |i - i'| = |j - j'|)$$
$$\equiv \bigwedge_{i,i',j,j' \in \{1,\ldots,8\} | i \neq i' \wedge j \neq j'} (\neg x_{i,j} \vee \neg x_{i',j'} \vee |i - i'| \neq |j - j'|)$$

Au moins une reine par ligne

"Pour toute ligne, il existe au moins une reine sur cette ligne."

$$\bigwedge_{i \in \{1,\ldots,8\}} (\bigvee_{j \in \{1,\ldots,8\}} x_{i,j})$$

3.5 Algorithme DPLL

Ne sera pas à l'examen.

3.6 Transformation de Tseitin

Il se peut que le problème ne s'exprime pas facilement par une formule en FNC. Le but de la transformation de Tseitin est d'ajouter de nouvelles variables et des équivalences.

Exemple: Considérons $\Phi = (x \wedge q) \vee \neg(y \vee r)$, Dans la transformation de Tseitin, on va remplacer $x \wedge q$ par une nouvelle variable x_1 et $\neg(y \vee r)$ par une nouvelle variable x_2. Voici la formule considérée :

$$(x_1 \vee x_2) \wedge (x_1 \leftrightarrow x \wedge q) \wedge (x_2 \leftrightarrow \neg(y \vee r))$$

Il reste encore à mettre les deux formules en FNC : (cf. 3.1.2)

- $x_1 \leftrightarrow x \wedge q$
$$\begin{aligned} &\equiv (x_1 \rightarrow (x \wedge q)) \wedge ((x \wedge q) \rightarrow x_1) \\ &\equiv (\neg x_1 \vee (x \wedge q)) \wedge (\neg(x \wedge q) \vee x_1) \\ &\equiv (\neg x_1 \vee x) \wedge (\neg x_1 \vee q) \wedge (\neg x \vee \neg q \vee x_1) \end{aligned}$$
- $x_2 \leftrightarrow \neg(y \vee r)$
$$\begin{aligned} &\equiv (x_2 \rightarrow \neg(y \vee r)) \wedge (\neg(y \vee r) \rightarrow x_2) \\ &\equiv (\neg x_2 \vee \neg y \wedge \neg r) \wedge (y \vee r \vee x_2) \\ &\equiv (\neg x_2 \vee \neg y) \wedge (\neg x_2 \vee \neg r) \wedge (y \vee r \vee x_2) \end{aligned}$$

Dès lors, nous pouvons écrire la formule sous FNC suivante :

$\Psi = (x_1 \vee x_2) \wedge (\neg x_1 \vee p) \wedge (\neg x_1 \vee q) \wedge (\neg x \vee \neg y \vee x_1) \wedge (\neg x_2 \vee \neg q) \wedge (\neg x_2 \vee \neg r) \wedge (y \vee r \vee x_2)$

La transformation de Tseitin est intéressante lorsqu'on devra mettre sous FNC des formules qui sont sous forme normale disjonctive $C_1 \vee C_2 \vee \cdots \vee C_n$. Car il suffit d'introduire une variable x_i pour chaque C_i et on obtient la formule :

$$(x_1 \vee x_2 \vee \cdots \vee x_n) \wedge (x_1 \leftrightarrow C_1) \wedge \cdots \wedge (x_n \leftrightarrow C_n)$$

Si on suppose que $C_i = l_1 \wedge \ldots l_k$ où l_i sont des littéraux, alors mettre $x_i \leftrightarrow C_i$ sous FNC est assez simple :

$$x_i \leftrightarrow C_i \equiv (\bigwedge_{j=1}^{k} (\neg x_i \vee l_j)) \wedge (x_i \vee \bigvee_{j=1}^{k} (\neg l_j))$$

Il ne faut surtout pas hésiter à ajouter des variables pour minimiser le nombre de clauses.

3.7 Variantes du problème SAT

3.7.1 2-SAT

Les clauses ne contiennent qu'au plus deux littéraux. C'est solvable en temps polynomial alors que 3-SAT est NP-complet.

Exemple:

$$(x_1 \vee x_2) \wedge x_3 \wedge (x_4 \vee \neg x_1) \wedge (\neg x_3 \vee \neg x_2)$$

3.7.2 QSAT

Ici, on décide la satisfaisabilité de formules de la forme :

$$\forall x_1 \exists q_1 \forall x_2 \exists q_2 \dots \forall x_n \exists q_n \cdot \Phi$$

Où Φ est une formule en FNC. On peut montrer que QSAT est PSPACE-complet car QSAT peut être résolu en espace polynomial et tous les problèmes pouvant être résulus en espace polynomial peuvent se réduire au problème QSAT en temps polynomial, autrement dit, QSAT est aussi difficile que tous les problèmes pouvant être résolu en espace polynomial.

Exemple: Voici un exemple de formule QSAT :

$$\forall x \exists q \cdot (x \vee q) \wedge (\neg x \vee q)$$

3.7.3 MAX-SAT

Étant donné une formule Φ en FNC et un entier $k \in \mathbb{N}$, peut-on satisfaire au moins k clauses ?

Exemple: Pour :

$$x \wedge y \wedge (\neg x \vee \neg y)$$

On ne peut satisfaire au plus que deux clauses.

3.7.4 WEIGHTED-MAX-SAT

On attribue des poids à chaque clause, on se donne un entier r et on veut savoir si on peut satisfaire un ensemble de clauses dont la somme des poids est au moins r.

Exemple: Pour :

$$x \wedge y \wedge (\neg x \vee \neg y)$$

avec poids(x)=1, poids(y)=1 et poids($\neg x \vee \neg y$)=2 et $r = 3$. En mettant x à vrai et y à faux, on satisfait la première et la troisième clause et on a le poids total de 3.

4 Automates

4.1 Introduction

Exemple: Un automate fini :
- lit la séquence des lettres de gauche à droite.
- possède un nombre fini d'états.
- en fonction de la lettre courante et de la lettre lue, se déplace vers un autre état.
- possède un unique état initial ainsi que des états finaux.
- accepte un mot si et seulement s'il se termine sur un état final.

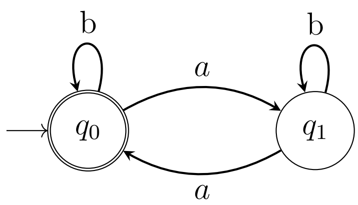


FIGURE 4.1 – Exemple d'automate fini

4.2 Définitions et exemples

Définition 4.1. Langage
- un alphabet est un ensemble fini, que l'on note Σ.
- ses éléments sont appelés lettres ou symboles.
- un mot est une suite de symboles, ϵ est le mot vide.
- l'ensemble des mots sur Σ est noté Σ^*.
- un langage est un sous-ensemble de Σ^*. ($L \subseteq \Sigma^*$)

Définition 4.2. Automate fini

Un automate fini A sur un alphabet Σ est un 4-uplet (Q, q_0, F, δ) où :
- Q est un ensemble fini d'états.
- $q_0 \in Q$ est l'état initial.
- $F \subseteq Q$ est l'ensemble des états finaux.
- $\delta : Q \times \Sigma \to Q$ est la **fonction** de transition.

Définition 4.3. Exécution

Une exécution d'un automate A est une suite finie $e = q_0\sigma_1 q_1\sigma_2 ... \sigma_n q_n (n \geq 0)$ telle que :
- $\forall i \in \{0, ..., n\}, q_i \in Q$.
- $\forall i \in \{1, ..., n\}, \sigma_i \in \Sigma$.
- $\forall i \in \{0, ..., n-1\}, \delta(q_i, \sigma_{i+1}) = q_{i+1}$.

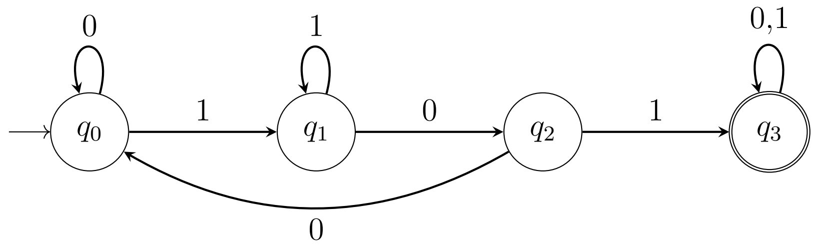
Une exéution e est dite **acceptante** si $q_n \in F$.

Définition 4.4. Langage accepté

Le langage accepté par un automate A (noté $L(A)$) est l'ensemble des mots pour lesquels il existe une exécution acceptante de A.

$$L(A) = \{w \in \Sigma^* | \exists e \text{ exécution acceptante de } A \text{ sur } w\}$$

Exemple: Pour $\Sigma = \{0, 1\}$:



Le langage accepté par cet automate est :

$$L(A) = \{w \in \{0, 1\}^* | w \text{ contient le facteur } 101\}$$

Définition 4.5. Automate Complet

Un automate A est dit *complet* si sa fonction de transition est totale.

Lemme 4.1. Transformation d'un automate en un automate complet On peut toujours transformer un automate A en un automate B complet qui accepte le même langage, tel que $L(A) = L(B)$.

Remarque: Effectivement, il *suffit* de rajouter un état supplémentaire (état puit) non final et ajouter les transitions manquantes vers cet état.

4.2.1 Test du vide

Le problème du vide est le suivant :

— Entrée ⇒ Étant donné un automate A sur un alphabet Σ.

— Sortie ⇒ est-ce que $L(A) = \emptyset$?

Définition 4.6. Etats atteignables

Soit $A = (Q, q_0, F, \delta)$ un automate sur un alphabet Σ. Un état $q \in Q$ est dit *atteignable* si il existe un mot $w \in \Sigma^*$ et une exécution de A sur w qui termine en q.

Remarque: L'ensemble des états atteignables d'un automate peut être calculé en temps $O(n + m)$, où n est le nombre d'états et m le nombre de transitions.

Théorème 4.1. Test du vide

Étant donné un automate A avec n états et m transitions, on peut tester en temps $O(n+m)$ si $L(A) = \emptyset$.

L'algorithme serait le suivant :

1. Calculer l'ensemble des états atteignables R de A.
2. tester si $R \cap F = \emptyset$.

4.2.2 Opérations Booléennes

Définition 4.7. Complément

Le complément d'un langage $L \subseteq \Sigma^*$ est le langage, noté $\overline{L}$, défini par :

$$\overline{L} = \{w \in \Sigma^* | w \notin L\} = \Sigma^* \setminus L$$

Exemple: Si L est l'ensemble des mots sur $\{a, b\}$ qui contiennent au moins un a, alors $\overline{L}$ est l'ensemble des mots qui contiennet au moins deux a, ou pas de a.

Définition 4.8. Union et intersection

Soient $L_1, L_2 \subseteq \Sigma^*$ deux langages. L'union et l'intersection de L_1 et L_2 sont définies par :

$$L_1 \cup L_2 = \{w \in \Sigma^* | w \in L_1 \textbf{ ou } w \in L_2\}$$
$$L_1 \cap L_2 = \{w \in \Sigma^* | w \in L_1 \textbf{ et } w \in L_2\}$$

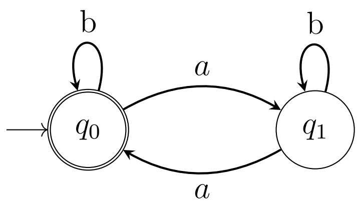
Théorème 4.2. Clôtures des automates par opératiions Booléennes

Soient A, A_1, A_2 des automates finis sur un alphabet Σ. Il existe des automates A_c, U, I tels que :

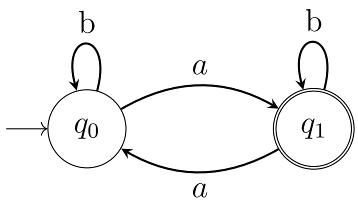
$$L(A_c) = \overline{L(A)}$$
$$L(U) = L(A_1) \cup L(A_2)$$
$$L(I) = L(A_1) \cap L(A_2)$$

Exemple: Si $A = (Q, q_0, F, \delta)$ est complet (s'il n'est pas complet, il faut le compléter avant), il suffit de prendre $A_c = (Q, q_0, Q \setminus F, \delta)$.

- Si $A =$



- Alors, $A_c =$



Définition 4.9. Produit d'automates

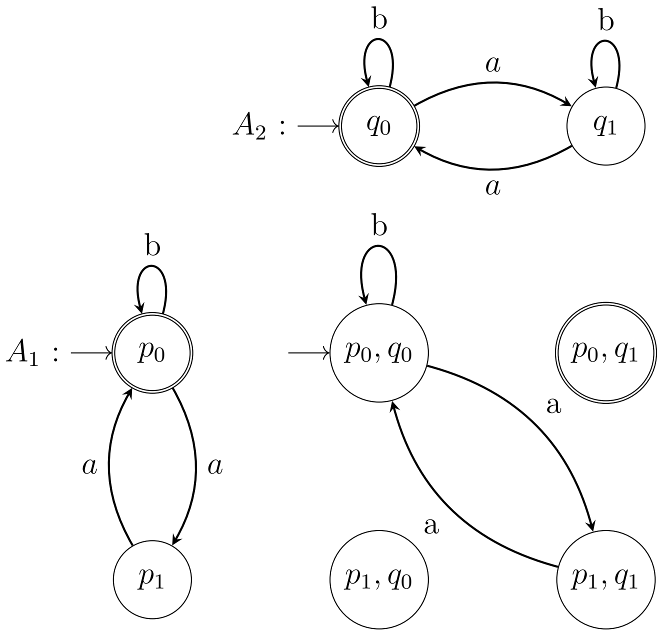
Soient $A_1 = (Q_1, q_{01}, F_1, \delta_1)$ et $A_2 = (Q_2, q_{02}, F_2, \delta_2)$ deux automates sur un alphabet Σ. Le produit de A_1 et A_2 (noté $A_1 \otimes A_2$) est le pré-automate défini par :

$$A_1 \otimes A_2 = (Q_1 \times Q_2, (q_0^1, q_0^2), \delta_{12})$$

où, $\forall\ (q_1, q_2) \in Q_1 \times Q_2$ et $\forall\ \sigma \in \Sigma$:

$$\delta_{12}((q_1, q_2), \sigma) = \begin{cases} \text{indéfini} & \text{si} \delta_1(q_1, \sigma) \text{ est indéfinie} \\ \text{indéfini} & \text{si} \delta_2(q_2, \sigma) \text{est indéfinie} \\ (\delta_1(q_1, \sigma), \delta_2(q_2, \sigma)) & \text{sinon.} \end{cases}$$

Exemple: Voici un produit d'automates :



- $L(A_1)$ est l'ensemble des mots qui contiennent un nombre pair de b.
- $L(A_2)$ est l'ensemble des mots qui contiennent un nombre pair de a.
- Toute exécution de $A_1 \otimes A_2$ sur un mot w simule ene parallèle l'exécution de A_1 sur w, ainsi que celles de A_2 sur w.
  - pour $w = abba$
  - sur A_1, on a $e_1 = p_0\ a\ p_0\ b\ p_1\ b\ p_0\ a\ p_0$.
  - sur A_2, on a $e_2 = q_0\ a\ q_1\ b\ q_1\ b\ q_1\ a\ q_0$.
  - sur $A_1 \otimes A_2$, on a $e_{12} = (p_0, q_0)\ a\ (p_0, q_1)\ b\ (p_1, q_1)\ b\ (p_0, q_1)\ a\ (p_0, q_0)$.
- Pour avoir $L(A_1) \cap L(A_2)$, il suffit de prendre $F_\cap = F_1 \times F_2$ pour les états finaux de $A_1 \otimes A_2$.
- Pour avoir $L(A_1) \cup L(A_2)$, il suffit de prendre $F_\cup = (F_1 \times Q_2) \cup (Q_1 \times F_2)$ pour les états finaux de $A_1 \otimes A_2$.

Théorème 4.3. Clôture par union et intersection

Soient $A_1 = (Q_1, q_{01}, F_1, \delta_1)$ et $A_2 = (Q_2, q_{02}, F_2, \delta_2)$ deux automates. Soit $A_1 \otimes A_2 = (Q_1 \times Q_2, (q_0^1, q_0^2), \delta_{12})$ le pré-automate produit.

- Si A_1 et A_2 sont **complets** et $U = (Q_1 \times Q_2, (q_0^1, q_0^2), (F_1 \times Q_2) \cup (Q_1 \times F_2), \delta_{12})$, alors
$$L(U) = L(A_1) \cup L(A_2)$$

- Si $I = (Q_1 \times Q_2, (q_0^1, q_0^2), (F_1 \times F_2), \delta_{12})$, alors
$$L(I) = L(A_1) \cap L(A_2)$$

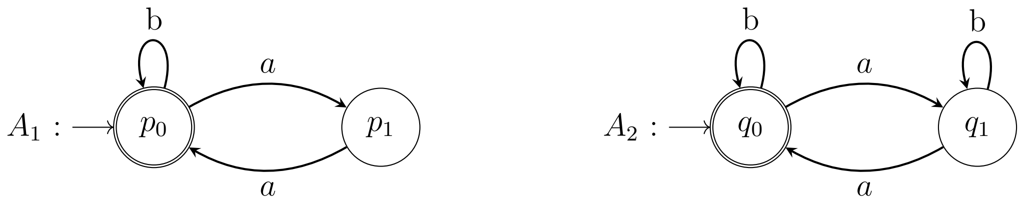
Définition 4.10. Automates équivalents

Deux automates A_1 et A_2 sont **équivalents** si $L(A_1) = L(A_2)$.

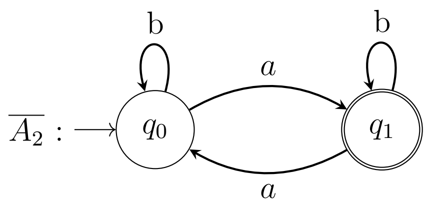
Théorème 4.4. Automates équivalents

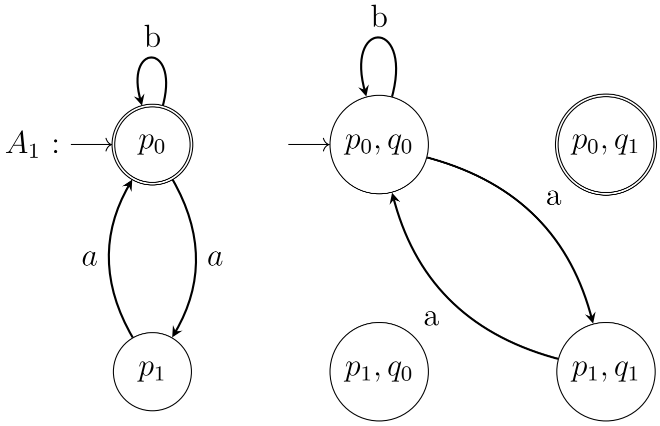
Étant donnés deux automates A_1 et A_2, il est décidable en temps polynomial si $L(A_1) \subseteq L(A_2)$, et si $L(A_1) = L(A_2)$.

Exemple: Soient A_1 et A_2 deux automates :



Est-ce que $L(A_1) \subseteq L(A_2)$?





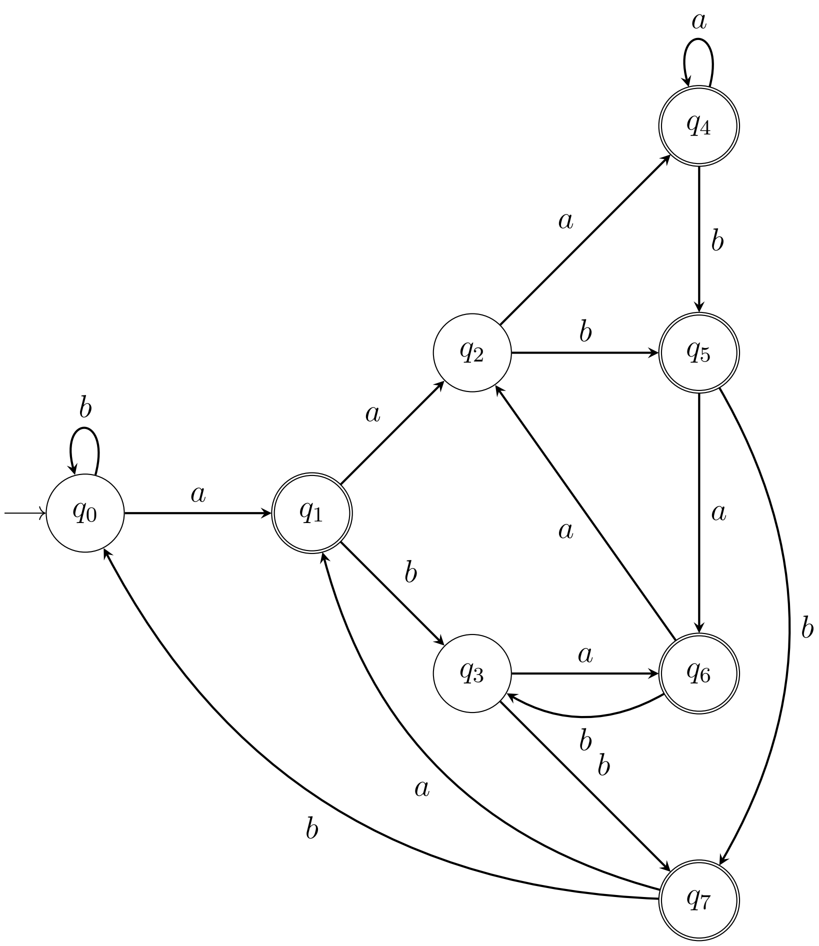
Nous pouvons dès lors vérifier que $L(A_1) \cap \overline{L(A_2)} = \emptyset$ et que donc on a bien $L(A_1) \subseteq L(A_2)$. Les transitions sortantes de (p_1, q_0) et (p_0, q_1) ne sont pas utiles car ces états ne sont pas atteignables.

4.3 Automates non-déterministes

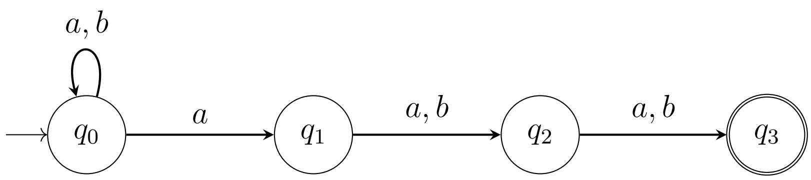
Exemple: Prenons l'alphabet $\Sigma = \{a, b\}$. Donner un automate qui accepte :

$$L_3 = \{u \in \Sigma^* \mid |u| \geq 3 \text{ et } u[|u| - 3] = a\}$$

Un automate déterministe répondant à cette question est le suivant :



Nous pouvons le réduire en un automate non-déterministe :



Dans le cas du mot *abaab*, nous avons plusieurs exécutions possibles :

$$q_0 \xrightarrow{a} q_0 \xrightarrow{b} q_0 \xrightarrow{a} q_0 \xrightarrow{a} q_0 \xrightarrow{b} q_0$$
$$q_0 \xrightarrow{a} q_0 \xrightarrow{b} q_0 \xrightarrow{a} q_0 \xrightarrow{a} q_1 \xrightarrow{b} q_2$$
$$q_0 \xrightarrow{a} q_0 \xrightarrow{b} q_0 \xrightarrow{a} q_1 \xrightarrow{a} q_2 \xrightarrow{b} q_3$$
$$\dots$$

Comme l'exécution 3 est acceptante, le mot est accepté, effectivement, il suffit qu'une des exécutions possible soit acceptante pour que le mot soit accepté. À contrario, pour le mot *abbab*, aucune exécution n'est acceptante, donc le mot n'est pas accepté.

Définition 4.11. Automate non-déterministe

Un automate fini non-déterministe (AFN) A sur un alphabet Σ est un 4-uplet (Q, q_0, F, Δ) où :

- Q est un ensemble fini d'états.
- $q_0 \in Q$ est l'état initial.
- $F \subseteq Q$ est l'ensemble des états finaux.
- $\Delta \subseteq Q \times \Sigma \times Q$ est une relation, appelée **relation** de transition.

Définition 4.12. Langage accepté d'un automate non-déterministe

Étant donné un AFN A, le langage accepté par A, noté $L(A)$ est l'ensemble des mots pour lesquels il existe une exécution acceptante de A.

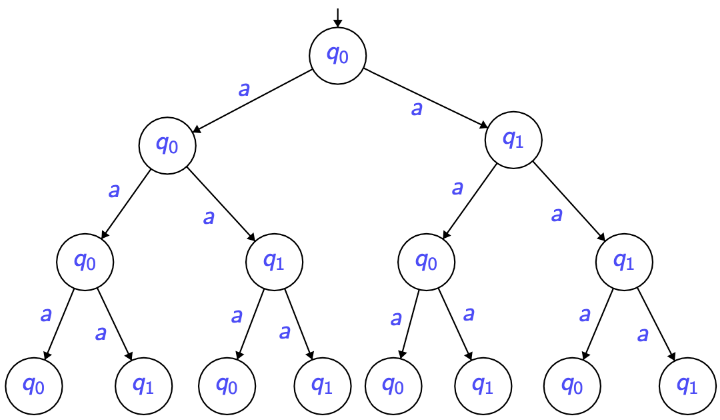
$$L(A) = \{w \in \Sigma^* | \exists e \text{ exécution acceptante de } A \text{ sur } w\}$$

Remarque: Voici deux propriétés des automates non-déterministes :

1. Pour un AFN à n états, il y a au plus n^m exécutions sur un mot de longueur m.
2. Pour un AFN A et un mot u, la complexité de décider si $u \in L(A)$ est polynomiale. Ceci sera démontré juste après.

4.3.1 Arbre des exécutions

Toutes les exécutions peuvent être représentées par un arbre, par exemple :



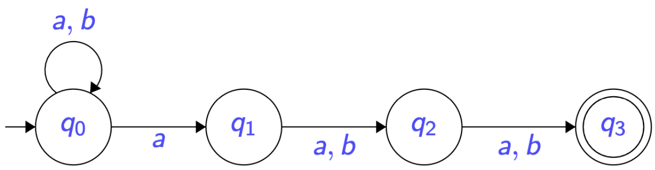
Pour voir si un mot appartient au langage, voici ce qui sera développé :

- on va exploiter l'idée précédente pour avoir une manière efficace de tester $u \in L(A)$.
- Soit $P \subseteq Q$ et $\sigma \in \Sigma$. On note :
  $$\text{Post}_A(P, \sigma) = \{p | \exists p' \in P \cdot (p', \sigma, p) \in \Delta\}$$
  l'ensemble des états qu'on peut atteindre à partir des états de P en lisant σ.
- L'algorithme est le suivant :
  TEST(A,P,u) :
  case u = epsilon :**return** P $\cap$ F $\neq \emptyset$
  case u = σv ($\sigma \in \Sigma$) : **return** TEST(A, $Post_A(P, \sigma)$, v)
- $u \in L(A) \Leftrightarrow TEST(A, \{q_0\}, u)$

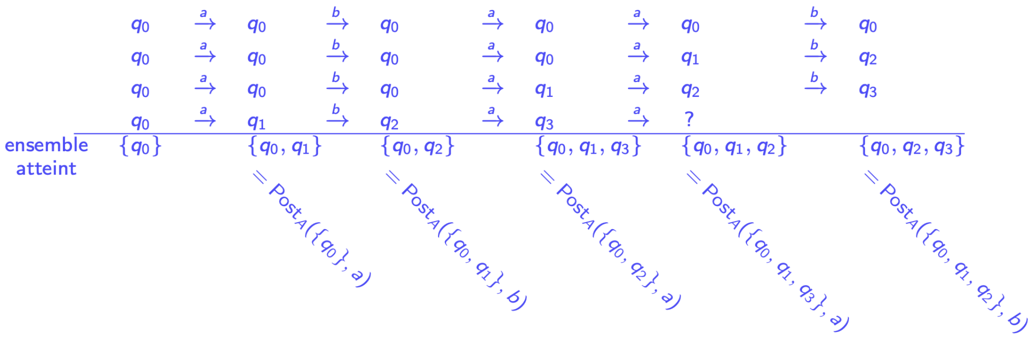
Théorème 4.5. Appartenance au langage non-déterministe

Étant donné un AFN A avec n transitions et un mot u de longueur m, on peut tester en temps $O(|u|m)$ si $u \in L(A)$.

Remarque: Effecitvement, il suffit de remarquer dans l'algorithme que calculer $Post_A(P, \sigma)$ prend $O(m)$



Exécutons l'algorithme sur le mot *abaab* :



L'ensemble final est $\{q_0, q_1, q_2, q_3\}$ et il contient un mot acceptant et donc le mot est accepté.

4.3.2 Test du vide

Théorème 4.6. Test du vide

Étant donné un AFN A avec n états et m transitions, on peut tester en temps $O(n + m)$ si $L(A) = \emptyset$.

C'est la même chose que pour les automates déterministes.

4.3.3 Déterminisme VS non-déterminisme

Tout langage accepté par un automate fini (déterministe) peut être accepté par un automate fini non-déterministe (et inversément).

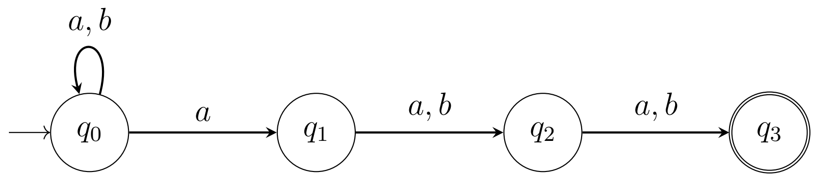
Théorème 4.7. Déterminisme VS non-déterminisme

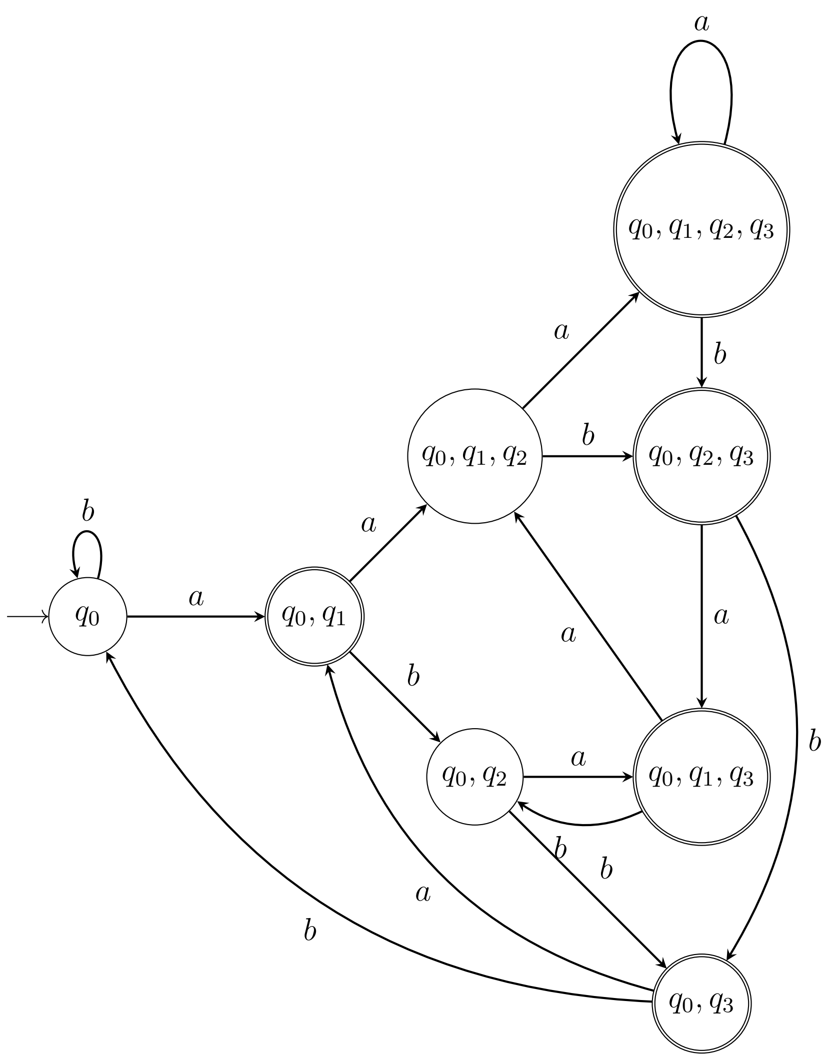
Soit L un langage sur l'alphabet Σ. L est accepté par un automate fini si et seulement s'il est accepté par un automate fini non-déterministe.

Exemple: Voici un exemple et l'explication de cette conversion :

- l'idée est la même que pour tester l'appartenance au langage :
  l'automate déterministe B qui simule l'automate non-déterministe A calcule le sous-ensemble d'états atteints.
- Les états de B seront donc des sous-ensembles d'états de A.
- À partir d'un sous-ensemble $P \subseteq Q$, en lisant une lettre σ, B va vers l'état $Post_A(P, \sigma)$.

- Les états acceptants de B sont les sous-ensembles de Q qui contiennent un état final de A.





L'automate B construit a exponentiellement plus d'états que A. On peut montrer que $\forall n \geq 0$, le plus petit automate fini déterministe acceptant le langage L_n des mots de longueur au moins n dont la $n^{\text{ème}}$ lettre en partant de la fin est a sur l'alphabet $\{a, b\}$, a 2^n états. Tandis que le plus petit AFN pour L_n a $O(n)$ états.

4.4 Expressions rationnelles

Définition 4.13. Expression rationnelle

Une expression rationnelle E sur un alphabet Σ est une expresssion qui respecte la grammaire suivante :

$$E ::= \epsilon \mid a \mid \emptyset \mid (E + E) \mid (E \cdot E) \mid (E^*)$$

pour tout $a \in \Sigma$.

Définition 4.14. Opérations sur les langages

Soient $L, L_1, L_2 \subseteq \Sigma^*$ trois langages. Alors :

- $L_1 \cdot L_2 = \{u_1 u_2 | u_1 \in L_1 \wedge u_2 \in L_2\}$ (noté aussi $L_1 L_2$)
- $L^* = \{u_1, \ldots, u_k | k \geq 0, u_i \in L \forall i \in \{1, \ldots, k\}\}$

Exemple: Voici deux exemples :

- Si $L_1 = \{a, b\}$ et $L_2 = \{a, bb\}$ alors $L_1 \cdot L_2 = \{aa, abb, ba, bbb\}$
- Si $L = \{a\}$ alors $L^* = \{a^n | n \geq 0\}$

Définition 4.15. Sémantique des expresssions rationnelles

La sémantique d'une expression rationnelle E sur Σ est donnée par un langage, noté $L(E)$, défini inductivement par :

— $L(\epsilon) = \{\epsilon\}$
— $L(a) = \{a\}$ pour tout $a \in \Sigma$
— $L(\emptyset) = \emptyset$
— $L(E_1 + E_2) = L(E_1) \cup L(E_2)$
— $L(E_1 \cdot E_2) = L(E_1) \cdot L(E_2)$
— $L(E^*) = L(E)^*$

Exemple: Sur $\Sigma = \{a, b\}$, voici quelques exemples :

— $L((a + b)^* a(a + b)*)$ est l'ensemble des mots qui contiennent au moins un a.
— $L(a^* b^*)$ est l'ensemble des mots qui sont des séquences de a suivies des séquences de b.

4.4.1 Expressions vers automates

Théorème 4.8. Théorème de Kleene

Tout langage est reconnaissable par un automate si et seulement s'il est définissable par une expression rationnelle.

La construction se fait par induction sur les expressions. Pour toute expression E, on va construire un AFN A_E tel que $L(A_E) = L(E)$ (il suffit ensuite de déterminiser A_E avec la construction des sous-ensembles pour terminer la preuve).

- si $E = \epsilon$, alors $A_E = (\{q_0\}, q_0, \{q_0\}, \Delta := \emptyset)$
- si $E = a$, avec $a \in \Sigma$, alors $A_E = (\{q_0, q_1\}, q_0, \{q_1\}, \Delta := \{(q_0, a, q_1)\})$
- si $E = \emptyset$, alors $A_E = (\{q_0\}, q_0, \emptyset, \Delta := \emptyset)$

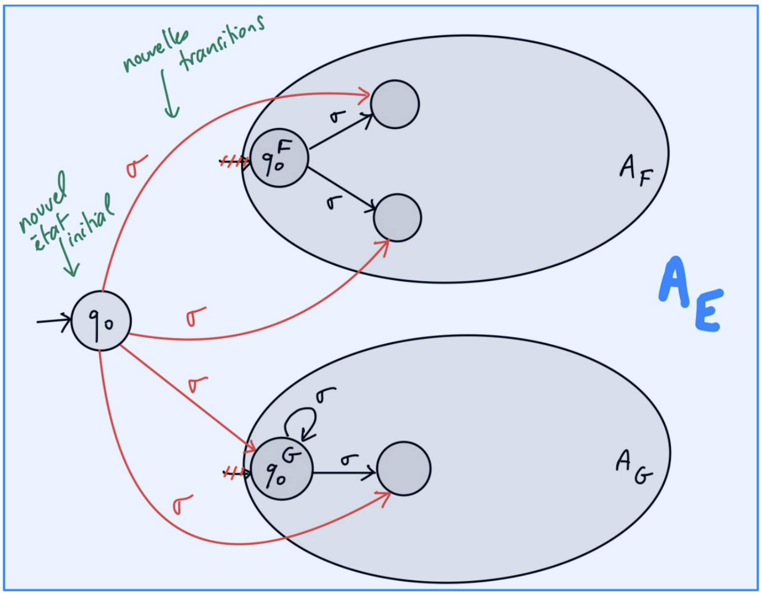
4.4.1.1 $E = F + G$

On construit par induction A_F et A_G, à partir desquels on construit A_E tel que :

$$L(A_E) = L(A_F) \cup L(A_G)$$

C'est la clôture par union (cf. Théorème 4.3) seulement, grâce au non-déterminisme, on peut faire plus simple :

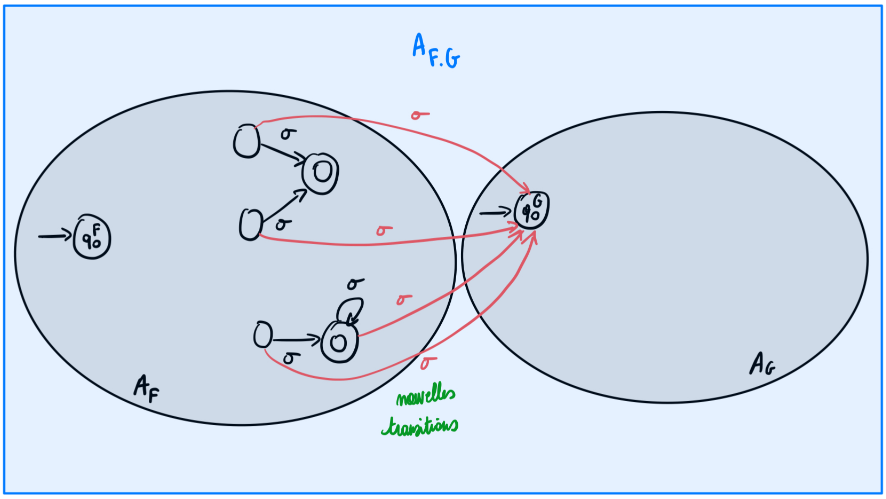
- en lisant la première lettre σ, on va soit dans le premier automate soit dans le deuxième, en utilisant le non-déterminisme.
- précisément, A_E a un état initial q_0. On note q_0^F l'état initial de A_F et q_0^G l'état initial de A_G. Pour toute lettre σ, pour toute transition $(q_0^F, \sigma, q) \in \Delta_F$, on ajoute la transition (q_0, σ, q) à Δ_E. De même pour A_G.
- On rend le nouvel état initial acceptant si q_0^F ou q_0^G est acceptant. (pour accepter le mot vide)



4.4.1.2 $E = FG$

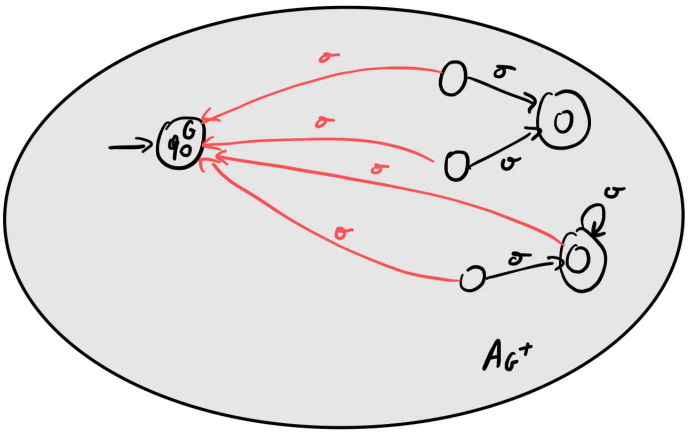
On construit A_F et A_G par induction, puis pour toute lettre σ, pour tout état de A_F qui allait vers un état final de A_F en lisant σ, on ajoute une transition vers l'état initial de A_G.

⚠ Les états acceptants de A_F ne sont plus acceptants dans A_E et l'état inital de A_E est l'état initial de A_F.



4.4.1.3 $E = G^*$

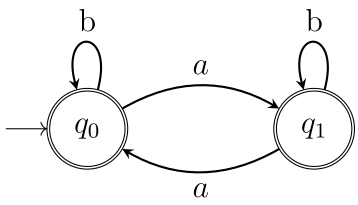
Il faut faire attention au fait que $\epsilon \in L(E)$. On réécrit d'abord E comme $E = \epsilon + G^+$ où G^+ est l'ensemble des mots de la forme $u_1 u_2 \dots u_k$ avec $k \geq 1$ et $u_i \in L(G)$ pour tout i.
Ensuite, on constrit A_ϵ et A_G par induction. On montre maintenant comment obtenir A_{G^+} et enfin A_E sera obtenu en utilisant la clôture par union de A_ϵ et A_{G^+}. Pour construire A_{G^+}, c'est similaire à la concaténation sauf qu'on ajoute des transitions vers l'état initial de G :

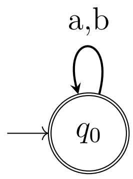


4.5 Minimisation

Étant donné un automate A, l'objectif de la minimisation est de construire un automate complet M tel que M a un nombre d'états minimal et M est équivalent à A. (cf. Définition 4.10)

Exemple: Voici un exemple d'un automate et de son automate minimal équivalent :





Définition 4.16. Sémantique

Soit $A = (Q, q_0, F, \delta)$ un automate complet sur un alphabet Σ. Pour tout mot $u \in \Sigma^*$, pour tout état $q \in Q$, on note :

- $q \cdot u$ l'état atteint à partir de q en lisant u. (il existe car A est complet)
- L_q le langage formé des mots u tels que $q \cdot u \in F$.
$$L_q = \{u \in \Sigma^* | q \cdot u \in F\}$$
est le langage des mots acceptés à partir de q.
- pour tout $p, q \in Q, p \equiv_A q$ si $L_p = L_q$.

Remarque: $L_{q_0} = L$, $\epsilon \in L_q$ pour tout $q \in F$. $\equiv_A$ est une relation d'équivalence, on note $[p]_A$ la classe de tout état p.

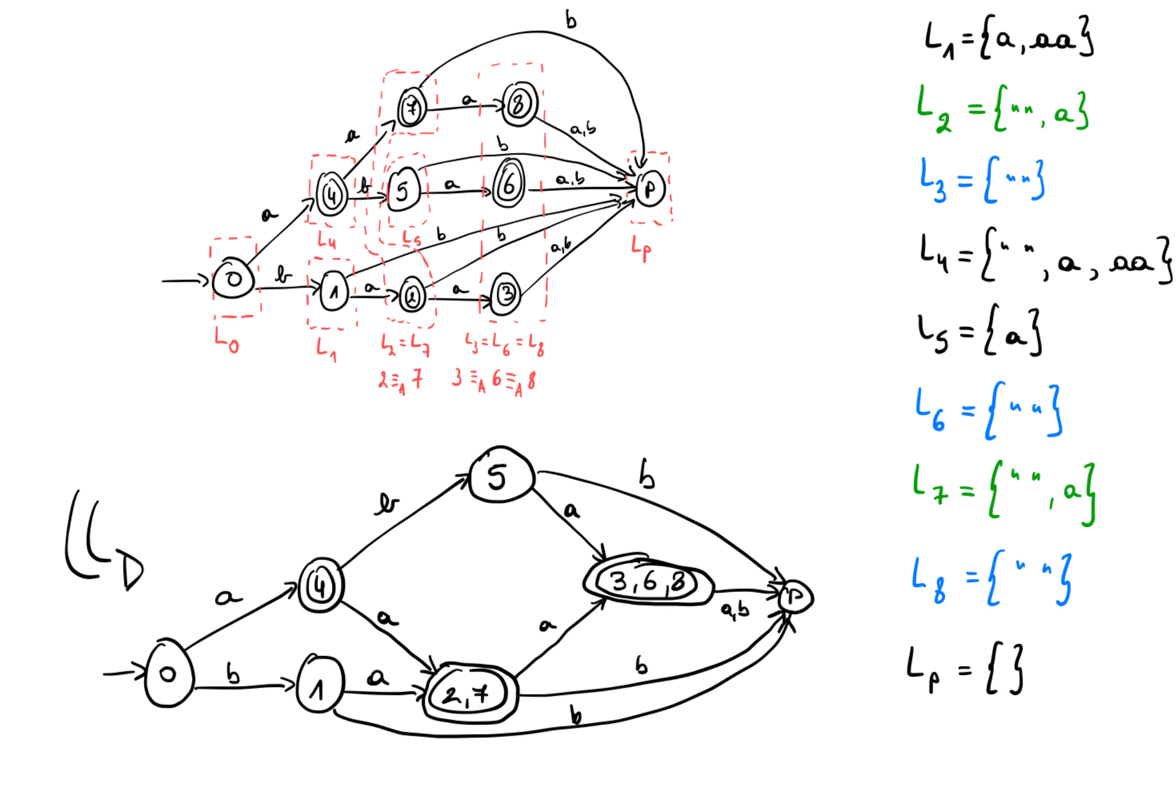
À partir d'un automate A, on calcule un automate M_A comme suit :

1. Il suffit de calculer toutes les classes d'équivalence de Q pour $\equiv_A$, ce qui donnera les états.
2. La classe de q_0 est l'état initial.
3. Toute classe qui contient un état final est finale.
4. On met une transition de l'état $[q] \equiv_A$ à l'état $[q \cdot a]_{\equiv_A}$ en lisant a, pour tout $a \in \Sigma$. et tout $q \in Q$.

Théorème 4.9. Automate minimal

L'automate minimal M_A est minimal en nombre d'états et il est l'unique automate minimal complet qui accepte $L(A)$.

Exemple: Voici un exemple de calcul de l'automate minimal :



En pratique, ce n'est pas très efficace, on peut faire mieux, sans passer par le test d'équivalence et par raffinement successif de relations d'équivalence qui convergent vers $\equiv_A$. Cela est même possible en $O(n \cdot log_2(n))$ où $n = |Q|$. Pour les AFNs, le problème d'optimisation est plus compliqué, décider si un AFN est équivalent à un AFN avec au plus k états est PSPACE-dur.

5 Introduction à la théorie de la complexité

5.1 Classes $\mathcal{P}$ et $\mathcal{NP}$

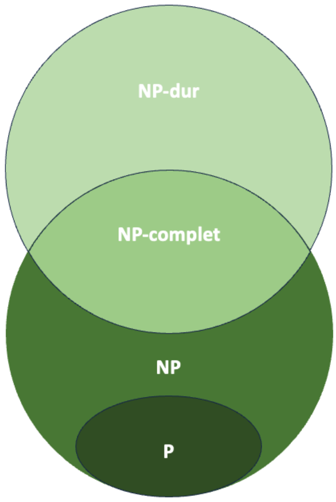


FIGURE 5.1 – Classes de complexité

5.1.1 Problèmes de décision

Définition 5.1. Problème de décision

Un problème de décision est un langage $P \subseteq \Sigma^*$.

Remarque: Chaque langage P représente un problème dont la réponse est oui ou non, en l'identifiant à sa fonction caractéristique χ_P :

$$\begin{aligned} \chi_P \ : \ & \Sigma^* \to \{0,1\} \\ & u \mapsto \begin{cases} 1 & \text{si } u \in P \\ 0 & \text{sinon.} \end{cases} \end{aligned}$$

Étant donné un mot $u \in \Sigma^*$, il faut décider si $u \in P$ ou non.

Exemple: Avec $\Sigma = \{0,1\}$,

$$\text{PRIME} = \{10, 11, 101, 111, \dots\}$$

l'ensemble des nombres premiers en binaire.

Il est parfois compliqué de représenter un problème comme un langage, effectivement, comment faire pour :

- ENTRÉE : un graphe G non dirigé et un entier $k \in \mathbb{N}$.
- SORTIE : 1 ssi on peut colorier G avec k couleurs.

On pourrait représenter ceci avec l'alphabet $\Sigma = \{0, 1, \#, \$\}$ avec (i, j) chaque arête qui pourrait être codée par le mot $\overline{i}\#\overline{j}$ où $\overline{i}$ est le codage binaire du sommet i. La paire (G, k) pourrait se coder par le mot :

$$\overline{i_1}\#\overline{j_1}\$\overline{i_2}\#\overline{j_2}\$ \cdots \$\overline{i_m}\#\overline{j_m}\$\overline{k}$$

> Le codage peut influencer la complexité. Il sera parfois nécessaire de changer le codage pour obtenir une complexité plus faible.

5.1.2 Problème d'optimisation

- Un problème d'optimisation est un problème où l'on souhaite maximiser/minimiser une certaine quantité.
  $\rightarrow$ Trouver la longueur d'un plus court chemin entre deux sommets (s et t) d'un graphe.
- Un problème d'optimisation peut être associé à un problème de décision.
  $\rightarrow$ Existe-t-il un chemin de longueur au plus k de s à t ?
- Si on sait résoudre le problème de décision, on peut parfois résoudre le problème d'optimisation.
  $\rightarrow$ Pour trouver le plus court chemin de s à t dans un graphe à n sommets, on peut faire une recherche dichotomique.

5.1.3 Algorithme de décision

Définition 5.2. Algorithme de décision

Un problème $P \subseteq \Sigma^*$ est décidé par un algorithme A si pour tout mot $u \in \Sigma^*$;
- A se termine et retourne 1 si $u \in P$.
- A se termine et retourne 0 si $u \notin P$.

Un problème est décidable s'il existe un algorithme qui le décide.

5.1.4 La classe $\mathcal{P}$

Définition 5.3. Classe $\mathcal{P}$

La classe $\mathcal{P}$ est la classe des problèmes pouvant être **décidés** en temps polynomial. Plus précisément, un problème $P \subseteq \Sigma^*$ est dans $\mathcal{P}$ s'il existe un algorithme A et une constante k tel que pour tout mot u de longueur n,
- A retourne 1 en temps $O(n^k)$ si $u \in P$.
- A retourne 0 en temps $O(n^k)$ si $u \notin P$.

Exemple: Exemples de problèmes dans $\mathcal{P}$:
- décider si un tableau est trié.
- décider si un entier codé en unaire est premier (facile)
- décider si un entier codé en binaire est premier (difficile)

5.1.5 Algorithme de vérification

Un algorithme de vérification est un algorithme qui sur base d'une solution candidate à un problème de décision, décide si cette solution est valide ou non.

Définition 5.4. Algorithme de vérification

Un algorithme de vérification pour un problème $P \subseteq \Sigma^*$ est un algorithme de décision (retournant 1 ou 0) A prenant deux mots en argument, qui termine pour toute entrée, tel que :

$$P = \{u \in \Sigma^* | \exists v \in \Sigma^*, A(u, v) = 1\}$$

Lorsque $A(u, v) = 1$, v est appelé un certificat pour u.

5.1.6 La classe $\mathcal{NP}$

Définition 5.5. Classe $\mathcal{NP}$

La classe $\mathcal{NP}$ est la classe des problèmes pouvant être **vérifiés** en temps polynomial par un algorithme de vérification. Plus précisément, un problème P est dans $\mathcal{NP}$ s'il existe un algorithme de vérification A de complexité polynomiale en temps et une constante k, tels que pour toute entrée u, les deux affirmations suivantes sont équivalentes :

1. $u \in P$
2. il existe un certificat v de longueur polynomiale dans u (i.e. $|v| = O(|u|^k)$) tel que $A(u, v) = 1$.

Exemple: Exemples de problèmes dans $\mathcal{NP}$:

- décider qu'un ensemble de clauses est satisfaisable.
- voyageur de commerce : étant donné n villes, les distances entre les villes et un entier d, on voudrait savoir s'il existe un cycle de longueur $\leq d$ passant par toutes les villes une et une seule fois
- coloriage de graphes : peut-on choisiir un graphe avec moins de k couleurs sans que deux sommets aient la même couleur ?
- tous les problèmes de la classe $\mathcal{P}$ sont dans $\mathcal{NP}$, $P \subseteq NP$.

Remarque: Tout problème $\mathcal{NP}$ peut être décidé par un algorithme de complexité exponentielle en temps car étant donné un mot u en entrée, il suffit d'énumérer tous les certificats de longueur au plus $a \cdot |u|^k$ et d'appeler l'algorithme de vérification. Pour SAT, par exemple, il suffit d'énumérer toutes les interprétations possibles et les tester, pour le coloriage, il suffit dénumérer tous les coloriages possibles et les tester.
Il existe une **fameuse conjecture** en théorie de la complexité qui est la suivante :

$$\mathcal{P} \neq \mathcal{NP}$$

Autrement dit, on ne sait pas encore démontrer que vérifier une solution candidate en temps polynomial est plus puissant que trouver une solution en temps polynomial.

5.1.7 Temps non-déterministe polynomial

La classe $\mathcal{NP}$ peut également être définie comme la classe des problèmes pouvant être décidés en temps polynomial par un algorithme non-déterministe. Si la réponse au problème est oui, alors il existe une exécution de l'algorithme après laquelle l'algorithme répondra oui.

Exemple: SAT, choisir aléatoirement une valeur de vérité pour chaque variable et vérifier en temps linéaire qu'elles satisfont la formule.

5.1.8 Réductions, $\mathcal{NP}$-dureté et $\mathcal{NP}$-complétude

Définition 5.6. $\mathcal{NP}$-dur

Un problème de décision P est $\mathcal{NP}$-dur si pour tout problème P' de $\mathcal{NP}$ se réduit à P en temps polynomial, i.e. qu'il existe un algorithme T de complexité polynomiale en temps, qui transforme tout mot u' en un mot $T(u')$ tel que $u' \in P'$ ssi $T(u') \in P$.

Un problème qui est dans $\mathcal{NP}$ et $\mathcal{NP}$-dur est dit $\mathcal{NP}$-complet, ce sont les plus compliqués de la classe $\mathcal{NP}$.

> Un problème P peut être dans $\mathcal{NP}$-dur sans être dans $\mathcal{NP}$.

Pour démontrer que P est dans $\mathcal{NP}$-complet, il faut :

1. Démontrer qu'il est dans $\mathcal{NP}$.
2. Démontrer qu'il est dans $\mathcal{NP}$-dur. Pour démontrer qu'un problème P est $\mathcal{NP}$-dur, il faut partir d'un problème connu comme étant $\mathcal{NP}$-dur, qu'on réduit dans notre problème en temps polynomial. Le premier problème $\mathcal{NP}$-dur est le **Théorème de Cook**.

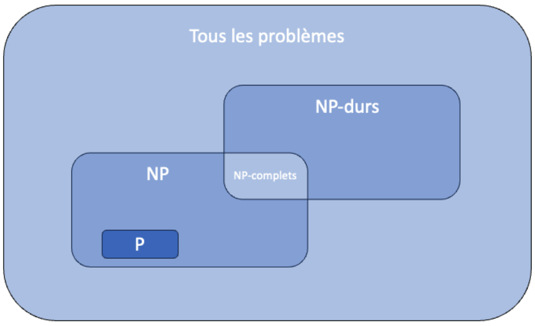


FIGURE 5.2 – Relations entre les classes de complexité

Théorème 5.1 Si $\mathcal{P} \neq \mathcal{NP}$ et un problème A est $\mathcal{NP}$-complet, alors $A \notin \mathcal{P}$.

Démonstration: On va supposer que $A \in \mathcal{P}$ et en déduire la contradiction $\mathcal{P} = \mathcal{NP}$. On sait que $\mathcal{P} \subseteq \mathcal{NP}$, il nous reste à démontrer que $\mathcal{NP} \subseteq \mathcal{P}$. Soit B un problème de $\mathcal{NP}$ quelconque, montrons qu'il est dans $\mathcal{P}$.
Comme A est $\mathcal{NP}$-complet, B se réduit à A en temps polynomial. Comme $A \in \mathcal{P}$, nous avons un algorithme en temps polynomial pour résoudre B :

- ENTRÉE : Instance I de B
  1. Transformer I en une instance I' de A en temps polynomial. $O(n^c)$
  2. Résoudre I' en temps polynomial. $O(n^d)$

Supposons que la réduction se fasse en temps $O(n^c)$ pour une constante c et que A se résout en temps $O(n^d)$ pour une constante d. Alors B se résout en temps $O(n^{cd})$:

1. Puisque l'étape 1 se fait en temps $O(n^c)$, où n est la taille de I, la taille de I' est en $O(n^c)$.
2. L'étape 2 est appliquée sur une entrée de taille $O(n^c)$, donc elle prend un temps $O((n^c)^d) = O(n^{cd})$.

😎

Lemme 5.1 Si vous prouvez qu'un problème A est dans $\mathcal{P}$ et est $\mathcal{NP}$-complet, soit vous avez démontré $\mathcal{P} = \mathcal{NP}$ (et vous pouvez réclamer le million), soit vous avez fait une erreur.

Lemme 5.2 Si vous ne trouvez pas d'algorithme en temps polynomial pour votre problème, alors peut-être qu'il est $\mathcal{NP}$-complet. Dans ce cas, vous avez peu de chances d'en trouver un.

Exemple: Voici quelques exemples de problèmes $\mathcal{NP}$-complets :
- SAT
- 3-SAT
- voyageur de commerce
- coloriage de graphes
- bin-packing

5.1.9 Réduction de SAT vers 3-SAT

Théorème 5.2. 3SAT
3SAT est $\mathcal{NP}$-complet.

Démonstration:

(1) On doit démontrer que 3-SAT est dans $\mathcal{NP}$. On sait que SAT est dans $\mathcal{NP}$ et comme 3-SAT est Un cas particulier de SAT, on obtient que 3-SAT est dans $\mathcal{NP}$.

(2) On doit démontrer que 3-SAT est dans $\mathcal{NP}$-dur. Pour ce faire, on démontre que SAT se réduit à 3-SAT en temps polynomial car on sait que SAT est $\mathcal{NP}$-dur. On doit construire un algorithme R de complexité polynomiale en temps qui transforme toute instance I de SAT en une instance $R(I)$ de 3-SAT telle que I est satisfaisable ssi $R(I)$ est satisfaisable.

Exemple: Commençons par un exemple :

$$(x_0 \vee x_1 \vee \neg x_2 \vee x_3 \vee x_4) \wedge (x_1 \vee \neg x_5 \vee x_7)$$

Le premier terme n'est pas un 3-clause, on va donc le transformer en 3-clauses en ajoutant des variables intermédiaires.

$$(x_0 \vee x_1 \vee y) \wedge (\neg y \vee \neg x_2 \vee x_3 \vee x_4)$$

Le premier terme est maintenant en 3-clauses, mais le deuxième ne l'est pas. On va donc le transformer en 3-clauses :

$$(\neg y \vee \neg x_2 \vee z) \wedge (\neg z \vee x_3 \vee x_4)$$

Voilà, on a tout transformé en 3-clauses. On peut maintenant écrire notre instance de 3-SAT :

$$(x_0 \vee x_1 \vee y) \wedge (\neg y \vee \neg x_2 \vee z) \wedge (\neg z \vee x_3 \vee x_4) \wedge (x_1 \vee \neg x_5 \vee x_7)$$

Maintenat généralisons :
- Toute clause $C = l_1 \vee l_2 \vee \cdots \vee l_k$ est transformée en une conjonction de clauses $T(C)$ comme suit :
  - si $k = 1$: $T(C) = l_1 \vee l_1 \vee l_1$
  - si $k = 2$: $T(C) = l_1 \vee l_2 \vee l_2$
  - si $k = 3$: $T(C) = C$
  - si $k > 3$: $T(C) = (l_1 \vee l_2 \vee y) \wedge T(\neg y \vee l_3 \vee \cdots \vee l_k)$
    On applique récursivement sur le deuxième terme.
- Pour transformer un ensemble de clauses $I = C_1 \wedge C_2 \wedge \cdots \wedge C_m$, on applique T sur chaque clause en prenant soin de ne jamais introduire des variables qui ont déjà été introduites.

(3) Démontrer que S est satisfaisable ssi $T(S, X)$ l'est. TODO

(4) Démontrer que la réduction est en temps polynomial. TODO

😎

5.1.10 Bin Packing

Le problème du Bin Packing est le suivant : étant donné n objets de poids $p_1, ..., p_n$, peut-on les ranger dans k boîtes contenant au max C kg ?

Ce problème est $\mathcal{NP}$-complet. Effectivement :

1. BinPacking est dans $\mathcal{NP}$: on peut vérifier en temps polynomial (une assignation de chaque objet à un numéro de sac entre 1 et k) si une solution candidate satisfait la contrainte de capacité.
2. BinPacking est $\mathcal{NP}$-dur : Ceci est démontrable en faisant la réduction en temps polynomial du problème 2-partition, qui est un problème $\mathcal{NP}$-complet.

   Démonstration:

   → **Instance de 2-partition** : n entiers $c_1, ..., c_n$. On veut savoir s'il existe $S \subseteq \{1, ..., n\}$ tel que $\sum_{i \in S} c_i = \sum_{i \notin S} c_i$. Soit $S = \sum_{i=1}^{n} c_i$. On peut supposer que S est paire sinon il n'y a pas de solution.

   → **Réduction vers BinPacking** : On prend $C = S/2$ et les objets $1, \cdots, n$ de poids $c - 1, \cdots, c_n$ et $k = 2$. Il y a une solution au problème 2-partition ssi il y a une solution au problème BinPacking. De plus, on peut construire cette instance de BinPacking en temps polynomial.

   😎

3. BinPacking est $\mathcal{NP}$-complet : on a montré qu'il est dans $\mathcal{NP}$ et $\mathcal{NP}$-dur.

Il est également possible de montrer que GRAPHCOLOR est $\mathcal{NP}$-complet en réduisant le problème 3-SAT vers GRAPHCOLOR en temps polynomial, pour voir ceci, voir le cours. (slides 25 jusqu'à 34)

Remarque: Il existe également d'autres classes de complexité comme PSPACE, EXPTIME, NPSPACE... Elles respectent les inclusions suivantes :

$$\mathcal{P} \subseteq \mathcal{NP} \subseteq \text{PSPACE} \subseteq \text{EXPTIME} \subseteq \text{NEXPTIME} \subseteq \cdots \subseteq \text{décidables}$$

5.2 Problèmes indécidables

Définition 5.7. Problème indécidable

Un problème de décision P sur un alphabet Σ est indécidable s'il n'existe pas d'algorithme A prenant un mot sur Σ et retournant, en un nombre fini d'étapes de calcul, la valeur 0 ou 1, tel que pour tout mot x sur Σ, $A(x) = 1$ ssi $x \in P$.
En d'autres termes, un problème est indécidable s'il n'existe pas d'algorithme pour le résoudre.

5.2.1 Problème de l'arrêt

```
while(True): print("hello")
```

Ce programme ne s'arrête jamais.

Définition 5.8. Problème de l'arrêt

- ENTRÉE : le code source d'un programme P lisant des chaînes de caractères et une chaîne de caractères x.
- SORTIE : 1 ssi P s'arrête sur l'entrée x, 0 sinon.

Démonstration: On procède par l'absurde en supposant qu'il existe un programme HALT(c_p, x) qui décide le problème de l'arrêt, pour tout programme P donné par son code c_p et toute chaîne de caractères x.
À partir de HALT, on définit le programme PARADOX suivant :

```
def PARADOX(c:string):
    if HALT(c,c): loop forever
    else: stop
```

On appelle PARADOX(c_{PARADOX}), où c_{PARADOX} est le code source de PARADOX. On a deux cas :

1. Si PARADOX($c_{PARADOX}$) s'arrête, alors c'est que HALT$(c_{PARADOX}, c_{PARADOX}) = 0$, donc PARADOX($c_{PARADOX}$) boucle infiniment.
2. Si PARADOX($c_{PARADOX}$) ne s'arrête pas, alors c'est que HALT$(c_{PARADOX}, c_{PARADOX}) = 1$, donc PARADOX($c_{PARADOX}$) s'arrête.

Dans les deux cas, on obtient une contradiction, donc le programme HALT ne peut pas exister. 😎

5.2.2 Problème de la Correspondance de Post

Soit $\Sigma = \{0, 1\}$. Un mot u sur Σ est une séquence finie d'éléments de Σ. Deux mots u et v peuvent être concaténés pour former un nouveau mot uv. L'élément neutre pour la concaténation est le mot vide ϵ.
Par exemple : $u = 01$ est un mot, $v = 110$ est un mot, $uv = 01110$, $\epsilon u = u\epsilon = u = 01$.

Définition 5.9. Problème de la Correspondance de Post

- ENTRÉE : $(u_1, v_1), \cdots, (u_n, v_n), n \geq 1, n$ paires de mots (possiblement vides) sur Σ.
- SORTIE : 1 ssi il existe une séquence finie d'indices $i_1, \cdots, i_k \in \{1, \cdots, n\}$ telle que $k \geq 1$ et
$$u_{i_1} \cdots u_{i_k} = v_{i_1} \cdots v_{i_k}$$

Exemple: Voici quelques exemples de ce problème :

- $(u_1, v_1) = (100, 00)$, $(u_2, v_2) = (0, 01)$, Solution (parmi d'autres) : $2, 1$
$$u_2u_1 = 0100 = v_2v_1$$
- $(u_1, v_1) = (1, 100)$, $(u_2, v_2) = (0, \epsilon)$, Solution : $1, 2, 2$
$$u_1u_2u_2 = 100 = v_1v_2v_2$$
- $(u_1, v_1) = (1, 0)$, $(u_2, v_2) = (0, 1)$, Solution (parmi d'autres) : Pas de solution

Remarque: Résultat admis.

5.2.3 Prouver l'indécidabilité d'un problème

- Pour montrer l'indécidabilité d'un problème de décision P_1, on peut partir d'un problème de décision P_2 connu indécidable et on montre l'existence d'un algorithme (réduction) qui pour toute instance I_2 de P_2 construit une instance I_1 de P_1 telle que I_1 a une solution ssi I_2 a une solution.

- De cette façon, on montre que P_1 est "aussi difficile" que P_2. S'il existait un algorithme pour résoudre P_1, alors cela nous donnerait un algorithme pour résoudre P_2 : appliquer la réduction puis l'algorithme pour P_1. Cela contredirait le fait que P_2 soit indécidable, donc il n'existe pas d'algorithme pour résoudre P_1.

Remarque: La notion de réduction est la même que pour démontrer la NP-dureté d'un problème, à la seule différence qu'on ne demande pas que l'algorithme qui calcule la réduction se fasse en temps polynomial.

6 Logique des prédicats

6.1 Introduction

En logique des prédicats ;

- on ajoute les quantificateurs.
- on généralise les valeurs que peuvent prendre les variables.
- on ajoute des relations poiur décrire certianes relation entre ces valeurs.
- on ajoute des symboles de fonctions à la syntaxe.

Exemple: $\forall x \forall y \cdot \text{PremierEntreEux}(x, y) \leftrightarrow \exists x' \exists y' \cdot x.x' + y.y' = 1$

- PremierEntreEux est un prédicat à deux arguments.
- 1 est appelée constante.
- $xx' + yy'$ est un terme formé avec les fonctions $\times$ et .

6.2 Syntaxe

6.2.1 Alphabet

L'alphabet d'un langage du premier ordre comporte d'abord les symboles suivants qui sont communs à tous ces langages :

- les connecteurs : $\neg, \wedge, \vee, \rightarrow \leftrightarrow$;
- les paranthèses : (,) ;
- le quantificateur universel $\forall$ et le quantificateur existentiel $\exists$;
- un ensemble infini V de symboles de variables $x, y, z, \ldots$;

Définition 6.1. Langage de la logique du premier ordre

Un langage $\mathcal{L}$ de la logique du premier ordre est caractérisé par :

- des symboles de relations (prédicats), notés $p, q, r, s, \ldots$;
- des symboles de fonctions, notés $f, g, h, \ldots$;
- des symboles de constantes, notés $c, d, e, \ldots$;

À chaque prédicat p, respectivement fonction f, on associe un entier strictement positif appelé l'arité de p, respectivement de f, càd le nombre d'arguments de p, respectivement f. On notera parfois $p|_n$ et $f|_n$ pour indiquer que p (respectivement f) est un symbole de relation (respectivement de fonction) d'arité n.
Le prédicat "=" sera toujours présent.

Exemple: Exemples de langages :

- $\mathcal{L}_1 = \{r|_1, c\}$ contient un prédicat unaire r et une constante c ;
- $\mathcal{L}_2 = \{r|_2, f|_1, g|_2, h|_2, c, d\}$ contient un prédicat binaire r, une fonction unaire f, deux symboles de fonctions binaires g et h et deux constantes c et d.

6.2.2 Construction des termes

Définition 6.2. Termes d'un langage

L'ensemble des termes d'un langage $\mathcal{L}$ est le plus petit ensemble qui contient les symboles de constantes et de variables et qui est clos par application des fonction.
L'ensemble des termes, noté $\mathcal{T}$, est le plus petit ensemble satisfaisant :

1. Tout symbole de constante ou variable est un terme.
2. Si f est un symbole de fonction d'arité n et $t_1, t_2, \cdots, t_n$ sont des termes alors $f(t_1, t_2, \cdots, t_n)$ est un terme. est un terme.

Exemple: Voici des exemples :

- Les seuls termes du langages $\mathcal{L}_1$ sont la constante c et les variables.
- Les expressions suivantes sont des termes du langage $\mathcal{L}_2$:
  - $f(c)$
  - $f(h(f(c), d))$
  - $f(y)$
  - $f(h(f(x), f(d)))$

Un terme est **clos** s'il est sans variable. Ici, $f(c)$ est clos.

6.2.3 Construction des formules

Définition 6.3. Formules d'un langage

L'ensemble des formules du langage $\mathcal{L}$, que l'on désigne par $\mathcal{F}(\mathcal{L})$, est défini par la grammaire suivante :

$$\phi ::= p(t_1, \cdots, t_n) | \phi \wedge \phi | \phi \vee \phi | \neg \phi | \phi \rightarrow \phi | \phi \leftrightarrow \phi | \exists x \cdot \phi | \forall x \cdot \phi | (\phi)$$

- $t_1, \cdots, t_n$ sont des termes ;
- p est un symbole de relation ;
- $\exists x$ est le quantificateur existentiel ;
- $\forall x$ est le quantificateur universel ;

Exemple: Voici des exemples :

- La formule $r(c) \vee \neg \exists x \cdot r(x)$ est une formule du langage $\mathcal{L}_1$.
- Exemples de formules du langage $\mathcal{L}_2$:
  - $\forall x \cdot \exists y (g(x, y) = c \wedge g(x, y) = c)$
  - $\forall x \cdot \neg(f(x) = c)$

6.2.4 Règles de précédence

Pour les Booléens, ce sont les mêmes règles de précédence que dans la logique propositionnelle (cf. 1). Les quantificateurs ont la même priorité que la négation.

$$\forall x \cdot \neg p(x, y) \vee p(y, x) \equiv (\forall x \cdot \neg (p(x, y))) \vee p(y, x)$$

6.2.5 Variables libres et liées

Définition 6.4. Occurence de variable

Une occurence d'une variable dans une formule est un couple constitué de cette variable et d'une place effective, càd qui ne suit pas un quantificateur.

Exemple: Dans la formule :

$$r(x,z) \to \forall z \cdot (r(y,z) \vee y = z)$$

La varible x possède une occurence, la variable y deux et la variable z trois.

Définition 6.5. Variables libres ou liées

- Une occurence d'une variable x dans une formule ϕ est une occurence **libre** si elle ne se trouve dans aucune sous-formule de ϕ, qui commence par une quantification $\forall x$ ou $\exists x$.
- Dans le cas contraire, l'occurence est dite **liée**.
- Une variable est libre dans une formule si elle a au moins une occurence libre dans cette formule.
- Une formule est close est une formule sans variable libre.
- On note Libres(ϕ) l'ensemble des variables libres de ϕ.

Exemple: Dans $\exists x \cdot p(x,y)$, l'occurence de x est liée et celle de y est libre.

6.3 Sémantique

6.3.1 Interprétation des formules

Définition 6.6. Structure

Une structure $\mathcal{M}$ pour un langage $\mathcal{L}$ se compose d'un ensemble non vide M appelé le domaine et d'une interprétation des symboles de prédicats par des relations sur M, des symboles de fonctions par des fonctions de M et des constantes par des éléments de M.
Plus précisément, une structure est composée de :

- d'un sous-ensemble de M^n, noté $p^{\mathcal{M}}$, pour chaque symbole de prédicat r d'arité n dans $\mathcal{L}$;
- d'une fonction **totale** de M^m dans M, notée $f^{\mathcal{M}}$, pour chaque symbole de fonction f d'arité m dans $\mathcal{L}$;
- d'un élément de M, noté $c^{\mathcal{M}}$, pour chaque symbole de constante c dans $\mathcal{L}$.

Exemple:
- Pour le langage $\mathcal{L}_1 = (r|_1 c)$, la structure $\mathcal{M}_1 = (\mathbb{N}, r^{\mathcal{M}_1}, c^{\Updownarrow_1})$ avec $r^{\mathcal{M}_1}$ l'ensemble des nombres premiers et $c^{\mathcal{M}_1} = 2$ est une interprétation de $\mathcal{L}_1$.
- Pour le langage $\mathcal{L}_2 = (r|_2, f|_1, g|_2, h|_2, c, d)$, on peut prendre la structure sur les réels :
$$\mathcal{M}_2 = (\mathbb{R}, \leq, +1, +, \times, 0, 1)$$
avec la fonction $+1$ qui à x associe $x+1$.

Théorème 6.1. Formule satisfaite

Une formule ϕ construite sur un langage $\mathcal{L}$ est satisfaite dans une structure $\mathcal{M}$ et pour une valuation v donnanat une valeur aux variables de l'ensemble $\mathcal{V}$ (noté $\mathcal{M}, v \vDash \emptyset$) ssi :

- si $\phi \equiv r(t_1, t_2, \cdots, t_n)$ et $t_i^{\mathcal{M},v} = b_i$ pour $i = 1, \cdots, n$, alors ϕ est vraie ssi

$(b_1, b_2, \cdots, b_n) \in r^{\mathcal{M}}$;

- si $\phi \equiv \neg\psi_1, \phi \equiv \psi_1 \vee \psi_2, \phi \equiv \psi_1 \wedge \psi 2, \phi \equiv \psi_1 \to \psi_2, \phi \equiv \psi_1 \leftrightarrow \psi_2$ alors la valeur de ϕ est calculée àpd valeurs de psi_1 et psi_2 comme dans le cas propositionnel.
- si $\phi \equiv \exists x \cdot \psi$, alors ϕ est vraie ssi **il existe** une valuation v' telle que $\mathcal{M}, v' \vDash \psi$ et v' est d'accord ($=v'(x) = v(x)$) avec v sur Libres(phi).
- si $\phi \equiv \forall x \cdot \psi$, alors ϕ est vraie ssi **pour toute** valuation v' qui est d'accord avec v sur Libres(ϕ), on a $\mathcal{M}, v' \vDash \psi$.

Lorsque $\mathcal{M}, v \vDash \emptyset$, on dit que $\mathcal{M}, v$ satisfait ϕ ou encore que $(\mathcal{M}, v)$ est un modèle de ϕ. De pus, lorsque ϕ est une formule close, alors sa valeur de vérité dans un couple $(\mathcal{M}, v)$, ne dépend pas de v. On omettera de mentionner v dans ce cas.

Exemple:

- Prenons $\mathcal{L}_1 = \{r|_2, c\}$. La formule close suivante :

$$\begin{aligned} &\forall x \cdot r(x,x) \\ &\wedge \forall x \cdot \forall y \cdot (r(x,y) \to r(y,x)) \\ &\wedge \forall x \cdot \forall y \cdot \forall z \cdot (r(x,y) \wedge r(y,z) \to r(x,z)) \end{aligned}$$

exprime qu'une structure (D, R, a) est un modèle de la formule ssi R est une relation d'équivalence.

- est-ce que $\exists x \cdot \forall y \cdot r(x,y)$ est vraie dans $(\mathbb{N}, \leq)$?
"Est-ce qu'il existe un entier $x \in \mathbb{N}$ tel que $\forall y \in \mathbb{N}$, $x \leq y$?" oui, $x = 0$.

6.3.2 Interprétation des termes dans une structure

Définition 6.7. Valuation

Etant donnés un ensemble de variables $\mathcal{V}$ et un domaine M, une *valuation* pour les variables de $\mathcal{V}$ dans M est une fonction $v : \mathcal{V} \to M$ qui attribue à chaque variable $x \in \mathcal{V}$, une valeur $v(x) \in M$.

Définition 6.8. Interprétation de termes

L'interprétation d'un terme t (dont les variables sont dans $\mathcal{V}$) dans une structure de domaine M et selon une valuation v est un élément $t^{\mathcal{M},v} \in M$, inductivement défini de la façon suivante :

- si $t = c$ alors $t^{\mathcal{M},v} = c^{\mathcal{M}}$;
- si $t = x$ alors $t^{\mathcal{M},v} = v(x)$ est $v(x)$;
- si $t = f(t_1, \cdots, t_n)$ alors $t^{\mathcal{M},v} = f^{\mathcal{M}}(t_1^{\mathcal{M},v}, \cdots, t_n^{\mathcal{M},v})$.

Exemple: Soit $\mathcal{L}_2 = (r|_2, f|_1, g|_2, h|_2, c, d)$ et $\mathcal{M}_3 = (\mathbb{N}, \leq, +1, +, \times, 0, 1)$.
L'interprétation dans $\mathcal{M}_3$ du terme

$$t_1 \equiv g(y, h(c, x))$$

selon la valuation v telle que $v(x) = 3, v(y) = 4, v(z) = 6$ est :

$$t_1^{\mathcal{M}_3,v} = 4 + (0 \times 3) = 4$$

L'interprétation du terme

$$t_2 \equiv f(g(d, h(y, z)))$$

est :

$$t_2^{\mathcal{M}_3,v} = 1 + (4 \times 6) + 1 = 26$$

6.3.3 Formules satisfaisables, valides et équivalentes

1. Une formule ϕ close est **satisfaisable** si elle un modèle. (il n'existe pas d'algo pour vérifier la satisfaisabilité d'une formule).
2. Une formule ϕ close est **valide** si toutes les structures sont des modèles de ϕ.
3. Deux formules ϕ_1, ϕ_2 telles que Libres(ϕ_1) = Libres(ϕ_2) sont **équivalentes** si la formule $\forall x_1 \cdots \forall x_n(\phi_1 \leftrightarrow \phi_2)$ est valide, avec $\{x_1, \cdots, x_n\}$=Libres(ϕ_1).

> **Exemple:** Voici quelques exemples de formules équivalentes :
> - $\forall x \cdot (\phi \land \psi)$ et $(\forall x \cdot \phi) \land (\forall x \cdot \psi)$
> - $\exists x \cdot (\phi \lor \psi)$ et $(\exists x \cdot \phi) \lor (\exists x \cdot \psi)$

6.3.4 Exemples de traductions de texte vers formules

- L'anglais vit dans la maison rouge : $\forall x[\text{anglais}(x) \to \text{rouge}(x)]$
- Le Suédois a des chiens : $\forall x[\text{suédois}(x) \to \text{chiens}(x)]$
- La maison verte est directement à gauche de la maison blanche :
  $\forall x \forall y(\text{succgauche}(x, y) \land \text{verte}(x)) \to \text{blanche}(y)$