Chapitre 8 : Logique

Table des matières

1 Introduction

La logique est un domaine mathématique dont l'objet d'étude est l'ensemble des propriétés que l'on peut exprimer, leur valeur de vérité et la notion de démonstration. La logique propose un cadre formel pour manipuler ces notions, où l'on distingue la *syntaxe*, *i.e* la manière d'exprimer les propriétés, de la *sémantique*, *i.e* des interprétations que l'on peut donner à la syntaxe.

Les démonstrations, vues en tant qu'objets mathématiques, ne seront étudiées qu'en MPI.

2 Syntaxe des formules logiques

2.1 Logique propositionnelle

2.1.1 Introduction

On décompose les propriétés que l'on cherche à exprimer pour faire ressortir leur structure logique. Les énoncés que l'on ne peut pas décomposer sont appelés *atomes*. En logique propositionnelle, on abstrait les atomes en des variables dites propositionnelles.

2.1.2 Définition (formules de la logique propositionnelle)

Soit $\mathcal{V}$ un ensemble de variables propositionnelles.

On définit inductivement l'ensemble $\mathcal{F}$ des formules de la logique propositionnelle par

$$\frac{x \in \mathcal{V}}{x \in \mathcal{F}} \qquad \frac{\varphi \in \mathcal{F}}{\neg\varphi \in \mathcal{F}}$$

$$\frac{\varphi_1 \in \mathcal{F}\ \varphi_2 \in \mathcal{F}}{\varphi_1 \vee \varphi_2 \in \mathcal{F}} \qquad \frac{\varphi_1 \in \mathcal{F}\ \varphi_2 \in \mathcal{F}}{\varphi_1 \wedge \varphi_2 \in \mathcal{F}} \qquad \frac{\varphi_1 \in \mathcal{F}\ \varphi_2 \in \mathcal{F}}{\varphi_1 \to \varphi_2 \in \mathcal{F}}$$

On appelle $\neg$ la *négation*, $\vee$ la *disjonction*, $\wedge$ la *conjonction*, et $\to$ l'*implication*.

On appelle aussi *équivalence* le symbole $\leftrightarrow$ défini par

$$\varphi_1 \leftrightarrow \varphi_2 = (\varphi_1 \to \varphi_2) \wedge (\varphi_2 \to \varphi_1)$$

2.1.3 Remarque

Malgré leur nom, les symboles de la définition précédente n'ont aucune signification.

Le sens des symboles tient au domaine de la sémantique. On parle ici de syntaxe abstraite.

On peut utiliser une autre représentation pour les syntaxes abstraites, appelée *grammaire*, qui s'écrit comme suit :

$$\varphi ::= x \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \to \varphi_2$$

où x parcourt $\mathcal{V}$.

L'étude précise des grammaires relève du programme de MPI.

2.1.4 Vocabulaire

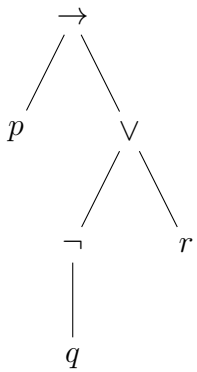
Les symboles $\neg, \vee, \wedge, \rightarrow, \leftrightarrow$ sont appelés connecteurs logiques. Ce sont eux qui définissent la structure logique des énoncés. Le nombre d'arguments d'un connecteur logique est appelé son *arité*.

2.1.5 Représentation des formules

On représente généralement les formules de manière linéaire en ajoutant des parenthèses pour désambigüiser la lecture. Par exemple, $p \vee q \wedge r$ peut se lire $p \vee (q \wedge r)$ ou $(p \vee q) \wedge r$. Par convention, la négation est prioritaire sur les autres connecteurs, donc $\neg p \vee q$ se lit $(\neg p) \vee q$ et pas $\neg(p \vee q)$.

En raison de leur définition inductive, les formules ont une représentation arborescente naturelle : chaque règle d'inférence utilisée définit un symbole de tête / connecteur principal qui est la racine du sous-arbre correspondant.

Exemple : $p \rightarrow (\neg q \vee r)$ est représenté par :



Remarque : on obtient un arbre (binaire) dont les nœuds internes sont les connecteurs logiques et les nœuds externes sont les atomes. L'arité d'un connecteur est l'arité du nœud correspondant.

L'écriture linéaire de la formule correspond au parcours en profondeur infixe, en adoptant une représentation préfixe pour les nœuds d'arité 1.

On pourrait envisager d'utiliser les parcours préfixes et postfixe pour représenter les formules.

2.1.6 Vocabulaire

– Une *sous-formule* d'une formule φ donnée est la formule associée à un sous-arbre de l'arbre représentant φ.

On définit inductivement l'ensemble $SF(\varphi)$ des sous-formules de φ par :

$$\forall x \in \mathcal{V},\ SF(x) = \{x\} \qquad SF(\neg\varphi) = \{\neg\varphi\} \cup SF(\varphi)$$

$$\forall \circ \in \{\vee, \wedge, \rightarrow\},\ SF(\varphi_1 \circ \varphi_2) = \{\varphi_1 \circ \varphi_2\} \cup SF(\varphi_1) \cup SF(\varphi_2)$$

Exemple :

$$SF(p \to (\neg q \vee r)) = \{p \to (\neg q \vee r), \neg q \vee r, p, \neg q, r, q\}$$

– La *taille* $|\varphi|$ d'une formule φ est le nombre de connecteurs logiques de la formule, *i.e* le nombre de nœuds internes de l'arbre A_φ associé à φ.

Proposition :

$$\frac{|A_\varphi| - 1}{2} \leqslant |\varphi| \leqslant |A_\varphi| - 1$$

□ Démonstration :

L'arbre A_φ contient au moins une feuille donc $|\varphi| \leqslant |A_\varphi| - 1$
(cas d'égalité pour φ de taille n : $\varphi = \underbrace{\neg\neg\cdots\neg}_{n \text{ fois}} p$)

A_φ est un arbre binaire ayant $|\varphi|$ nœuds internes, donc il a au plus $|\varphi| + 1$ feuilles (Chap 6, 1.1.10), donc $|A_\varphi| \leqslant |\varphi| + |\varphi| + 1 = 2\,|\varphi| + 1$ ■

– La *hauteur* d'une formule est la hauteur de l'arbre associé, *i.e* le nombre maximal de connecteurs à traverser pour atteindre une variable propositionnelle.
Exemple : la hauteur de $p \to (\neg q \vee r)$ est 3.

2.2 Logique du premier ordre

2.2.1 Définition (*langage du premier ordre*)

Un *langage du premier ordre* est défini par une signature Σ, composée de :

– symboles de fonction, chacun muni d'une arité $k \in \mathbb{N}$. Les symboles de fonction d'arité 0 sont appelés symboles de constante ;

– symboles de prédicat ou de relation, chacun muni d'une arité $k \in \mathbb{N}$. Les symboles de prédicat d'arité 0 sont appelés symboles de constante propositionnelle.

2.2.2 Exemple : le langage de la théorie des ensembles

Le langage de la théorie des ensembles est défini par la signature suivante :

– symboles de fonction : $\varnothing$ (arité 0), $\{\cdot\}$ (arité 1), $\cdot \cup \cdot$ et $\cdot \cap \cdot$ (arité 2), $\cdot^c$ (arité 1) ;

– symboles de prédicat : $\cdot = \cdot$, $\cdot \subseteq \cdot$, $\cdot \in \cdot$ (arité 2).

2.2.3 Définition (*termes* et formules de la logique du premier ordre)

Soit Σ une signature et $\mathcal{V}$ un ensemble de variables.

Σ et $\mathcal{V}$ définissent des ensembles de termes et de formules, *via* les grammaires suivantes :

– Termes :

$$t ::= x \mid f(t_1, \ldots, t_k)$$

où

$$\begin{cases} x \text{ parcourt } \mathcal{V} \\ (f, k) \text{ parcourt les symboles de la fonction et leur arité} \end{cases}$$

– Formules :

$$\varphi ::= p(t_1, \ldots, t_k) \mid \neg\varphi \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \to \varphi_2 \mid \forall x\ \varphi \mid \exists x\ \varphi$$

où :

$$\begin{cases} x \text{ parcourt } \mathcal{V} \\ (p, k) \text{ parcourt les symboles de prédicat et leur arité} \end{cases}$$

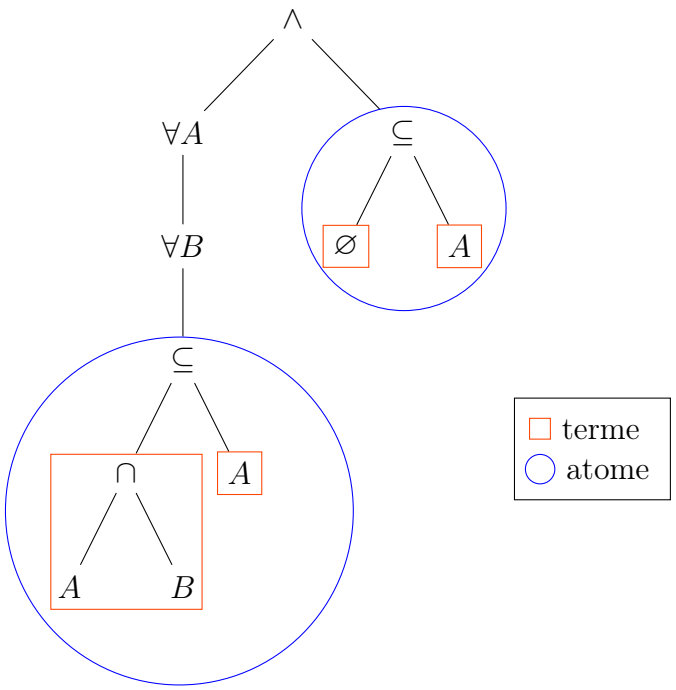
On appelle $\forall$ le *quantificateur universel*, et $\exists$ le *quantificateur existentiel*.
Les atomes de cette logique sont les formules de la forme $p(t_1, \ldots, t_k)$.

$$\exists! x,\ P(x) \ \equiv\ \exists x\ (P(x) \wedge \forall y (P(y) \to y = x))$$

On parle de logique du premier ordre car on ne peut quantifier que sur des variables représentant des termes. Si l'on peut quantifier sur des variables représentant des formules, on parle de logique du second ordre.

2.2.4 Exemple

En théorie des ensembles, la formule $(\forall A, \forall B, A \cap B \subseteq A) \wedge \varnothing \subseteq A$ est représentée de manière arborescente par :



2.2.5 Remarque

• La logique du premier ordre est aussi appelée calcul des prédicats.

• Dans une formule du premier ordre, les variables peuvent être "capturées" par un quantificateur ou indépendantes de toute quantification.
On peut voir une formule comme une propriété des variables indépendantes de toute quantification et donc remplacer ces variables par des termes concrets.

• Le calcul propositionnel est un cas particulier du calcul des prédicats où il n'y a que des constantes propositionnelles (les quantificateurs et les termes deviennent inutiles).

2.2.6 Définition (variables libres / liées)

Les *variables libres* d'une formule φ sont les variables qui ne sont pas "capturées" par un quantificateur. On les définit inductivement par

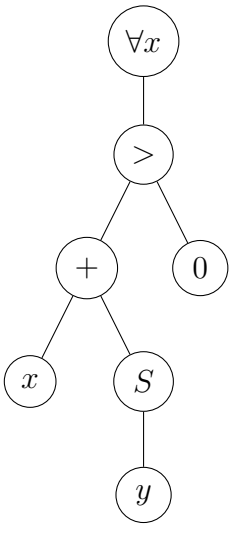
$$FV(p(t_1,\dots,t_n)) = \bigcup_{i=1}^{n} \text{Vars}(t_i) \quad \text{où} \quad \begin{cases} \text{Vars}(x) = \{x\} \\ \text{Vars}(f(t_1,\dots,t_n)) = \bigcup_{i=1}^{n} \text{Vars}(t_i) \end{cases}$$

$$FV(\neg\varphi) = FV(\varphi)$$

$$\forall \circ \in \{\vee, \wedge, \to\},\ FV(\varphi_1 \circ \varphi_2) = FV(\varphi_1) \cup FV(\varphi_2)$$

$$\forall Q \in \{\exists, \forall\},\ FV(Qx\ \varphi) = FV(\varphi) \setminus \{x\}$$

Exemple :
$FV(\forall x,\ x + Sy > 0) = \{y\}$



$FV((\forall x\ x + y = 1) \wedge (\forall y\ x + y = 1)) = \{x, y\}$
Une variable est dite *liée* si elle n'est pas libre. Un même nom de variable peut avoir des occurrences libres et des occurrences liées. Dans une formule de la forme $Qx\ \varphi$ où $Q \in \{\forall, \exists\}$, on dit que φ est la portée de la liaison pour x. Une variable est donc libre si elle admet une occurrence hors de la portée de toutes les liaisons pour cette variable

et une variable est liée si toutes ses occurrences sont dans la portée d'une liaison pour cette variable.

Une formule dont toutes les variables sont liées est dite *close*.

2.2.7 Remarque

Le nom des variables liées n'est pas important.

Exemple : $\forall x,\ x = x$ et $\forall y,\ y = y$ expriment la même propriété.

On identifiera donc les formules au renommage près de leurs variables liées.

On appelle cela *l'α-équivalence*.

Lors du renommage de variables liées, il faut faire attention au phénomène de *capture* de variables, par exemple $\forall y,\ x + y = 1$ n'est pas la même formule que $\forall x,\ x + x = 1$.

2.2.8 Définition (*substitution*)

Soit φ une formule, x une variable, et t un terme.

La substitution de t à x dans φ, notée $\varphi[x := t]$ est définie inductivement par :

$$p(t_1, \ldots, t_n)[x := t] = p(t_1[x := t], \ldots, t_n[x := t])$$

où

$$\begin{cases} x[x := t] = t \\ y[x := t] = y \ \forall y \in \mathcal{V} \setminus \{x\} \\ f(t_1, \ldots, t_n)[x := t] = f(t_1[x := t], \ldots, t_n[x := t]) \end{cases}$$

$(\neg\varphi)[x := t] = \neg(\varphi[x := t])$

$\forall \circ \in \{\vee, \wedge, \to\},\ (\varphi_1 \circ \varphi_2)[x := t] = \varphi_1[x := t] \circ \varphi_2[x := t]$

$\forall Q \in \{\forall, \exists\},\ (Qx\ \varphi)[x := t] = Qx\ \varphi$

$\forall Q \in \{\exists, \forall\},\ (Qy\ \varphi)[x := t] = Qy\ (\varphi[x := t])$ si $y \neq x$ et $y \notin \mathrm{Vars}(t)$

Exemple :

$(\forall x,\ x = x)[x := 2 + 2] = \forall x,\ x = x$ car $(\forall x,\ x = x) \underset{\alpha}{\equiv} (\forall y,\ y = y)$

$((\forall x,\ x + y = 1) \wedge (\forall y,\ x + y = 1))[x := 2] = (\forall x,\ x + y = 1) \wedge (\forall y,\ 2 + y = 1)$

$(\forall y,\ y = y + x)[x := 1 + y] \neq \forall y(y = y + 1 + y)$

Mais plutôt $\forall z,\ z = z + 1 + y$: on renomme les occurrences liées dans φ des variables de t avant de les substituer.

2.2.9 Remarque

Le principe de l'α-équivalence et les restrictions de la substitution sont liées aux questions de sémantique : l'α-équivalence et la substitution doivent en quelque sorte conserver la signification logique des formules.

3 Sémantique de la logique propositionnelle

3.1 Vocabulaire

3.1.1 Introduction

Définir une sémantique revient à donner du sens aux symboles utilisés dans la syntaxe abstraite. On doit donc choisir un ensemble de valeurs qui servent d'interprétations aux termes construits à l'aide de la syntaxe et on doit décrire l'effet des symboles sur cet ensemble de valeurs.

Exemple : on considère des termes arithmétiques définis par :

$$t ::= x \mid c \mid t_1 + t_2 \mid t_1 - t_2 \mid t_1 \times t_2$$

où x parcourt un ensemble $\mathcal{V}$ de variables et c parcourt $\mathbb{N}$, l'ensemble des constantes.

On peut définir une sémantique en choisissant $\mathbb{N}$ pour l'ensemble des valeurs, $c + 1$ comme interprétation de c, la fonction min comme interprétation de $+$, la fonction max pour $-$, et l'addition pour $\times$.

On peut bien-sûr donner une autre sémantique à ces termes, plus en cohérence avec les règles de l'arithmétique.

Problème : l'interprétation des variables : elle dépend d'un contexte qui donne une valeur à chaque variable.

Les sémantiques sont donc paramétrées par un environnement.

3.1.2 Définition (*valuation*)

Une *valuation* est une fonction de l'ensemble $\mathcal{V}$ des variables dans l'ensemble des valeurs choisies pour définir la sémantique. On parle aussi d'environnement, ou, dans le cas de la logique propositionnelle, de *distribution* de vérité. L'ensemble des valeurs de vérité est noté $\{V, F\}$ où V est la valeur vraie et F la valeur fausse.

3.1.3 Définition (valeur de vérité d'une formule)

Soit φ une formule et v une valuation.

On définit inductivement l'interprétation de φ pour v, notée $[\![\varphi]\!]_v$, par :

$$\forall x \text{ variable propositionnelle, } [\![x]\!]_v = v(x)$$

$$[\![\neg\varphi]\!]_v = \begin{cases} V \text{ si } [\![\varphi]\!]_v = F \\ F \text{ sinon} \end{cases}$$

$$[\![\varphi_1 \vee \varphi_2]\!]_v = \begin{cases} F \text{ si } [\![\varphi_1]\!]_v = [\![\varphi_2]\!]_v = F \\ V \text{ sinon} \end{cases}$$

$$[\![\varphi_1 \wedge \varphi_2]\!]_v = \begin{cases} V \text{ si } [\![\varphi_1]\!]_v = [\![\varphi_2]\!]_v = V \\ F \text{ sinon} \end{cases}$$

$$[\![\varphi_1 \to \varphi_2]\!]_v = \begin{cases} F \text{ si } [\![\varphi_1]\!]_v = V \text{ et } [\![\varphi_2]\!]_v = F \\ V \text{ sinon} \end{cases}$$

On dit v est un modèle de $\varphi \Leftrightarrow [\![\varphi]\!]_v = V$

3.1.4 Définition (*tautologie* / *antilogie* / *satisfiabilité*)

Soit φ une formule.
On dit que φ est
- une *tautologie* $\Leftrightarrow$ toute valuation est un modèle de φ. On note alors $\vDash \varphi$;
- une *antilogie* $\Leftrightarrow$ elle n'admet aucun modèle ;
- *satisfiable* $\Leftrightarrow$ elle admet au moins un modèle.

3.1.5 Remarque

On ajoute parfois à la syntaxe une tautologie notée $\top$ et une antilogie notée $\bot$.
On peut toutefois les encoder :

$$\top = x \vee \neg x \qquad \bot = x \wedge \neg x$$

La tautologie $\varphi \vee \neg\varphi$ est appelée loi du tiers exclu.
Exo Montrer que les formules suivantes sont des tautologies :
$p \to (q \to p), (p \to q) \vee (q \to r), (p \to (q \to r)) \to ((p \to q) \to (p \to r))$

Ex : Falso Quadlilet : $\bot \to \varphi$
$\neg\neg\varphi \to \varphi$ (non admis par tout le monde)
principe de démonstration par l'absurde : $\neg\neg\varphi = \neg\varphi \to \bot$

3.1.6 Définition (*table de vérité*)

La *table de vérité* de φ est la table indexée par les valuations des variables de φ et qui contient comme entrée correspondant à une valuation v la valeur $[\![\varphi]\!]_v$.
On représente la table de φ, de variables $x_1, \ldots, x_n$, en plaçant une colonne pour chaque x_i et une colonne pour φ.
Pour chaque valuation v, l'entrée correspondant à x_i est $v(x_i)$ et l'entrée correspondant à φ est $[\![\varphi]\!]_v$.

3.1.7 Exemple : table de vérité de $p \leftrightarrow q$

$p \leftrightarrow q = (p \to q) \wedge (q \to p)$

p	q	$p \to q$	$q \to p$	$p \leftrightarrow q$
F	F	V	V	V
F	V	V	F	F
V	F	F	V	F
V	V	V	V	V

Remarque : Ici on a considéré des colonnes supplémentaires pour des sous-formules pour simplifier le calcul.

3.1.8 Remarque

Construire une table de vérité est un algorithme simple pour déterminer si une formule est satisfiable / une tautologie / une antilogie.

Cependant, si φ a n variables distinctes, alors il y a 2^n lignes dans sa table de vérité.

De plus, étant donné v, déterminer $[\![\varphi]\!]_v$ se fait en temps $\mathcal{O}(|\varphi|)$ et φ peut avoir au plus $|\varphi| + 1$ variables distinctes.

Cela donne donc un algorithme de complexité $\mathcal{O}(|\varphi|\, 2^{|\varphi|})$

3.1.9 Proposition

Il y a $2^{(2^n)}$ tables de vérités distinctes pour des formules à n variables distinctes.

☐ Démonstration :

Il y a 2^n lignes dans une table et pour chaque ligne on a le choix entre 2 valeurs de vérité ■

3.2 Conséquence et équivalence

3.2.1 Définition (*conséquence*)

Soient φ et ψ deux formules.

On dit que ψ est une conséquence logique de φ, noté $\varphi \vDash \psi$, si tout modèle de φ est un modèle de ψ.

3.2.2 Proposition

Soient φ et ψ deux formules.

$$\varphi \vDash \psi \iff \vDash \varphi \to \psi$$

☐ Démonstration :

$$\begin{aligned}
\varphi \vDash \psi &\iff \text{tout modèle de } \varphi \text{ est un modèle de } \psi \\
&\iff \forall \text{ valutation } v, \text{ soit } [\![\varphi]\!]_v = V = [\![\psi]\!]_v, \text{ soit } [\![\varphi]\!]_v = F \\
&\iff \forall \text{ valuation } v, [\![\varphi \to \psi]\!]_v = V \\
&\iff \vDash \varphi \to \psi \quad \blacksquare
\end{aligned}$$

3.2.3 Généralisation aux ensembles de formules

– Modèle : un modèle d'un ensemble Γ de formules est une valuation v telle que

$$\forall \varphi \in \Gamma,\ [\![\varphi]\!]_v = V$$

– Satisfiabilité : un ensemble Γ de formules est dit :
- satisfiable si Γ admet un modèle ;
- contradictoire si Γ n'admet pas de modèle.

– Conséquence logique : une formule φ est conséquence logique d'un ensemble Γ de formules si tout modèle de Γ est modèle de φ. On note alors $\Gamma \vDash \varphi$.

Remarques :
– $\varphi \vDash \psi \ \Leftrightarrow\ \{\varphi\} \vDash \psi$
– L'intuition est la suivante : un ensemble de formules Γ définit une théorie (un ensemble d'axiomes) et on s'intéresse aux formules vraies dans cette théorie (les conséquences logiques de Γ).

3.2.4 Proposition

> Soit Γ un ensemble de formules, et φ une formule.
>
> $$\Gamma \vDash \varphi \ \Leftrightarrow\ \Gamma \cup \{\neg\varphi\} \text{ est contradictoire}$$

☐ Démonstration :

$\Gamma \vDash \varphi \quad \Leftrightarrow$ tout modèle de Γ est un modèle de φ

$$\Leftrightarrow \quad \forall \text{ valuation } v, \ \left| \begin{array}{l} \text{soit } v \text{ n'est pas un modèle de } \Gamma \\ \text{soit } \begin{cases} v \text{ est un modèle de } \Gamma \\ [\![\varphi]\!]_v = V \end{cases} \end{array} \right.$$

$$\Leftrightarrow \quad \forall \text{ valuation } v, \ \left| \begin{array}{l} \text{soit } v \text{ n'est pas un modèle de } \Gamma \\ \text{soit } \begin{cases} v \text{ est un modèle de } \Gamma \\ [\![\neg\varphi]\!]_v = F \end{cases} \end{array} \right.$$

$\Leftrightarrow\ \Gamma \cup \{\neg\varphi\}$ n'admet aucun modèle
$\Leftrightarrow\ \Gamma \cup \{\neg\varphi\}$ est contradictoire. ■

3.2.5 Proposition

> Soient $(\varphi_k)_{k\in[\![1\ ;\ n]\!]}$ et ψ des formules.
>
> $$\{\varphi_k \mid k \in [\![1\ ;\ n]\!]\} \vDash \psi \ \Leftrightarrow\ \bigwedge_{k=1}^{n} \varphi_k \to \psi$$

☐ Démonstration :

Lemme : Soit Γ un ensemble de formules, et φ, ψ deux formules.

$$\Gamma \cup \{\varphi\} \vDash \psi \;\Leftrightarrow\; \Gamma \vDash \varphi \to \psi$$

□ Démonstration du lemme :

$$\begin{aligned}
\Gamma \cup \{\varphi\} \vDash \psi \quad &\Leftrightarrow \quad \forall \text{ valuation } v, \;\Big|\; \begin{array}{l}\text{soit } v \text{ n'est pas un modèle de } \Gamma \cup \{\varphi\} \\ \text{soit } [\![\varphi]\!]_v = V\end{array} \\
&\Leftrightarrow \quad \forall \text{ valuation } v, \;\Big|\; \begin{array}{l}\text{soit } v \text{ n'est pas un modèle de } \Gamma \\ \text{soit } [\![\varphi]\!]_v = F, \text{ soit } [\![\varphi]\!]_v = V = [\![\psi]\!]_v\end{array} \\
&\Leftrightarrow \quad \forall \text{ valuation } v, \;\Big|\; \begin{array}{l}\text{soit } v \text{ n'est pas un modèle de } \Gamma \\ \text{soit } [\![\varphi \to \psi]\!]_v = V\end{array} \\
&\Leftrightarrow \quad \Gamma \vDash \varphi \to \psi \quad \blacksquare
\end{aligned}$$

On procède alors par récurrence en remarquant que

$$\forall \varphi_1, \varphi_2, \; (\varphi_1 \wedge \varphi_2) \to \varphi \text{ et } \varphi_1 \to (\varphi_2 \to \varphi)$$

ont les mêmes modèles ■

Remarque : cela ne s'exprime que pour les ensembles finis de formules, car les formules sont des objets finis. Le théorème de compacité (H.P) permet toujours de se ramener à un ensemble fini de formules.

3.2.6 Définition (*équivalence*)

Soient φ, ψ deux formules.
On dit que φ et ψ sont équivalentes, noté $\varphi \equiv \psi$, si φ et ψ ont les mêmes modèles.

3.2.7 Remarque

- $\varphi \equiv \psi \;\Leftrightarrow\; (\varphi \vDash \psi \text{ et } \psi \vDash \varphi) \;\Leftrightarrow\; \varphi \leftrightarrow \psi$ (d'après 3.2.1, p.11)
- $\equiv$ est une relation d'équivalence
- Deux formules sont équivalentes si elles ont la même table de vérité, donc il y a au plus $2^{(2^n)}$ classes d'équivalences pour des formules à n variables.

On verra comment construire une formule associée à chaque table de vérité possible.

3.2.8 Équivalences classiques

- Élément neutre :

$$\varphi \wedge \top \equiv \varphi \qquad \varphi \vee \bot \equiv \varphi$$

- Élément absorbant :

$$\varphi \wedge \bot \equiv \bot \qquad \varphi \vee \top \equiv \top$$

- Commutativité :

$$\varphi_1 \wedge \varphi_2 \equiv \varphi_2 \wedge \varphi_1 \qquad \varphi_1 \vee \varphi_2 \equiv \varphi_2 \vee \varphi_1$$

- Associativité :

$$\varphi_1 \wedge (\varphi_2 \wedge \varphi_3) \equiv (\varphi_1 \wedge \varphi_2) \wedge \varphi_3 \qquad \varphi_1 \vee (\varphi_2 \vee \varphi_3) \equiv (\varphi_1 \vee \varphi_2) \vee \varphi_3$$

- Distributivité :

$$\varphi_1 \wedge (\varphi_2 \vee \varphi_3) \equiv (\varphi_1 \wedge \varphi_2) \vee (\varphi_1 \wedge \varphi_3)$$

$$\varphi_1 \vee (\varphi_2 \wedge \varphi_3) \equiv (\varphi_1 \vee \varphi_2) \wedge (\varphi_1 \vee \varphi_3)$$

- Idempotence :

$$\varphi \wedge \varphi \equiv \varphi \qquad \varphi \vee \varphi \equiv \varphi$$

- Involutivité :

$$\neg\neg\varphi \equiv \varphi$$

- Règles de complément (tiers exclu) :

$$\varphi \wedge \neg\varphi \equiv \bot \qquad \varphi \vee \neg\varphi \equiv \top$$

- Lois de De Morgan :

$$\neg(\varphi \wedge \psi) \equiv \neg\varphi \vee \neg\psi \qquad \neg(\varphi \vee \psi) \equiv \neg\varphi \wedge \neg\psi$$

- Décomposition de l'implication :

$$\varphi \to \psi \equiv \neg(\varphi \wedge \neg\psi) \equiv \neg\varphi \vee \psi$$

- Contraposition :

$$\varphi \to \psi \equiv \neg\psi \to \neg\varphi$$

- Currification :

$$(\varphi_1 \wedge \varphi_2) \to \varphi_3 \equiv \varphi_1 \to (\varphi_2 \to \varphi_3)$$

3.2.9 Remarque

Ces équivalences définissent des règles de simplification qui présentent la valeur logique des formules. Ces règles peuvent aussi s'appliquer aux sous formules d'une formule donnée. On dit que l'équivalence passe au contexte et que c'est une relation de congruence. Ces règles permettent de donner une forme particulière aux formules afin de simplifier l'étude de leur satisfiabilité.

3.3 Formes normales

3.3.1 Définition (*littéraux, classes, formes normales*)

– Un *littéral* est une formule qui est soit une variable, soit la négation d'une variable ;

– Une *classe conjonctive* (resp. *disjonctive*) est une conjonction (resp. disjonction) de littéraux ;

– Une formule sous *forme normale disjonctive* (resp. *conjonctive*) est une disjonction (resp. conjonction) de classes conjonctives (resp. disjonctives).

3.3.2 Exemple

$(\neg x \wedge \neg y) \vee (\neg x \wedge \neg z \wedge t)$ est une FND (Forme Normale Disjonctive)
$(x \vee y) \wedge (\neg x \vee \neg y)$ est une FNC (c'est XOR)

3.3.3 Théorème

> Toute formule est équivalente à une forme normale conjonctive (resp. disjonctive).

□ Démonstration

(1) Il suffit de montrer le théorème pour les FND :

Soit φ une formule. Alors $\neg\varphi$ est équivalente à une FND.

Cette FND, $\bigvee_{i=1}^{n} c_i$ où $\forall i \in [\![1\ ;\ n]\!]$, la classe c_i s'écrit $\bigwedge_{j=1}^{m_i} l_{i,j}$ avec les $l_{i,j}$ des littéraux.

Donc

$$\varphi \equiv \neg\neg\varphi \equiv \neg\left(\bigvee_{i=1}^{n} c_i\right) \equiv \bigwedge_{i=1}^{n} \neg c_i$$

$$\equiv \bigwedge_{i=1}^{n} \neg\left(\bigwedge_{j=1}^{m_i} l_{i,j}\right) \equiv \bigwedge_{i=1}^{n}\bigvee_{j=1}^{m_i} \neg l_{i,j}$$

$$\equiv \underbrace{\bigwedge_{i=1}^{n}\bigvee_{j=1}^{m_i} \widetilde{l_{i,j}}}_{\text{FNC}}$$

où

$$\forall (i,j),\ \widetilde{l_{i,j}} = \begin{cases} \neg x_{i,j} & \text{si } l_{i,j} = x_{i,j} \in \mathcal{V} \\ x_{i,j} & \text{si } l_{i,j} = \neg \underbrace{x_{i,j}}_{\in \mathcal{V}} \end{cases}$$

(2) Soit φ une formule. Montrons que φ est équivalente à une FND.
Soit v une valuation. On construit la classe

$$c_v = \bigwedge_{\substack{x\in\mathcal{V}\\ v(x)=V}} x \wedge \bigwedge_{\substack{x\in\mathcal{V}\\ v(x)=F}} \neg x$$

(classe conjonctive)
Exo : v est l'unique modèle de c_v.

Alors $\varphi \equiv \bigvee_{v \mid [\![\varphi]\!]_v=V} c_v$ (FND) ■

3.3.4 Remarque

– On a démontré qu'il existe une formule associée à toute table de vérité.

– La normalisation d'une formule peut réutiliser une formule de taille exponentielle en la taille de la formule initiale.
Exemple :

$$\varphi = \bigwedge_{k=1}^{n} (x_k \wedge y_k)$$

est une FNC (dont la FND est de taille exponentielle en n)
Exo cette formule admet 3^n modèles.

– Si on ne s'intéresse qu'à la satisfiabilité d'une formule, il est inutile de préserver l'équivalence : il suffit de considérer une formule qui est équistaisfiable, *i.e* satisfiable si et seulement si la formule initiale est satisfiable.

3.3.5 Transformation de Tseitin (H.P)

Idée : on ajoute de nouvelles variables qui seront "équivalentes" aux sous-formules.
Soit φ une formule. $\forall \psi \in SF(\varphi)$, on ajoute une variable p_ψ et on construit une FNC f_ψ définie inductivement par :

$$\begin{array}{lcl}
f_x & = & (\neg p_x \vee x) \wedge (p_x \vee \neg x) \\
f_{\neg\psi} & = & (\neg p_{\neg\psi} \vee \neg p_\psi) \wedge (p_{\neg\psi} \vee p_\psi) \\
f_{\psi_1\vee\psi_2} & = & (\neg p_{\psi_1\vee\psi_2} \wedge p_{\psi_1} \vee p_{\psi_2}) \wedge (p_{\psi_1\vee\psi_2} \vee \neg p_{\psi_1}) \wedge (p_{\psi_1\vee\psi_2} \vee \neg p_{\psi_2}) \\
f_{\psi_1\wedge\psi_2} & = & (\neg p_{\psi_1\wedge\psi_2} \vee p_{\psi_1}) \wedge (\neg p_{\psi_1\wedge\psi_2} \vee p_{\psi_2}) \wedge (p_{\psi_1\wedge\psi_2} \vee \neg p_{\psi_1} \vee \neg p_{\psi_2})
\end{array}$$

On suppose avoir éliminé les simplifications ce qui ne fait que doubler la taille de la formule dans le pire cas.

La transformée de Tseitin de φ est alors

$$p_\varphi \wedge \bigwedge_{\psi\in SF(\varphi)} f_\psi$$

qui est une FNC.

Remarque : la transformée de Tseitin se calcule en temps linéaire en la taille de la formule.

Proposition (admise) : φ et $p_\varphi \wedge \bigwedge_{\psi \in SF(\varphi)} f_\psi$ sont équisatisfiable.

Idée : f_ψ exprime $\psi \leftrightarrow p_\psi$.

4 Problème de la satisfiabilité des formules propositionnelles

4.1 Présentation du problème

4.1.1 Définition (*Problème SAT*)

Le problème SAT est le problème de décision suivant : étant donné une formule φ de la logique propositionnelle, φ est-elle satisfiable ?

4.1.2 Remarque

Il existe un algorithme pour répondre à ce problème (on dit que SAT est décidable), mais l'algorithme vu en 3.1.8 (p.11) est de complexité exponentielle.

Question : peut-on faire mieux ?

4.1.3 Proposition

> Une clause conjonctive est une antinomie (= antilogie) $\Leftrightarrow$ elle contient une variable et sa négation.

□ Démonstration :

$\boxed{\Leftarrow}$ Si la clause s'écrit $x \wedge \neg x \wedge c$ (à l'ordre près des littéraux), alors

$$\forall v,\ [\![x \wedge \neg x \wedge c]\!]_v = F$$

car $[\![x \wedge \neg x]\!]_v = F$.

$\boxed{\Rightarrow}$ Par contraposée, on suppose que la clause ne contient pas une variable et sa négation. On montre qu'elle est satisfiable.

On peut supposer que les littéraux portent sur des variables deux à deux distinctes. ($x \wedge x = x$ et $\neg x \wedge \neg x = \neg x$)

On note alors $l_1, \ldots, l_n$ les littéraux, et $x_1, \ldots, x_n$ les variables associées.

On définit une valuation v par

$$\forall i \in [\![1\ ;\ n]\!],\ v(x_i) = \begin{cases} V \text{ si } l_i = x_i \\ F \text{ si } l_i = \neg x_i \end{cases}$$

Alors

$$\left[\!\left[\bigwedge_{i=1}^{n} l_i \right]\!\right]_v = V$$

donc la clause est satisfiable ■

4.1.4 Corollaire

> On peut résoudre SAT en temps polynomial pour les FND

□ Démonstration :

Une FND est satisfiable $\Leftrightarrow$ l'une de ses clauses est satisfiable

$\Leftrightarrow$ l'une de ses clauses ne contient pas une variable et sa négation.

Vérifier si une clause ne contient pas une variable et sa négation se fait en temps $\mathcal{O}(n^2)$ si la clause contient n littéraux, voire $\mathcal{O}(n)$ si les variables sont représentées par des entiers.

On applique cette vérification à chaque clause ■

4.1.5 Remarque

– On ne sait pas s'il est possible de résoudre SAT en temps polynomial (*cf* cours sur la classe NP).

– C'est déjà le cas pour certains cas particuliers de SAT.

4.1.6 Définition (k-SAT)

Étant donné $k \in \mathbb{N}^*$, k-SAT est le problème de décision suivant : étant donné une formule φ sous FNC telle que chaque clause contient au plus k littéraux, φ est-elle satisfiable ?

4.1.7 Remarque

– 1-SAT est résoluble en temps polynomial (*cf* 4.1.3, p.17)

– On verra que 2-SAT est résoluble en temps polynomial (*cf* cours sur la théorie des graphes).

– La transformation de Tseitin fournit pour toute formule φ une instance de 3-SAT qui est de taille linéaire en la taille de φ et qui est équisatifiable à φ, donc on ne sait pas résoudre k-SAT en temps polynomial pour $k \geqslant 3$.

– SAT et ses restrictions sont des problèmes importants car ils servent à modéliser de nombreux problèmes. On parle de réduction de ces problèmes à SAT.

4.1.8 Exemple de réduction à SAT

Le problème du Sudoku

• Rappel des règles : on dispose d'une grille $n \times n$ qu'on veut remplir avec des entiers de $[\![1\ ;\ n]\!]$ de telle sorte que chaque ligne / colonne / bloc de taille $l \times l$, où $n = l^2$ contienne chaque entier exactement une fois.

Une grille peut être partiellement pré-remplie afin qu'il n'existe qu'une seule solution.

• Réduction à SAT : on introduit n^3 variables propositionnelles $(x_{i,j}^k)_{i,j,k \in [\![1\ ;\ n]\!]}$.

Idée : $x_{i,j}^k$ signifie "la case (i,j) contient l'entier k".

On peut représenter les règles à l'aide de formules :

– Chaque case contient un unique chiffre :

$$\bigwedge_{i=1}^{n} \bigwedge_{j=1}^{n} \bigvee_{k=1}^{n} \left(x_{i,j}^k \wedge \bigwedge_{k' \neq k} \neg x_{i,j}^{k'} \right)$$

– Chaque ligne contient au moins une fois chaque chiffre :

$$\bigwedge_{k=1}^{n} \bigwedge_{i=1}^{n} \bigvee_{j=1}^{n} x_{i,j}^k$$

– Chaque ligne contient au plus une fois chaque chiffre :

$$\bigwedge_{i=1}^{n} \bigwedge_{j=1}^{n} \bigwedge_{k=1}^{n} \left(x_{i,j}^k \to \bigwedge_{j' \neq j} \neg x_{i,j'}^k \right)$$

– Ect.

($\bigwedge$ est la version finie de $\forall$, et $\bigvee$ est la version finie de $\exists$)

On peut représenter les cases pré-remplies par des clauses unitaires $x_{i,j}^k$ (si la case (i,j) est pré-remplie avec k)

La conjonction de toutes ces formules donne une instance de SAT telle que tout modèle de cette conjonction représente une solution.

4.2 Résolution du problème SAT

4.2.1 Algorithme de Quine

L'algorithme de Quine est un algorithme de résolution de SAT pour les FNC.

Idée : on teste toutes les valuations possibles, mais en construisant de manière incrémentale les valuations en simplifiant la formule lors de la construction.

Si $v(x) = V$ et si une clause c contient x, alors $[\![c]\!]_v = V$ donc on peut retirer c de la formule.

Si $v(x) = V$ et si une clause c contient $\neg x$, alors $[\![c]\!]_v = [\![c \setminus \{\neg x\}]\!]_v$ donc on peut retirer $\neg x$ de c.

Cas symétrique si $v(x) = F$.

Si on parvient à retirer toutes les clauses, alors la formule est satisfiable.

Si on retire tous les littéraux d'une clause, alors elle n'est pas satisfiable dans la valuation partielle : il faut donc changer de valuation partielle en revenant sur les choix précédents.

Algorithme :

Entrée : ensemble de clauses disjonctives C

> fonction Quine(C) :
>
> Si $C = \varnothing$ alors renvoyer vrai
>
> Si la clause vide $\in C$, alors envoyer faux
>
> Choisir x, une variable apparaissant dans une clause de C
>
> $C' \leftarrow C$, où toute clause contenant x a été supprimée et où $\neg x$ a été retiré de toutes les clauses.
>
> Si Quine(C') alors renvoyer vrai
>
> Sinon : *// on choisit $v(x) = F$*
>
> $C'' \leftarrow C$ où toute clause contenant $\neg x$ a été supprimée et où x a été retiré de toutes les clauses
>
> Renvoyer Quine(C'')

Dans le pire cas, on teste toutes les valuations, d'où la complexité exponentielle.

4.2.2 Implémentation de l'algorithme de Quine

On peut représenter les clauses par des listes de littéraux et les FNC par des listes de clauses.

```
type litteral =
    | Var of string
    | NVar of string

type clause = litteral list
type fnc = clause list

let rec quine (f : fnc) : bool =
    match f with
    | [] -> true
    | _ when List.mem [] f -> false
    | (Var x :: _) :: _ | (NVar x :: _) :: _ ->
        if quine (List.map (List.filter (fun l -> l <> NVar x))
           (List.filter (fun c -> not (List.mem (Var x) c)) f)
        ) then true
        else quine (List.map (List.filter (fun l -> l <> Var x))
            (List.filter (fun c -> not (List.mem (NVar x) c)) f)
        )
    | _ -> failwith "Impossible"
```

4.2.3 Optimisations

On peut utiliser plusieurs techniques afin de rendre cet algo plus efficace :

– Propagation des clauses unitaire : si une clause ne contient plus qu'un seul littéral, la valeur de ce littéral est imposée. On peut donc effectuer les simplifications associées à cette valeur sans avoir besoin de revenir sur ce choix.

Cela donne l'algorithme DPLL (Davis Putnam Logemann Loveland) (H.P)

– On peut utiliser des heuristiques pour le choix du littéral, par exemple le littéral le plus fréquent, celui qui apparaît dans une clause la plus courante possible, ou un littéral choisi au hasard.

– On peut ajouter des clauses à la formule pour tenir compte des échecs : une valuation partielle correspond à un ensemble de choix.

– On peut "apprendre" une clause pour ne pas refaire de choix similaire dans un autre appel récursif.

Cela donne l'algorithme CDCL (Conflict Driver Clause Learning) (H.P)

DPLL et CDCL sont à la base des SAT-solvers modernes.

4.3 Compléments d'algorithmique

4.3.1 Introduction

L'algorithme de Quine est un cas particulier d'algorithme de recherche par *force brute*. Le principe de la recherche par force brute, ou exploration exhaustive, est de tester toutes les valeurs possibles pour en trouver une qui répond au problème que l'on se pose.

On peut appliquer cette méthode pour résoudre des problèmes de décision (existe-t-il une valeur qui satisfait des contraintes données ?) ou des problèmes d'optimisation (trouver une valeur qui satisfait certaines contraintes tout en maximisant / minimisant une certaine quantité)

Avantages	Inconvénients
– Méthode complète : on trouve nécessairement une solution (optimale) si elle existe ; – Souplesse : on peut aisément modifier l'algorithme à l'aide d'heuristiques pour le rendre plus efficace dans certains cas ; – Méthode simple : implémentation aisée et simple à débugger ; – On peut trouver toutes les solutions possibles, si c'est ce que l'on cherche.	– Complexité élevée (ex : trouver la clé secrète d'un schéma cryptographique, encodée sur n bits nécessite 2^n essais)

4.3.2 Modalité d'implémentation

Un algorithme de recherche par force brute s'écrit simplement ainsi :

> Pour toute valeur v :
> Si v satisfait les contraintes :
> S'arrêter avec v comme solution
> Échec de la recherche.

Version "toutes les solutions" :

> $S \leftarrow \varnothing$
> Pour toute valeur v :
> Si v satisfait les contraintes :
> $S \leftarrow S \cup \{v\}$
> Renvoyer S

Pour l'implémentation d'un tel algorithme, le paradigme de programmation et la manière dont on parcourt les valeurs contraignent la façon dont on écrit le code.

Pour une implémentation récursive, on peut manipuler l'ensemble des valeurs à traiter en retirant une à chaque appel récursif (coûteux en espace), ou, si l'on dispose d'une manière d'énumérer les valeurs en choisissant la valeur qui suit une valeur donnée, on peut partir d'une valeur initiale et faire un appel récursif sur la suivante jusqu'à exhaustion des valeurs (ex : listes chaînées, ou parcourt des indices d'un tableau par une fonction récursive).
Si on dispose d'une telle manière d'énumérer les valeurs, on peut facilement traduire l'algorithme récursif en boucle `while` (tant que l'on n'a pas trouver de solution et que l'on n'a pas parcouru toutes les valeurs, passer à la valeur suivante) ou une boucle `for` (pour i de 1 au nombre de valeurs, tester la valeur n°i).

Exemple : on veut savoir si x apparaît dans t de taille n :

Boucle `while` :

```
bool trouve = false;
int i = -1;
while (i < n - 1 && !trouve) {
    i++;
    trouve = t[i] == x;
}
//test de trouve
```

Boucle `for` :

```
int i;
for (i = 0 ; i < n ; i++) {
    if (t[i] == x)
        break;
}
//test de i
```

Remarques :

– On peut aussi simplifier la boucle `while` avec l'usage de `break`.

– Il n'y a pas d'instruction `break` en OCaml, mais on peut contrôler l'exécution à l'aide d'une exception :

```
exception Trouve of int
```

```
try
    for i = 0 to n - 1 do
        if t.(i) = x then
            raise (Trouve i)
    done;
    raise Not_found
with
| Trouve i -> i
```

4.3.3 Retour sur trace (*backtracking*)

L'algorithme de Quine correspond à une autre manière de parcourir l'ensemble des valeurs : on construit de manière incrémentale des valeurs partielles (valuation d'un sous-ensemble des variables), en passant d'une valeur partielle à une autre en effectuant un choix (choix de la valeur d'une variable) sur lequel on reviendra pour tester d'autres choix si celui-ci ne permet pas de trouver une solution.

On appelle cette méthode le *retour sur trace* (ou *backtracking*).

Le caractère incrémental impose de ne pas pouvoir boucler sur un ensemble de valeurs partielles et peut permettre d'éliminer des choix avant d'avoir construit une valeur complète.

Pseudo-code :

> Recherche(v_{init}) :
> Si v_{init} est complète et satisfait les contraintes :
> S'arrêter avec v_{init} comme solution
>
> Si v_{init} est partielle :
> Pour chaque choix valide à partir de v_{init} :
> $v_{\text{next}} \leftarrow$ résultat du choix
> Recherche(v_{next})

Exemple : Sudoku

Valeur : remplissage de la grille (pas forcément selon les règles)

Valeur partielle : remplissage partiel de la grille : il peut y avoir des cases vides

v_{init} : problème posé

Choix : écriture d'un chiffre dans une case vide

Choix valide : choix qui respecte les règles du jeu vis-à-vis des chiffres déjà inscrits.

Avec ces définitions, une valeur complète construite à l'aide d'une succession de choix valides est nécessairement une solution.

4.3.4 Optimisations

– Comme pour l'algorithme DPLL par rapport à l'algorithme de Quine, certains choix peuvent contraindre la complétion des valeurs partielles de telle sorte qu'il n'est pas

nécessaire de revenir sur les choix “imposés” ;

– On peut également parcourir qu’un sous-ensemble des valeurs possibles.

C’est un procédé classique en géométrie algorithmique : l’ensemble des valeurs à parcourir est en général construit à partir d’un ensemble de point du plan / de l’espace. Parcourir l’ensemble des points selon une direction permet de ne considérer que certaines valeurs. On appelle cela un algorithme par *droite de balayage*.

Idée : une droite / un plan perpendiculaire à la direction choisie balaye l’ensemble des points, et à chaque point rencontré, on effectue des opérations visant à construire une solution.

Exemple : retour sur le TP_{12}

Problème : étant donné un ensemble de points du plan, trouver une paire de points qui minimise la distance euclidienne entre ces points.

Ensemble des valeurs : l’ensemble des paires de points $\rightarrow$ algorithme force brute en $\mathcal{O}(n^2)$

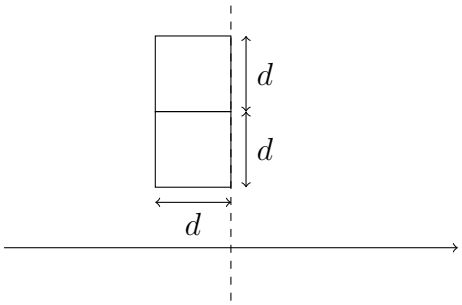
Rappel : algorithme “diviser pour régner” en $\mathcal{O}(n \log n)$ vu en TP.

Idée d’algorithme par droite de balayage : on parcourt les points par abscisse croissante en conservant la distance minimale courante d.

On note $(x_0, y_0), \ldots, (x_{n-1}, y_{n-1})$ les points triés par abscisse.

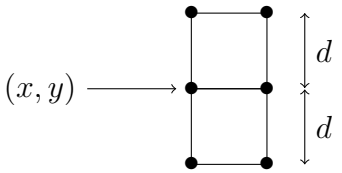
Si la distance minimale vaut d après le parcourt des k premiers points, il est inutile de considérer les points d’abscisse $< x_k - d$ pour la mise à jour de d.

De même, il est inutile de considérer les points d’ordonnée $< y_k - d$ ou $> y_k + d$

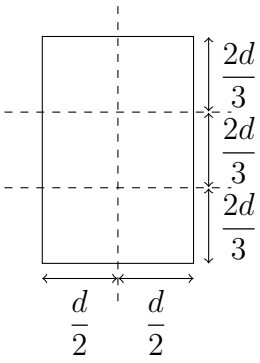


Attention : si les points d’abscisse $< x_k - d$ peuvent être définitivement oubliés, ceux d’abscisse $\geqslant x_k - d$ doivent être conservés même s’ils sont hors du rectangle dans l’éventualité d’un usage avec (x_{k+1}, y_{k+1})

Remarque : le rectangle autour du point courant contient au plus 5 points en plus du point courant.



S’il y en avait plus, l’un des rectangles de dimension $\dfrac{2d}{3} \cdot \dfrac{d}{2}$ contiendrait deux points.



La distance maximale entre 2 points dans un tel rectangle est :

$$\sqrt{\left(\frac{2d}{3}\right)^2+\left(\frac{d}{2}\right)^2}=\sqrt{\frac{4d^2}{9}+\frac{d^2}{4}}=\sqrt{\frac{25}{36}}d=\frac{5}{6}d<d$$

impossible.

On en déduit l'algorithme suivant :

> Trier les points par abscisse croissante, les nommer $(x_0,y_0),\dots,(x_{n-1},y_{n-1})$
> $E\leftarrow\{(x_0,y_0),(x_1,y_1)\}$
> $d\leftarrow \text{dist}((x_0,y_0),(x_1,y_1))$
> Pour k de 2 à $n-1$:
> Retirer de E les points d'abscisse $< x_k-d$ $(*)$
> Pour chaque point actif (x,y) tq $y_k-d\leqslant y\leqslant y_k+d$: $(**)$
> $d\leftarrow \min(d,\ \text{dist}((x_k,y_k),(x,y)))$
> $E\leftarrow E\cup\{(x_k,y_k)\}$
> Renvoyer d.

Complexité :

– $\mathcal{O}(n\log n)$ pour le tri

– Manipulation de E : cela dépend de la structure de données

$(*)$ inutile de faire un parcourt de E : il suffit de parcourir la liste des points triés en se souvenant du dernier point retiré.

Coût total de cette ligne : coût de n supressions dans E.

On aura aussi le coût de n insertions dans E

$(**)$ nécessite une structure organisée selon les ordonnées : on peut choisir pour E une structure d'ABR équilibrée : coût d'une insertion (suppression en $\mathcal{O}(\log n)$), coût de la recherche des (au plus) 5 éléments en $(**)$: $\mathcal{O}(\log n)$ Exo

Au total $\mathcal{O}(n\log n)$.