INFO-F-302 - Informatique Fondamentale

Synthèse 2014-2015

Florentin Hennecker

Table des matières

1 Logique propositionnelle

1.1 Coupure

(Les notations sont plus précises dans le cours)

$$C_1 = p_1 \vee ... \vee p_{i-1} \vee p \vee p_{i+1} \vee ... \vee p_n$$
$$C_1 = s_1 \vee ... \vee s_{i-1} \vee \neg p \vee s_{i+1} \vee ... \vee s_n$$

Étant donné deux clauses C_1 et C_2 qui ont une proposition commune p positive dans C_1 et négative dans C_2, la règle de coupure permet de déduire la clause :

$$C_3 = p_1 \vee ... \vee p_{i-1} \vee p_{i+1} \vee ... \vee p_n \vee s_1 \vee ... \vee s_{i-1} \vee s_{i+1} \vee ... \vee s_n$$

On note par $C_1, C_2 \vdash_p^c C_3$ le fait que C_3 est déductible de C_1, C_2 par coupure sur la proposition p.

Exemple 1 $\neg e \vee b \vee p, \neg p \vee a \vee b \vdash_p^c \neg e \vee a \vee$

Exemple 2 $\neg p, p, \vdash_p^c \perp$ (clause vide)

Théorème 1.1 *Si $C_1, C_2 \vdash_p^c C_3$, alors C_3 est une conséquence logique de $C_1 \wedge C_2$*

1.1.1 Preuve par coupure

Une preuve par coupure est la déduction d'une clause à partir d'autres clauses uniquement en utilisant des coupures. On note par $S \vdash^c C$ le fait que C est déductible de S par coupure.

1.1.2 Réfutation

Une *réfutation* d'un ensemble S de clauses par coupure est une dérivation de la clause vide à partir de S.

Théorème 1.2 *Si il existe une réfutation de S par coupure alors l'ensemble de clauses S est non satisfaisable.*

2 Le problème SAT

Un problème SAT a en entrée un ensemble de clauses S et en sortie la réponse à la question : "Est-ce que S est satisfaisable ?". On aimerait que l'algorithme retourne une valutation V qui satisfait S au cas où S est satisfaisable. Un solveur Sat est un programme qui décide le problème SAT. Ces solveurs ont une complexité dans le pire cas exponentielle.

Motivations
- beaucoup de problèmes s'expriment naturellement par des formules en FNC.
- les problèmes de la classe NP se réduisent tous au problème SAT en temps polynomial
- on peut donc écrire un bon algo pour SAT plutôt qu'un bon algo pour chacun de ces problèmes

2.1 Variantes de SAT

2-SAT les clauses ne contiennent qu'au plus deux littéraux. **Solvable en temps polynomial**

QSAT décider la satisfaisabilité de formules de la forme $\forall p_1 \exists q_1 ... \forall p_n \exists q_n . \phi$ où ϕ est une formule en CNF construite sur les propositions $p1, q_1, ...p_n, q_n$

MAX-SAT étant donnée une formule ϕ en CNF et un entier $k \in \mathbb{N}$, peut-on satisfaire au moins k clauses ?

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.

2.2 Interprétation partielle

Une *interprétation partielle* est un assignement noté $x/1$ ou $x/0$, qui signifie qu'on assigne la valeur 1 à x ou la valeur 0. Cela permet de simplifier les formules.

Exemple 1 : la formule $(x \vee y) \wedge (\neg x \vee z \vee \neg y)$ se simplifie en $z \vee \neg y$ sous l'interprétation partielle $x/1$.

On peut appliquer deux interprétations partielles à la formule ϕ, ce qui se note ainsi : $\phi[x/b][y/b']$

Exemple 2 Soit $a = x \vee y \vee z$ et $b = x \vee \neg y \vee \neg z$. Alors :
- $a[x/1] = \top$, $b[x/1] = \top$, $(a \wedge b)[x/1] = \top$
- $a[x/0] = y \vee z$
- $b[x/0] = \neg y \vee \neg z$
- $(a \wedge b)[x/0][y/0] = z$
- ...

2.3 Proposition pivot

L'algorithme DPLL va essayer des interprétations partielles, la proposition choisie est la *proposition pivot*. Il va successivement essayer de mettre la proposition est vrai ou à faux, et tester récursivement la satisfaisabilité de formules simplifiées obtenues. Le choix de la proposition pivot peut évidemment fortement influencer le résultat.

2.3.1 Premier critère de choix

DPLL choisit en priorité la proposition d'une **clause unitaire** (un seul littéral) comme proposition pivot.

Exemple Dans $x \wedge (y \vee \neg z)$, x est unitaire et doit obligatoirement être interprété par 1 pour satisfaire la formule.

Exemple de propagation de clauses unitaires Prenons

$$\phi = (x \vee y) \wedge \neg y \wedge (\neg x \vee y \vee \neg z)$$

Alors

$$\phi[y/0] = x \wedge (\neg x \vee \neg z)$$

Ce qui donne la nouvelle clause unitaire x qui impose $x = 1$:

$$\phi[y/0][x/1] = \neg z$$

Ce qui ne contient qu'une seule clause unitaire et on obtient finalement :

$$\phi[y/0][x/1][z/0] = \top$$

Donc ϕ est satisfaisable avec l'interprétation $V(x) = 1$ et $V(y) = V(z) = 0$

2.3.2 Deuxième critère de choix

Le deuxième critère de choix se base sur les **propositions à polarité unique**.

Exemple Dans la formule

$$\phi = (x \vee \neg y \vee z) \wedge (x \vee \neg z) \wedge (y \vee z) \wedge (x \vee \neg y)$$

x apparaît toujours positivement, on peut donc directement lui assigner la valeur 1 sans être obligé de tester la valeur 0.

2.4 Algorithme DPLL

DPLL(ϕ) retourne `VRAI` si ϕ est satisfaisable.

if $\phi = \top$ **then**
| retourner `VRAI`
else if $\phi = \bot$ **then**
| retourner `FAUX`
else if ϕ *contient une clause unitaire* x **then**
| retourner DPLL($\phi[x/1]$)
else if ϕ *contient une clause unitaire* $\neg x$ **then**
| retourner DPLL($\phi[x/0]$)
else if ϕ *contient une proposition* x *de polarité toujours positive* **then**
| retourner DPLL($\phi[x/1]$)
else if ϕ *contient une proposition* x *de polarité toujours négative* **then**
| retourner DPLL($\phi[x/0]$)
else
| choisir une proposition x au hasard et retourner (DPLL($\phi[x/0]$) ou DPLL($\phi[x/1]$))
end

2.5 Transformation de Tseitin

Parfois, on cherche un résoudre un problème qui ne s'exprime pas facilement en FNC. La transformation de Tseitin va ajouter des nouvelles variables et des équivalences.

Exemple Prenons $\phi = (p \wedge q) \vee \neg(q \vee r)$. Dans la transformation de Tseitin, on remplace $p \wedge q$ par x_1 et $\neg(q \vee r)$ par x_2. Il faut donc réécrire la formule comme ceci :

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

Il reste encore à mettre les deux formules $(x_1 \leftrightarrow (p \wedge q))$ et $(x_2 \leftrightarrow \neg(q \vee r))$ sous FNC.

La technique de Tseitin sera particulièrement intéressante lorsqu'on devra mettre sous FNC des formules qui sont sous forme normale *disjonctive*.

2.6 Autre exemple de rajout de variable intéressant

Supposons qu'on ait n variables $x_0, x_1, ..., x_{n-1}$ et qu'on veuille exprimer qu'exactement une de ces variables doit être vraie.

2.6.1 Solution naïve

On considère les deux formules suivantes en conjonction :
- **Au moins une :** $\bigvee_{i=0}^{n-1} x_i$
- **Au plus une :** $\bigwedge_{0 \leq i < j < n} \neg x_i \vee \neg x_j$

Il y a donc $\frac{n(n-1)}{2} + 1$ clauses.

2.6.2 Avec un encodage binaire

On peut faire avec $nlog_2(n) + 1$ clauses (voir cours).

2.7 Problèmes de décision

On peut définir un problème de décision comme un langage de mots sur un alphabet fini Σ. On note Σ^* l'ensemble des mots sur l'alphabet Σ, et ϵ le mot vide. Un langage sur Σ est un sous-ensemble $L \subseteq \Sigma^*$.

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

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

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

2.8 Problème d'optimisation

Un *problème d'optimisation* est un problème où l'on veut maximiser ou minimiser une certaine quantité. On peut associer un problème de décision à un problème d'optimisation, en donnant une borne. Si on sait résoudre le problème de décision associé à un problème d'optimisation, on peut parfois résoudre le problème d'optimisation.

2.9 Algorithme de décision

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

2.10 La 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 $\mathcal{P}$ si 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 $u \notin P$

Exemples :
- décider si un tableau est trié
- décider si un entier codé en binaire est premier (difficile, problème resté ouvert pendant longtemps)

2.11 Algorithme de vérification

Informellement, un algorithme de vérification est un algorithme qui, étant donnée une solution candidate à un problème de décision, décide si oui ou non cette solution est valide. Formellement :

Un algorithme de vérification pour un problème $P \subseteq \Sigma^*$ est un algorithme A prenant deux mots en argument, 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.

2.12 La classe $\mathcal{NP}$

La classe $\mathcal{NP}$ est la classe des problèmes pouvant être vérifiés en temps polynomial. Note : $\mathcal{P} \subseteq \mathcal{NP} \subseteq$ ExpTime.

La **grande conjecture** de l'informatique fondamentale est :

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

Un problème de décision P est $\mathcal{NP}$-complet si il est dans $\mathcal{NP}$ et tout autre problème P' de $\mathcal{NP}$ se réduit à P en temps polynomial.

3 La logique des prédicats

En logique des prédicats,
- on ajoute les quantificateurs
- on généralise les valeurs que peuvent prendre les variables
- on ajoute des relations (appelées prédicats) pour décrire certaines relations entre ces valeurs
- on ajoute des symboles de fonction à la syntaxe

Exemple $\forall x \forall y. PremierEntreEux(x, y) \leftrightarrow \exists x' \exists y'. x.x' + y.y' = 1$

Les ingrédients pour construire les formules sont : connecteurs Booléens, quantificateurs, symboles de relations, de fonctions, constantes et termes. Pour satisfaire une formule, il faudra définir une interprétation des symboles dans un domaine.

3.1 Langages du premier ordre

Un *langage* $\mathcal{L}$ de la logique du premier ordre est caractérisé par
- des symboles de relations ou prédicats, notés $p, q, r, s, ...$
- des symboles de fonctions, notés $f, g, h, ...$
- des symboles de constantes, notés $c, d, e, ...$

Á chaque prédicat p (resp. fonction f), on associe un entier strictement positif appelé l'*arité* de p (resp f), c'est-à-dire le nombre d'arguments de p (resp f). On notera parfois $p|_n$ $f|_n$.

On utilise le prédicat "=" pour dénoter l'égalité. Si "=" fait partie du vocabulaire du langage $\mathcal{L}$, on dit que $\mathcal{L}$ est égalitaire.

Exemples

$$\mathcal{L}_1 = \{r|_1, c\}$$

$$\mathcal{L}_2 = \{r|_2, f|_1, g|_2, h|_2, c, d\}$$

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

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

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

L'ensemble des *formules atomiques* d'un langage $\mathcal{L}$ est l'ensemble des formules de la forme :
- $p(t_1, t_2, ..., t_n)$ où p est un prédicat d'arité n et $t_1, t_2, ..., t_n$ sont des termes du langage $\mathcal{L}$
- $t_1 = t_2$ si $\mathcal{L}$ est égalitaire et t_1, t_2 sont des termes du langage $\mathcal{L}$

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

$$\phi ::= p(t_1, t_2, ..., t_n)|\phi \wedge \phi|\phi \vee \phi|\neg\phi|\phi \rightarrow \phi|\phi \leftrightarrow \phi|\exists x.\phi|\forall x.\phi|(\phi)$$

Toute formule d'un langage du premier ordre se décompose de manière unique sous l'une, et une seule, des formes suivantes :
- une formule atomique
- $\neg\phi$, où ϕ est une formule,
- $\phi \wedge \psi$, $\phi \vee \psi$, $\phi \rightarrow \psi$, $\phi \leftrightarrow \psi$ où ϕ et ψ sont des formules
- $\forall x.\phi$ ou $\exists x.\phi$ où ϕ est une formule et x une variable.

Occurences Une *occurence* d'une variable dans une formule est un couple constitué de cette variable et d'une place effective, c'est-à-dire qui ne suit pas un quantificateur. **Exemple :** dans $r(x, z) \rightarrow \forall z.(r(y, z) \vee y = z)$, la variable x possède une occurence, la variable y deux et z trois.

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 close est une formule sans variable libre.

3.1.1 Structures

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 :
- d'un sous ensemble de M^n, noté $r^{\mathcal{M}}$, pour chaque symbole de prédicat r d'arité n dans $\mathcal{L}$
- d'une fonction 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 L'ensemble des réels $\mathbb{R}$ permet de construire une structure pour $\mathcal{L}_2 = \{r, f, g, h, c, d\}$ de la façon suivante :
- on interprète le prédicat r comme l'ordre $\leq$ sur les réels
- on interprète f comme la fonction $+1$, g comme $+$ et h comme $\times$
- on interprète les constantes c et d comme 0 et 1

Cette structure se note

$$\mathcal{M}_2 = (\mathbb{R}, \leq, +1, +, \times, 0, 1)$$

4 Preuves de programmes

4.1 Triplets de Hoare

Informellement, les triplets de Hoare sont de la forme (ϕ, P, ψ) où P est un programme, ϕ (resp ψ) est une pré-condition (resp. post-condition) décrite par une formule de la logique du premier ordre. On notera en général $(\!|\phi|\!)P(\!|\psi|\!)$. Nous allons définir un système de preuve pour démontrer que, supposant que ϕ est satisfaite, ψ est satisfaite aussi après exécution du programme P.

4.2 Correction

Soit ϕ une condition (pré ou post) et $I : x \rightarrow \mathbb{Z}$ un état. On dit que I *satisfait* ϕ, noté $I \vDash \phi$, si $\mathbb{Z}, I \vDash \phi$.

On dit qu'un triplet $(\!|\phi|\!)P(\!|\psi|\!)$ est *partiellement satisfait* si pour tout état qui satisfait ϕ, l'état résultant de l'exécution de P satisfait ψ, pourvu que P termine. On note cette relation $\vDash_{par} (\!|\phi|\!)P(\!|\psi|\!)$.

On dit qu'un triplet $(\!|\phi|\!)P(\!|\psi|\!)$ est *satisfait* si pour tout état qui satisfait ϕ, l'exécution de P à partir de cet état termine et l'état résultant de cette exécution satisfait ψ. On note cette relation $\vDash (\!|\phi|\!)P(\!|\psi|\!)$.

4.3 Système de preuve $\vdash_{par}$

Composition

$$\frac{(\!|\phi|\!)C_1(\!|\eta|\!) \quad (\!|\eta|\!)C_2(\!|\psi|\!)}{(\!|\phi|\!)C_1; C_2(\!|\psi|\!)} Composition$$

Assignation

$$\frac{}{(\!|\psi[E/x]|\!)x := E(\!|\psi|\!)} Assignation$$

où $\psi[E/x]$ est la formule ψ où on a remplacé toutes les occurences de x par E. Cette règle est un axiome.
Exemple : $(\!|y + z \geq 0|\!)x := y + z(\!|x \geq 0|\!)$

Si-Alors

$$\frac{(\!|\phi \wedge B|\!)C_1(\!|\psi|\!) \quad (\!|\phi \wedge \neg B|\!)C_2(\!|\psi|\!)}{(\!|\phi|\!)\ \texttt{if}\ B\ \{C_1\}\ \texttt{else}\ \{C_2\}\ (\!|\psi|\!)} Si - Alors$$

Implication

$$\frac{\phi' \vDash \phi \quad (\!|\phi|\!)C(\!|\psi|\!) \quad \psi \vDash \psi'}{(\!|\phi'|\!)C(\!|\psi'|\!)} Implication$$

On pourra omettre une des implications si on n'a besoin que de l'une ou des deux.

While Partiel

$$\frac{(\!|\psi \wedge B|\!)C(\!|\psi|\!)}{(\!|\psi|\!)\ \texttt{while}\ B\ \{C\}\ (\!|\psi \wedge \neg B|\!)} WhilePar$$

La propriété ψ est appelée *invariant*. Elle est toujours vraie à chaque passage dans la boucle, ainsi qu'après le dernier passage.

4.4 Correction et complétude

Soit $(\!|\phi|\!)P(\!|\psi|\!)$ un triplet de Hoare. Alors $\vDash_{par} (\!|\phi|\!)P(\!|\psi|\!)$ si et seulement si $\vdash_{par} (\!|\phi|\!)P(\!|\psi|\!)$

4.5 Terminaison d'un programme

Comme l'instruction `while` est la seule source de non-terminaison, on va identifier une expression arithmétique E et démontrer qu'elle décroît strictemenet à chaque passage dans la boucle. Comme la quantité ne peut être négative, on aura bien une preuve de terminaison.

Une telle expression est appelée *variant*.

While Total

$$\frac{(\!|\eta \wedge B \wedge 0 \leq E = e_0|\!)C(\!|\eta \wedge 0 \leq E < e_0|\!)}{(\!|\eta \wedge 0 \leq E = e_0|\!)\ \texttt{while}\ B\ \{C\}\ (\!|\eta \wedge \neg B|\!)} WhileTot$$

- E est l'expression arithmétique qui décroît à chaque exécution de C
- e_0 est une variable logique
- η est l'invariant

5 Automates finis

Un automate fini lit une séquence de lettres de gauche à droite, possède un nombre fini d'états. En fonction de l'état courant et de la lettre lue, il se déplace vers un autre état.

L'état initial est représenté par une flèche sans source et les états finaux sont représentés par des doubles cercles. Le mot est accepté si et seulement si l'automate se trouve dans un état final à la fin du mot.

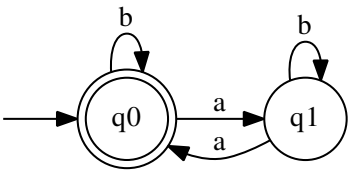


FIGURE 1 – Automate fini A

Langage accepté Le *langage accepté (ou reconnu)* par l'automate A (fig. 1), noté $L(A)$, est l'ensemble des mots qui contiennent un nombre pair de lettres a :

$$L(A) = \{w \in \{a,b\}^* | w \text{ contient un nombre pair de} a\}$$

Définition formelle Un automate fini A sur un alphabet Σ est un 4-uplet (Q, q_0, F, δ) où
- Q est un ensemble fini d'éléments appelés *états*
- q_0 est appelé *état initial*
- $F \subseteq Q$ est un ensemble d'états dits *finaux ou acceptants*
- $\delta : Q \times \Sigma \to Q$ est une fonction (pas nécessairement totale) appelée *fonction de transition*

Exécution Une *exécution* de A est une suite finie $e = p_0\sigma_1 p_1 \sigma_2 ... p_{n-1}\sigma_n p_n$ $(n \geq 0)$ telle que
- $p_0 = q_0$
- $\forall i \in \{0, ..., n\} : p_i \in Q$
- $\forall i \in \{1, ..., n\} : \sigma_i \in \Sigma$
- $\forall i \in \{0, ..., n-1\} : \delta(p_i, \sigma_{i+1})$ est définie et vaut $p_i + 1$

On dit que l'exécution e est *acceptante* si l'état atteint est final $(p_n \in F)$

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

Lemme On peut toujours transformer un automate A en un automate B complet qui accepte le même langage. (Idée : ajouter un état supplémentaire appelé état puits non final et ajouter les transition manquantes vers cet état)

5.1 Test du vide

Le problème VIDE est le suivant :
- **Entrée :** un automate A sur un alphabet Σ
- **Sortie :** est-ce que $L(A) = \varnothing$?

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

Etant donné un automate A avec n états et m transitions, on peut tester en $O(n+m)$ si $L(A) \neq \varnothing$ (en utilisant des algorithmes de graphe classiques comme le parcours en largeur par exemple).

5.2 Opérations Booléennes sur les langages

Complément de $L \subseteq \Sigma^*$: $\overline{L} = \{w \in \Sigma^* | w \notin L\} = \Sigma^* \setminus L$.

Union, Intersection : Trivial.

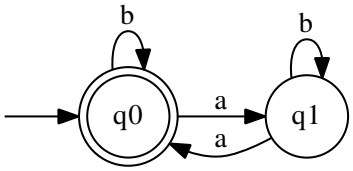
5.3 Clôture des automates par opérations Booléennes

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

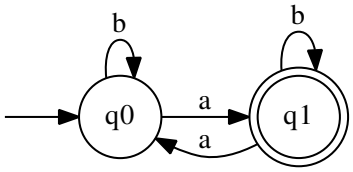
$$\begin{aligned} L(A_c) &= \overline{L(A_1)} \\ L(U) &= L(A_1) \cup L(A_2) \\ L(I) &= L(A_1) \cap L(A_2) \end{aligned}$$

5.3.1 Clôture par complément

Si $A = (Q, Q_0, F, \delta)$ n'est pas complet, il faut le compléter, puis $A_c = (Q, Q_0, Q \setminus F, \delta)$. Par exemple, si $A =$



alors $A_c =$



5.3.2 Produit d'automates

On appellera *pré-automate* sur Σ un triplet (Q, q_0, δ) où Q est un ensemble fini, $q_0 \in Q$ et $\delta : Q \times \Sigma \to Q$ est une fonction.

Soient $A_1 = (Q_1, q_0^1, F_1, \delta_1$ et $A_2 = (Q_2, q_0^2, F_2, \delta_2$ deux automates sur 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, pour tout $(q_1, q_2) \in Q_1 \times Q_2$, pour tout $\sigma \in \Sigma$,

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

Exemple Voici un exemple graphique

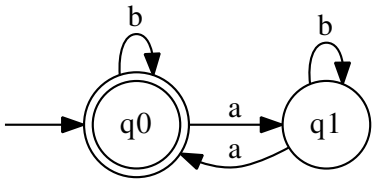


FIGURE 2 – A_1

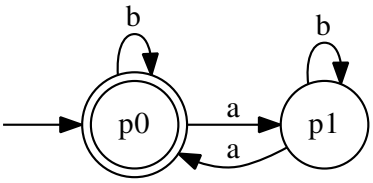


FIGURE 3 – A_2

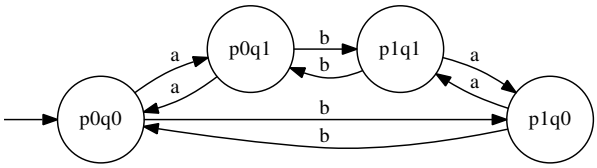


FIGURE 4 – $A_1 \otimes A_2$

Exemple intuitif Si $L(A_1)$ est l'ensemble des mots qui contiennent un nombre pair de b et $L(A_2)$ l'ensemble des mots qui contiennent un nombre pair de a, toute exécution de $A_1 \otimes A_2$ sur un mot w simule en parallèle l'exécution de A_1 sur w, ainsi que celle de A_2.

5.3.3 Intersection et Union

- Pour avoir $L(A_1) \cap L(A_2)$, il suffit de prendre $F_\cap = F_1 \times F_2$ pour 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 états finaux de $A_1 \otimes A_2$

Pour la clôture (construire un automate à partir du pré-automate produit et des états finaux ci-dessus), A_1 et A_2 doivent être complets.

5.3.4 Inclusion et Equivalence

Soient A_1 et A_2 deux automates sur un alphabet Σ. On dit que A_1 et A_2 sont équivalents si $L(A_1) = L(A_2)$.
On peut décider en temps polynomial si $L(A_1) \subseteq L(A_2)$, et si $L(A_1) = L(A_2)$

Algorithme $L(A_1) = L(A_2)$ ssi $L(A_1) \subseteq L(A_2)$ et $L(A_2) \subseteq L(A_1)$. On peut donc se concentrer sur l'inclusion. Il faut compléter A_1 et A_2. Ensuite, on a $L(A_1) \subseteq L(A_2)$ ssi $L(A_1) \cap \overline{L(A_2)} = \varnothing$

1. construire A_c tel que $L(A_c) = \overline{L(A_2)}$
2. construire I tel que $L(I) = L(A_1) \cap L(A_c)$ avec le produit $A_1 \otimes A_c$
3. tester le vide de I

Exemple Il y a un excellent exemple dans le cours.

5.4 Minimisation

Etant 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.

5.4.1 Relation de Myhill-Nerode

Soit $L \subseteq \Sigma^*$ un langage, et $u, v \in L$ deux mots de L. Les mots u et v sont dits équivalents pour L, noté $u \equiv_L v$ si $\forall w \in \Sigma^*, uw \in L \text{ssi} vw \in L$.

Exemple les mots $u = ababb$ et $v = baa$ sont équivalents pour le langage PAIR.

Proposition La relation $\equiv_L$ est une relation d'équivalence. (symétrique, réflexive, transitive)

Classes d'équivalence Etant donné un mot $u \in \Sigma^*$, la *classe d'équivalence* de u pour $\equiv_L$, notée $[u]_L$, est l'ensemble des mots équivalents à u :

$$\forall u \in \Sigma^*, [u]_L = \{v \in \Sigma^* | u \equiv_L v\}$$

On note $\Sigma^*/_{\equiv_L}$ l'ensemble quotient de Σ^* par $\equiv_L$, i.e. l'ensemble des classes d'équivalences.

5.4.2 Théorème de Myhill-Nerode

Soit $L \subseteq \Sigma^*$. Alors L est reconnaissable par un automate ssi $\Sigma^*/_{\equiv_L}$ est un ensemble fini.

Exemple Avec le langage PAIR, on a exactement deux classes d'équivalences :

$$\begin{aligned}\{a,b\}^*/_{\equiv_{PAIR}} &= \{\{\epsilon, aa, aba, baa, ...\}, \{a, ab, ba, aaa, ...\}\} \\ &= \{[\epsilon]_{\equiv_{PAIR}}, [a]_{\equiv_{PAIR}}\}\end{aligned}$$

5.4.3 Automate Quotient

Soit $L \subseteq \Sigma^*$ un langage reconnaissable par un automate. Donc $\Sigma^*/_{\equiv_L}$ est un ensemble fini.

L'*automate quotient* de L est l'automate complet $A_L = (Q, q_0, F, \delta)$ où
- $Q = \Sigma^*/_{\equiv_L}$
- $q_0 = [\epsilon]_L$
- $F = \{[u]_L \,|\, u \in L\}$
- δ est la fonction de transition définie par $\delta([u]_L, a) = [ua]_L$

Théorème Soit $L \subseteq \Sigma^*$. L'automate quotient A_L est le plus petit automate complet (en nombre d'états) qui reconnaît L, et il est unique (modulo renommage des états).

5.4.4 Calcul de l'automate minimal

Définitions Soit $A = (Q, q_0, F, \delta)$ un automate complet sur un alphabet Σ. Pour tout mot $u \in \Sigma^*$, pour tout état $q \in Q$, n note :
- $q \cdot u$ l'état atteint par A après lecture de u à partir de q (il existe car A est complet)
- L_q le langage formé des mots u tels que $q \cdot u \in F$ (les mots acceptés à partir de q).
- $\forall p, q \in Q$, $p \equiv_A q$ si $L_p = L_q$

Proposition Soit A un automate complet. Alors pour tout $u, v \in \Sigma^*$, $u \equiv_{L(A)} v$ ssi $q_0 \cdot u \equiv_A q_O \cdot v$.

Calcul
- Il suffit de calculer les classes d'équivalences de Q pour $\equiv_A$, ce qui donnera les états
- la classe de q_0 est l'état initial
- toute classe qui contient un état final est finale (en fait, si une classe contient un état final, alors tous ces états sont finaux)
- 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 $q \in Q$.

Proposition Si $p \equiv_A q$, alors $p \cdot q \equiv_A q \cdot a$ pour tout $a \in \Sigma$.

5.5 Expressions Rationnelles

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

$$E \;::=\; \epsilon \,|\, a \,|\, \varnothing \,|\, (E + E) \,|\, (E.E) \,|\, E^*$$

5.5.1 Autres opérations sur les langages

Soient $L, L_1, L_2, \subseteq \Sigma^*$ trois langages. Alors
- $L_1.L_2 = \{u_1u_2 \mid u_1 \in L_1 \wedge u_2 \in L_2\}$ (on écrira aussi L_1L_2)
- $L^* = \{u^n \mid u \in L \wedge n \geq 0\}$ (en particulier $\epsilon \in L^*$ si $L \neq \varnothing$)

5.5.2 Sémantique

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 $a \in \Sigma$
- $L(\varnothing) = \varnothing$
- $L((E_1 + E_2)) = L(E_1) \cup L(E_2)$
- $L((E_1.E_2)) = L(E_1).L(E_2)$
- $L(E^*) = L(E)^*$

Exemples Sur $\Sigma = \{a, b\}$.
- $L((a + b)^*a(a + b)^*)$ est l'ensemble des mots qui contiennent au moins un a
- $L(a^*b^*)$ est l'ensemble de mots qui sont des séquences de a suivies de séquences de b.

5.5.3 Théorème dit de Kleene

La classe des langages définissables par automates, et la classe des langages définissables par expressions rationnelles, sont les mêmes. Autrement dit, si $L \subseteq \Sigma^*$ est un langage, alors il existe un automate A tel que $L = L(A)$ ssi il existe une expression rationnelle E telle que $L = L(E)$.

Remarque Il existe un algorithme qui transforme tout automate en expression rationnelle et inversement.