INFO-F-302, Cours d'Informatique Fondamentale

Emmanuel Filiot
Département d'Informatique
Faculté des Sciences
Université Libre de Bruxelles

Année académique 2023-2024

Aux origines, la logique

- **Théorème d'incomplétude de Gödel** (1931) : il existe des énoncés [1] valides qui ne sont pas démontrables.

1. écrits en logique des prédicats

Aux origines, la logique

- **Théorème d'incomplétude de Gödel** (1931) : il existe des énoncés [1] valides qui ne sont pas démontrables.
- Gödel avait posé une question : **peut-on automatiser les mathématiques ?**
  Est-ce qu'il existe un procédé automatique (un algorithme) qui, étant donnée un énoncé, permet de décider s'il est vrai ou non ?

1. écrits en logique des prédicats

Aux origines, la logique

- **Théorème d'incomplétude de Gödel** (1931) : il existe des énoncés [1] valides qui ne sont pas démontrables.
- Gödel avait posé une question : **peut-on automatiser les mathématiques ?**
  Est-ce qu'il existe un procédé automatique (un algorithme) qui, étant donnée un énoncé, permet de décider s'il est vrai ou non ?
- Turing y répondit négativement (1936). En particulier :
  - il a défini la notion d'algorithme par un modèle abstrait d'ordinateur (machines de Turing)
  - il a démontré qu'il n'est pas possible de décider algorithmiquement qu'une machine de Turing termine ou non (problème de l'arrêt)
  - la propriété $P_i =$ "la i-ème machine s'arrête" est définissable en logique (pour une certaine "numérotation" des machines).
  - ceci répond à la question de Gödel : on ne peut prouver la propriété P_i (ou sa négation) algorithmiquement.

1. écrits en logique des prédicats

Théorie de la complexité

- Turing a démontré que certains problèmes (le problème de l'arrêt) sont indécidables
- lorsqu'un problème est décidable algorithmiquement, on souhaiterait en savoir plus : combien d'opérations sont nécessaires intrinsinquèment ? avec quelle mémoire ?
- ceci a donné naissance à la théorie de la complexité : classifier les problème selon leur complexité algorithmique intrinsèque (en temps et en mémoire)

Théorie de la complexité

- Turing a démontré que certains problèmes (le problème de l'arrêt) sont indécidables
- lorsqu'un problème est décidable algorithmiquement, on souhaiterait en savoir plus : combien d'opérations sont nécessaires intrinsinquèment ? avec quelle mémoire ?
- ceci a donné naissance à la théorie de la complexité : classifier les problème selon leur complexité algorithmique intrinsèque (en temps et en mémoire)
- dans une classe de complexité donnée, certains problèmes sont aussi difficiles que tous les autres (on dit qu'ils sont complets pour la classe)
- c'est le cas du problème de satisfiabilité en logique booléenne (il est complet pour la classe NP, prouvé en 1971 par Cook).
- intuitivement, si on peut résoudre un problème complet pour une classe, on peut résoudre tous les autres problèmes de la classe (avec la même "complexité" asymptotique).

Informatique Fondamentale

Un des objectifs du cours est de comprendre de manière plus précise ce qui a été dit précédemment :

- qu'est-ce que la logique ?
- qu'est-ce qu'une preuve ?
- qu'est-ce qu'un problème indécidable ? comment peut-on le démontrer ?
- qu'est-ce qu'une classe de complexité ? qu'est-ce qu'NP ?
- qu'est-ce qu'un problème complet pour NP ? comment peut-on le démontrer ?

Informatique Fondamentale

Un des objectifs du cours est de comprendre de manière plus précise ce qui a été dit précédemment :

- qu'est-ce que la logique ?
- qu'est-ce qu'une preuve ?
- qu'est-ce qu'un problème indécidable ? comment peut-on le démontrer ?
- qu'est-ce qu'une classe de complexité ? qu'est-ce qu'NP ?
- qu'est-ce qu'un problème complet pour NP ? comment peut-on le démontrer ?

Informatique fondamentale et modélisation

L'informatique fondamentale est une branche de l'informatique dont le principal but est la conception de **modèles** pour des problèmes, concepts ou applications informatiques, et le développement **d'algorithmes** pour analyser ces modèles.

Exemples de modèles

- graphes : modèles de réseaux – sociaux, routiers, web, etc. –, de dépendances (entre les variables d'un programme, entre les classes en programmation objet, etc.), ...

Exemples de modèles

- graphes : modèles de réseaux – sociaux, routiers, web, etc. –, de dépendances (entre les variables d'un programme, entre les classes en programmation objet, etc.), ...
- la logique : modélise les énoncés mathématiques, les propriétés d'un système informatique, d'un programme, etc.

Exemples de modèles

- graphes : modèles de réseaux – sociaux, routiers, web, etc. –, de dépendances (entre les variables d'un programme, entre les classes en programmation objet, etc.), ...
- la logique : modélise les énoncés mathématiques, les propriétés d'un système informatique, d'un programme, etc.
- les systèmes de déduction : modélisent le raisonnement mathématique via la notion de preuve

Exemples de modèles

- graphes : modèles de réseaux – sociaux, routiers, web, etc. –, de dépendances (entre les variables d'un programme, entre les classes en programmation objet, etc.), ...
- la logique : modélise les énoncés mathématiques, les propriétés d'un système informatique, d'un programme, etc.
- les systèmes de déduction : modélisent le raisonnement mathématique via la notion de preuve
- les machines de Turing : modèle de calcul

Exemples de modèles

- graphes : modèles de réseaux – sociaux, routiers, web, etc. –, de dépendances (entre les variables d'un programme, entre les classes en programmation objet, etc.), ...
- la logique : modélise les énoncés mathématiques, les propriétés d'un système informatique, d'un programme, etc.
- les systèmes de déduction : modélisent le raisonnement mathématique via la notion de preuve
- les machines de Turing : modèle de calcul
- les automates : modélisent des programmes informatiques "simples"

Exemples de modèles

- graphes : modèles de réseaux – sociaux, routiers, web, etc. –, de dépendances (entre les variables d'un programme, entre les classes en programmation objet, etc.), ...
- la logique : modélise les énoncés mathématiques, les propriétés d'un système informatique, d'un programme, etc.
- les systèmes de déduction : modélisent le raisonnement mathématique via la notion de preuve
- les machines de Turing : modèle de calcul
- les automates : modélisent des programmes informatiques "simples"
- chaînes de Markov : modélisent des systèmes probabilistes

Exemples de modèles

- graphes : modèles de réseaux – sociaux, routiers, web, etc. –, de dépendances (entre les variables d'un programme, entre les classes en programmation objet, etc.), ...
- la logique : modélise les énoncés mathématiques, les propriétés d'un système informatique, d'un programme, etc.
- les systèmes de déduction : modélisent le raisonnement mathématique via la notion de preuve
- les machines de Turing : modèle de calcul
- les automates : modélisent des programmes informatiques "simples"
- chaînes de Markov : modélisent des systèmes probabilistes
- etc.

Une notion centrale du cours : réductions entre problèmes

- une notion essentielle en informatique fondamentale
- modélisation d'un problème par un autre

Pour les problèmes de décision (réponse oui/non)

Réduire un problème A vers un problème B, c'est trouver une méthode permettant d'encoder toute entrée I_A du problème A comme une entrée I_B du problème B, telle que

I_A a une solution (la réponse est oui) si et seulement si I_B a une solution

- parfois on demandera que cette méthode soit un algorithme
- pour les problèmes qui ne sont pas des problèmes de décision (trier un tableau par exemple), on demandera que toute solution de I_A s'encode en une solution de I_B et réciproquement toute solution de I_B se décode en une solution de I_A

Exemple – Problème B : Bin Packing

Problème Bin Packing (B) :

- **INPUT :** n objets de poids $p_1, \ldots, p_n$, k sacs de capacité C
- **OUTPUT :** oui ssi on peut ranger tous les objets dans au plus k sacs sans dépasser la capacité de chaque sac

Exemple – Problème B : Bin Packing

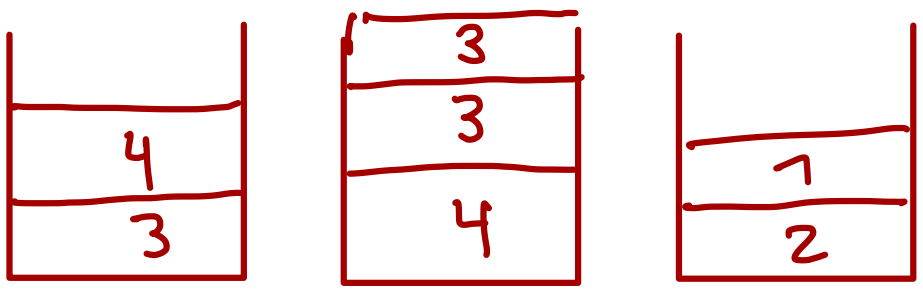
Problème Bin Packing (B) :

- **INPUT :** n objets de poids $p_1, \ldots, p_n$, k sacs de capacité C
- **OUTPUT :** oui ssi on peut ranger tous les objets dans au plus k sacs sans dépasser la capacité de chaque sac

On veut ranger les objets suivants dans des sacs de capacité 10kg.

Objets	1	2	3	4	5	6	7
Poids	3	4	4	3	3	2	1

Peut-on réussir avec 3 sacs ?



Exemple – Problème B : Bin Packing

Problème Bin Packing (B) :

- **INPUT :** n objets de poids $p_1, \ldots, p_n$, k sacs de capacité C
- **OUTPUT :** oui ssi on peut ranger tous les objets dans au plus k sacs sans dépasser la capacité de chaque sac

On veut ranger les objets suivants dans des sacs de capacité 10kg.

Objets	1	2	3	4	5	6	7
Poids	3	4	4	3	3	2	1

Peut-on réussir avec 3 sacs ? oui, on forme les sacs d'objets suivants : $\{1, 2\}, \{3, 4, 5\}, \{6, 7\}$.

Exemple – Problème B : Bin Packing

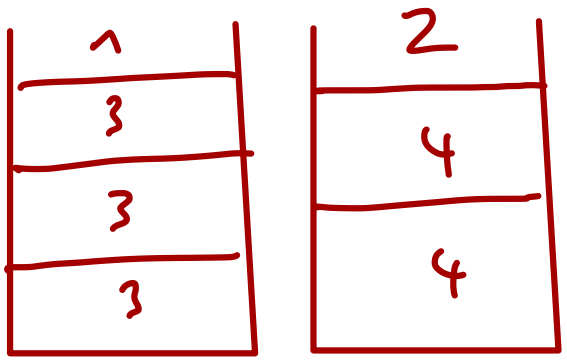
Problème Bin Packing (B) :

- **INPUT :** n objets de poids $p_1, \ldots, p_n$, k sacs de capacité C
- **OUTPUT :** oui ssi on peut ranger tous les objets dans au plus k sacs sans dépasser la capacité de chaque sac

On veut ranger les objets suivants dans des sacs de capacité 10kg.

Objets	1	2	3	4	5	6	7
Poids	3	4	4	3	3	2	1

Peut-on réussir avec 3 sacs ? oui, on forme les sacs d'objets suivants : $\{1, 2\}, \{3, 4, 5\}, \{6, 7\}$.
et avec 2 sacs ?



Exemple – Problème B : Bin Packing

Problème Bin Packing (B) :

- **INPUT :** n objets de poids $p_1, \ldots, p_n$, k sacs de capacité C
- **OUTPUT :** oui ssi on peut ranger tous les objets dans au plus k sacs sans dépasser la capacité de chaque sac

On veut ranger les objets suivants dans des sacs de capacité 10kg.

Objets	1	2	3	4	5	6	7
Poids	3	4	4	3	3	2	1

Peut-on réussir avec 3 sacs ? oui, on forme les sacs d'objets suivants : $\{1, 2\}, \{3, 4, 5\}, \{6, 7\}$.
et avec 2 sacs ?
oui : $\{1, 2, 4\}, \{3, 5, 6, 7\}$. Cette solution est optimale (on ne peut pas réussir avec un seul sac puisque la somme totale de tous les poids dépasse 10).

Problème A : 2-partition

Problème 2-partition (A) :

- **INPUT :** m entiers naturels $c_1, \ldots, c_m$ tels que $S := \sum_{i=1}^{m} c_i$ est paire,
- **OUTPUT :** oui ssi il existe $J \subseteq \{1, \ldots, m\}$ tel que $\sum_{i \in J} c_i = \sum_{i \notin J} c_i$.

Problème A : 2-partition

Problème 2-partition (A) :

- **INPUT :** m entiers naturels $c_1, \ldots, c_m$ tels que $S := \sum_{i=1}^{m} c_i$ est paire,
- **OUTPUT :** oui ssi il existe $J \subseteq \{1, \ldots, m\}$ tel que $\sum_{i \in J} c_i = \sum_{i \notin J} c_i$.

Exemple avec $1, 4, 2, 3, 2$: $1 + 2 + 3 = 2 + 4$.

Problème A : 2-partition

Problème 2-partition (A) :

- **INPUT :** m entiers naturels $c_1, \ldots, c_m$ tels que $S := \sum_{i=1}^{m} c_i$ est paire,
- **OUTPUT :** oui ssi il existe $J \subseteq \{1, \ldots, m\}$ tel que $\sum_{i \in J} c_i = \sum_{i \notin J} c_i$.

Exemple avec $1, 4, 2, 3, 2$: $1 + 2 + 3 = 2 + 4$.

Réduction vers BinPacking ?

Problème A : 2-partition

Problème 2-partition (A) :

- **INPUT :** m entiers naturels $c_1, \ldots, c_m$ tels que $S := \sum_{i=1}^{m} c_i$ est paire,
- **OUTPUT :** oui ssi il existe $J \subseteq \{1, \ldots, m\}$ tel que $\sum_{i \in J} c_i = \sum_{i \notin J} c_i$.

Exemple avec $1, 4, 2, 3, 2$: $1 + 2 + 3 = 2 + 4$.

Réduction vers BinPacking ? On prend $C = S/2$ et les objets $1, \ldots, m$ de poids $c_1, \ldots, c_m$, et $k = 2$. Il y a une solution à toute instance de 2-partition si et seulement si il y a en une à l'instance de BinPacking ainsi construite.

Problème A : 2-partition

Problème 2-partition (A) :

- **INPUT :** m entiers naturels $c_1, \ldots, c_m$ tels que $S := \sum_{i=1}^{m} c_i$ est paire,
- **OUTPUT :** oui ssi il existe $J \subseteq \{1, \ldots, m\}$ tel que $\sum_{i \in J} c_i = \sum_{i \notin J} c_i$.

Exemple avec $1, 4, 2, 3, 2$: $1 + 2 + 3 = 2 + 4$.

Réduction vers BinPacking ? On prend $C = S/2$ et les objets $1, \ldots, m$ de poids $c_1, \ldots, c_m$, et $k = 2$. Il y a une solution à toute instance de 2-partition si et seulement si il y a en une à l'instance de BinPacking ainsi construite.

Remarque : si nous disposons d'un algorithme pour résoudre le problème BinPacking, alors on peut l'utiliser pour résoudre le problème 2-partition !

9 Objectifs du cours

Etre capable

- de modéliser des problèmes de manière précise et rigoureuse en utilisant des langages logiques
- de définir des réductions entre problèmes
- d'identifier la classe de complexité algorithmique d'un problème
- de démontrer qu'un problème est intrinsèquement difficile algorithmiquement
- de construire et d'*analyser* des programmes simples représentables par automates

9 Objectifs du cours

Etre capable

- de modéliser des problèmes de manière précise et rigoureuse en utilisant des langages logiques
- de définir des réductions entre problèmes
- d'identifier la classe de complexité algorithmique d'un problème
- de démontrer qu'un problème est intrinsèquement difficile algorithmiquement
- de construire et d'*analyser* des programmes simples représentables par automates

+ vous préparer au master d'informatique (cours de calculabilité et complexité notamment)

10 Plan

1. modèles pour les énoncés mathématiques et le raisonnement : la logique et la déduction naturelle
2. modélisation de problèmes en logique propositionnelle
3. introduction aux notions de complexité de problèmes et de réduction entre problèmes
4. introduction à la calculabilité
5. modèles de programmes : les automates finis

11 Organisation pratique du cours

Annonces	Par email et reprises sur l'UV.
Matériel	Les slides constituent le syllabus. Ils seront disponibles sur l'UV.
Projet	**Pas de seconde session !**
Travaux pratiques	Assistants : Arnaud Leponce et Robin Petit
Evaluation	Examen écrit (3/4) et projet (1/4)
Contact	▶ **email** : `efiliot@ulb.be`

Logique Propositionnelle : Introduction

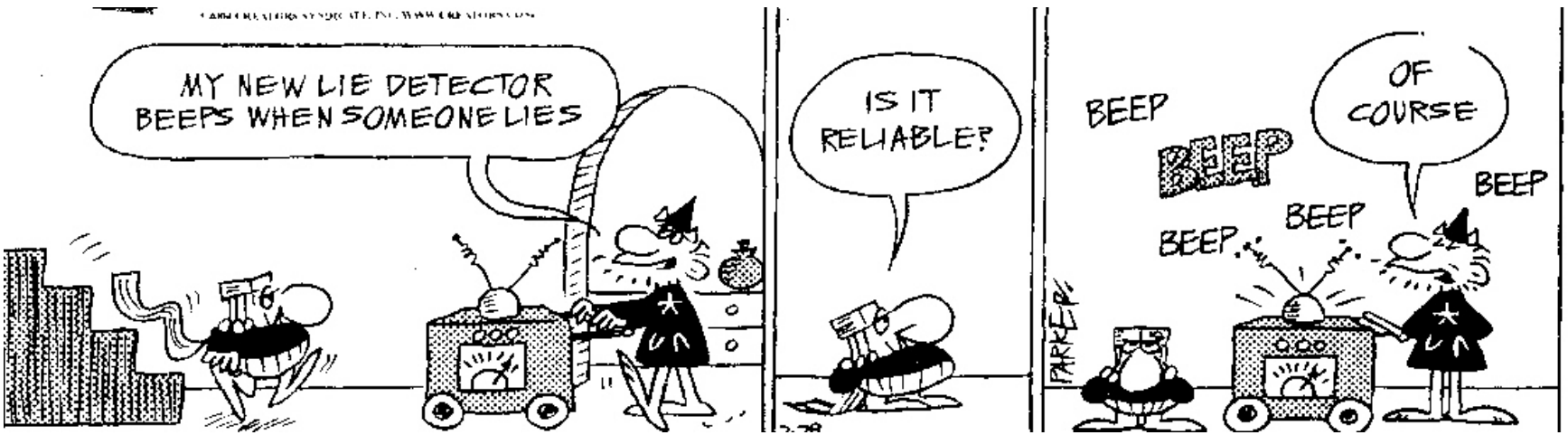
13 Qu'est-ce qu'un langage logique ?

- langage permettant de décrire avec précision et rigueur des énoncés et des raisonnements, basés notamment sur des connecteurs Booléens (et, ou, négation)
- langages équipés d'une sémantique claire et non-ambigüe
- les énoncés logiques seront appelés "formules"
- *but* : pallier à l'imprécision des langages naturels
- *but supplémentaire en informatique* : analyser de manière algorithmique les énoncés

14 Fausses vérités et paradoxes

- l'utilisation du langage naturel comme notation est imprécise et peut mener à des énoncés faux ...
  1. Tout ce qui est rare est cher
  2. *Or* une chose pas chère, c'est rare
  3. *Donc* une chose pas chère, c'est cher.
- ... ou à des paradoxes
  - Cette phrase est fausse. Problème de *l'auto référence*.
  - Le barbier rase tous les hommes qui ne se rasent pas eux-mêmes et seulement ceux-là. Qui rase le barbier ?

15 Un petit dernier ...



Exemples d'application de la logique en informatique

- design de circuits numériques
- vérification *hardware* et *software*, *model-checking*. Logiques temporelles.
- preuves de programmes
- bases de données : SQL.
- ...

17 Exemple de raisonnement

Considérons la situation décrite par les affirmations suivantes :

1. **Si** le train arrive en retard **et** il n'y a pas de taxis à la gare **alors** l'invité arrive en retard.
2. L'invité n'est pas en retard.
3. Le train est arrivé en retard.

Et la déduction suivante : il y avait des taxis à la gare.

17 Exemple de raisonnement

Considérons la situation décrite par les affirmations suivantes :

1. **Si** le train arrive en retard **et** il n'y a pas de taxis à la gare **alors** l'invité arrive en retard.
2. L'invité n'est pas en retard.
3. Le train est arrivé en retard.

Et la déduction suivante : il y avait des taxis à la gare.

Question

Pourquoi peut-on déduire qu'il y avait des taxis à la gare ?

Premièrement, si on met l'affirmation 1 et l'affirmation 3 ensemble, on peut affirmer que s'il n'y avait pas eu de taxis à la gare, alors l'invité serait arrivé en retard. D'après l'affirmation 2, l'invité n'est pas arrivé en retard. Donc il y avait des taxis à la gare.

18 Autre Exemple

Considérons un autre exemple :

1. **Si** il pleut **et** l'invité a oublié son parapluie **alors** l'invité est trempé.
2. L'invité n'est pas trempé.
3. Il pleut.

Et la déduction suivante : l'invité n'a pas oublié son parapluie.

18 Autre Exemple

Considérons un autre exemple :

1. **Si** il pleut **et** l'invité a oublié son parapluie **alors** l'invité est trempé.
2. L'invité n'est pas trempé.
3. Il pleut.

Et la déduction suivante : l'invité n'a pas oublié son parapluie.

Question

Pourquoi peut-on déduire que l'invité n'a pas oublié son parapluie ?

Premièrement, si on met l'affirmation 1 et l'affirmation 3 ensemble, on peut affirmer que si l'invité avait oublié son parapluie, alors il serait trempé. D'après l'affirmation 2, l'invité n'est pas trempé. Donc l'invité n'a pas oublié son parapluie.

19 Remarque

La deuxième démonstration suit la même **structure logique** que la première démonstration. Il suffit de remplacer les fragments de phrase comme suit :

	Exemple du train	Exemple du parapluie
	Le train est arrivé en retard	Il pleut
	Il y a des taxis à la gare	L'invité a son parapluie
	L'invité est en retard	L'invité est mouillé.

20 Formalisation Logique

Exemple du train	Exemple du parapluie	Proposition
Le train est arrivé en retard	Il pleut	p
Il y a des taxis à la gare	L'invité a son parapluie	q
L'invité est en retard	L'invité est mouillé	r

Démonstration

1. Hypothèse : si p et non q, alors r $(p \wedge \neg q) \rightarrow r$
2. Hypothèse : p
3. Hypothèse : non r
4. Déduction : si non q alors r $\neg q \rightarrow r$
5. Déduction : comme non r, alors q.

21 Formalisation logique

Nous pouvons *formaliser* les trois hypothèses de chacun de ces deux premiers exemples par la *formule logique* suivante :

$$((p \wedge \neg q) \rightarrow r) \wedge \neg r \wedge p$$

Et nous pouvons *formaliser* la déduction par :

$$((p \wedge \neg q) \rightarrow r) \wedge \neg r \wedge p \models q$$

Où "$\models$" se lit : "q est une conséquence logique de $((p \wedge \neg q) \rightarrow r) \wedge \neg r \wedge p$".

22 Conditions Booléennes dans les langages de programmation

Considérons le test suivant :

if $(x \geq 3$ **and** $z \leq 3)$ **or** $(y \leq 2$ **and** $z \leq 3)$ **then** ...

Il peut-être remplacé par

if $(x \geq 3$ **or** $y \leq 2)$ **and** $z \leq 3$ **then** ...

Car $(p \vee q) \wedge r$ est logiquement équivalent à $(p \wedge r) \vee (q \wedge r)$.

Logique Propositionnelle

Définitions de base : Syntaxe, Sémantique et Exemples

24 Construction des formules

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

- d'un ensemble, fini ou dénombrable, de *propositions* notées $x, y, z, \ldots$
  Dans la suite, nous notons les ensembles de propositions par les lettres X, Y,... ;
- de deux constantes : vrai (notée $\top$) et faux (notée $\bot$), parfois notée 1 et 0 ;
- d'un ensemble de *connecteurs logiques* : et (noté $\wedge$), ou (noté $\vee$), non (noté $\neg$), implique (noté $\rightarrow$), équivalent (noté $\leftrightarrow$) ;
- les parenthèses : $($,$)$.

25 Construction des formules

Soit X un ensemble de propositions. Les *formules de la logique propositionnelle* respectent la règle de formation `BNF` (Bachus-Naur Form) suivante :

$$\phi ::= \top \mid \bot \mid x \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \phi_1 \rightarrow \phi_2 \mid \phi_1 \leftrightarrow \phi_2 \mid \neg\phi_1 \mid (\phi_1)$$

Où $\top$, $\bot$ sont respectivement les constantes vrai et faux, $x \in X$ est une proposition et ϕ_1, ϕ_2 sont des formules propositionnelles bien formées.

La BNF se lit "une formule de la logique propositionnelle est soit la formule $\top$, soit la formule $\bot$, soit une proposition x, soit la conjonction de deux formules ϕ_1 et ϕ_2 (définition inductive), etc."

26 Quelques exemples de formules

Voici quelques exemples de formules et leur lecture intuitive :

- la formule propositionnelle "$x \rightarrow (y \wedge z)$", peut être lue de la façon suivante "x implique y et z", ou peut être également lue comme "si x est vrai alors y et z doivent être vrais" ;
- la formule proxositionnelle "$\neg(x \wedge y)$", peut-être lue de la façon suivante "il est faux que x et y soient vrais (en même temps)".

27 Ambiguïtés

L'utilisation de la *notation infixe* s'accompagne de problèmes de lecture :

Comment lire $x \wedge y \vee z$? $x \rightarrow y \rightarrow z$?

27 Ambiguïtés

L'utilisation de la *notation infixe* s'accompagne de problèmes de lecture :

Comment lire $x \wedge y \vee z$? $x \rightarrow y \rightarrow z$?

Pour lever les ambiguïtés, on utilise les parenthèses ou des règles de priorité entre opérateurs :

- si op_1 a une plus grande précédence (priorité) que op_2 alors
  $e_1\ op_1\ e_2\ op_2\ e_3$ est équivalent à $((e_1\ op_1\ e_2) op_2\ e_3)$
  - Par ex : $2 \times 3 + 5 = (2 \times 3) + 5 = 11 \neq 2 \times (3 + 5) = 16$.
- si op_2 a une plus grande précédence (priorité) que op_1 alors
  $e_1\ op_1\ e_2\ op_2\ e_3$ est équivalent à $(e_1 op_1\ (e_2 op_2\ e_3))$
  - Par ex : $2 + 3 \times 5 = 2 + (3 \times 5) = 17$.
- si op est associatif à gauche alors
  $e_1\ op\ e_2\ op\ e_3$ est équivalent à $(e_1\ op\ e_2) op\ e_3$
  - Par ex : $10/2/5 = (10/2)/5 = 1 \neq 10/(2/5) = 25$.
- associativité à droite : $e_1\ op\ e_2\ op\ e_3$ se lit $e_1 op (e_2 op\ e_3)$

On fixe des règles de priorité entre opérateurs afin d'avoir une lecture unique de la formule.

28 Règles de précédence en logique propositionnelle

Ordre de précédence $\prec$ sur les opérateurs :

$$\leftrightarrow \prec \rightarrow \prec \vee \prec \wedge \prec \neg$$

et associativité à gauche pour $\leftrightarrow, \vee, \wedge$ et à droite pour $\rightarrow$.

Exemples

$x \vee y \wedge z$	se lit	$x \vee (y \wedge z)$
$x \rightarrow y \rightarrow x$	se lit	$x \rightarrow (y \rightarrow z)$
$x \vee y \rightarrow z$	se lit	$(x \vee y) \rightarrow z$
$\neg x \wedge y$	se lit	$(\neg x) \wedge y$
$x \rightarrow y \wedge z \rightarrow t$	se lit	$x \rightarrow ((y \wedge z) \rightarrow t)$

28 Règles de précédence en logique propositionnelle

Ordre de précédence $\prec$ sur les opérateurs :

$$\leftrightarrow \prec \rightarrow \prec \vee \prec \wedge \prec \neg$$

et associativité à gauche pour $\leftrightarrow, \vee, \wedge$ et à droite pour $\rightarrow$.

Exemples

$x \vee y \wedge z$	se lit
$x \rightarrow y \rightarrow x$	se lit
$x \vee y \rightarrow z$	se lit
$\neg x \wedge y$	se lit
$x \rightarrow y \wedge z \rightarrow t$	se lit

Remarque

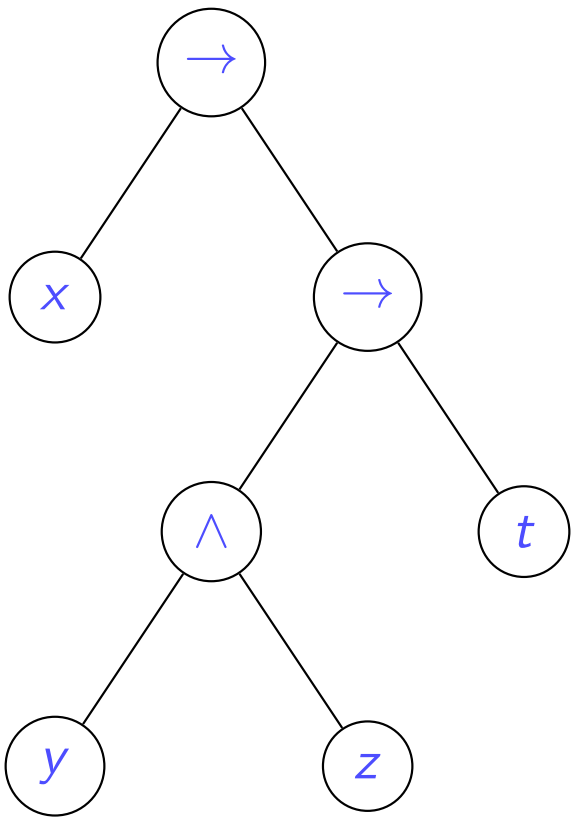
Les parenthèses permettent de contrecarrer ces règles, si elles ne conviennent pas. Elles permettent aussi de rendre une formule plus lisible, ou de ne pas devoir retenir les règles de précédence.

29 Arbre correspondant à une formule

La formule $x \rightarrow y \wedge z \rightarrow t$ est donc équivalente à

$$x \rightarrow ((y \wedge z) \rightarrow t)$$

et donc son arbre de lecture est :



30 La sémantique

- jusqu'ici, on a vu la syntaxe des formules, qui ne sont rien d'autre que des suites de caractères sans signification.
- la sémantique donne du sens aux formules, elle permet de les interpréter et de leur attribuer une *valeur de vérité* vrai ou faux.
- l'interprétation des symboles de proposition est cependant libre et c'est à nous de la fixer. On se donnera donc une une *fonction d'interprétation* V pour l'ensemble X de propositions considérées :

(ou valuation)

$$V : X \to \{0, 1\}$$

Cette fonction assigne à chaque proposition de X la valeur vrai ou la valeur faux.

- Dans la suite, nous utiliserons parfois le terme *valuation* , souvent utilisé dans la littérature, au lieu de fonction d'interprétation.
- A partir d'une fonction d'interprétation V des symboles de proposition, nous allons voir comment interpréter les formules.

31 La sémantique

Définition

- La *valeur de vérité* d'une formule propositionnelle ϕ formée à partir des propositions d'un ensemble X, évaluée avec la fonction d'interprétation V, est notée $[\![\phi]\!]_V$. En particulier, on a $[\![\phi]\!]_V \in \{0, 1\}$.
- La fonction $[\![\phi]\!]_V$ est définie par induction sur la syntaxe de ϕ de la façon suivante :
  - $[\![\top]\!]_V = 1$; $[\![\bot]\!]_V = 0$; $[\![x]\!]_V = V(x)$.
  - $[\![\neg\phi]\!]_V = 1 - [\![\phi]\!]_V$
  - $[\![\phi_1 \vee \phi_2]\!]_V = \max([\![\phi_1]\!]_V, [\![\phi_2]\!]_V)$
  - $[\![\phi_1 \wedge \phi_2]\!]_V = \min([\![\phi_1]\!]_V, [\![\phi_2]\!]_V)$
  - $[\![\phi_1 \to \phi_2]\!]_V = \max(1 - [\![\phi_1]\!]_V, [\![\phi_2]\!]_V) = \max([\![\neg\phi_1]\!]_V, [\![\phi_2]\!]_V)$
  - $[\![\phi_1 \leftrightarrow \phi_2]\!]_V = \min([\![\phi_1 \to \phi_2]\!]_V, [\![\phi_2 \to \phi_1]\!]_V)$. $= [\![\neg\phi_1 \vee \phi_2]\!]_V$

Nous notons $V \models \phi$ si et seulement si $[\![\phi]\!]_V = 1$, qui se lit "V satisfait ϕ".

32 La sémantique sous forme de tables de vérités

L'information contenue dans la définition précédente est souvent présentée sous forme de tables, appelées *tables de vérité* :

$\top$	$\bot$
1	0

ϕ	$\neg\phi$
1	0
0	1

33 La sémantique sous forme de tables de vérités

ϕ_1	ϕ_2	$\phi_1 \wedge \phi_2$	$\phi_1 \vee \phi_2$	$\phi_1 \rightarrow \phi_2$	$\phi_1 \leftrightarrow \phi_2$
1	1	1	1	1	1
1	0	0	1	0	0
0	1	0	1	1	0
0	0	0	0	1	1

34 Exemples

- Prenons l'interprétation $V_1(x) = 0$, alors on a :

$$[\![x]\!]_{V_1} = 0 \quad [\![\neg x]\!]_{V_1} = 1 \quad [\![x \wedge x]\!]_{V_1} = 0 \quad [\![x \wedge \neg x]\!]_{V_1} = 0$$
$$[\![x \rightarrow x]\!]_{V_1} = 1 \quad [\![x \leftrightarrow x]\!]_{V_1} = 1 \quad [\![\neg x \rightarrow x]\!]_{V_1} = 0$$

- Prenons la formule $\phi = (x \vee y) \wedge (\neg y \vee z)$ et l'interprétation $V_2(x) = V_2(z) = 1$ et $V_2(y) = 0$. Pour calculer la valeur de vérité de ϕ, on calcule d'abord les valeurs de vérité des sous-formules et on applique les tables de vérité.

$$\begin{gathered}(x \vee y) \wedge (\neg y \vee z) \\ (1 \vee 0) \wedge (\neg 0 \vee 1) \\ (1 \vee 0) \wedge (1 \vee 1) \\ 1 \wedge (1 \vee 1) \\ 1 \wedge 1 \\ 1\end{gathered}$$

Donc $[\![\phi]\!]_{V_2} = 1$.

35 Exemples

- Prenons la même formule $\phi = (x \vee y) \wedge (\neg y \vee z)$ et l'interprétation $V_3(z) = 1$ et $V_3(x) = V_3(y) = 0$.

$$
\begin{gathered}
(x \vee y) \wedge (\neg y \vee z) \\
(0 \vee 0) \wedge (\neg 0 \vee 1) \\
(0 \vee 0) \wedge (1 \vee 1) \\
0 \wedge (1 \vee 1) \\
0 \wedge 1 \\
0
\end{gathered}
$$

Donc $[\![\phi]\!]_{V_3} = 0$.

36 Exemples

On peut utiliser le principe de décomposition en sous-formule et l'application des tables de vérité pour obtenir la valeur de vérité de ϕ pour toutes les interprétations possibles des variables

x	y	z	$(x \vee y)$	$\neg y$	$(\neg y \vee z)$	$(x \vee y) \wedge (\neg y \vee z)$
1	1	1				
1	1	0				
1	0	1				
1	0	0				
0	1	1				
0	1	0				
0	0	1				
0	0	0				

36 Exemples

On peut utiliser le principe de décomposition en sous-formule et l'application des tables de vérité pour obtenir la valeur de vérité de ϕ pour toutes les interprétations possibles des variables

x	y	z	$(x \vee y)$	$\neg y$	$(\neg y \vee z)$	$(x \vee y) \wedge (\neg y \vee z)$
1	1	1	1			
1	1	0	1			
1	0	1	1			
1	0	0	1			
0	1	1	1			
0	1	0	1			
0	0	1	0			
0	0	0	0			

36 Exemples

On peut utiliser le principe de décomposition en sous-formule et l'application des tables de vérité pour obtenir la valeur de vérité de ϕ pour toutes les interprétations possibles des variables

x	y	z	$(x \vee y)$	$\neg y$	$(\neg y \vee z)$	$(x \vee y) \wedge (\neg y \vee z)$
1	1	1	1	0		
1	1	0	1	0		
1	0	1	1	1		
1	0	0	1	1		
0	1	1	1	0		
0	1	0	1	0		
0	0	1	0	1		
0	0	0	0	1		

36 Exemples

On peut utiliser le principe de décomposition en sous-formule et l'application des tables de vérité pour obtenir la valeur de vérité de ϕ pour toutes les interprétations possibles des variables

x	y	z	$(x \vee y)$	$\neg y$	$(\neg y \vee z)$	$(x \vee y) \wedge (\neg y \vee z)$
1	1	1	1	0	1	
1	1	0	1	0	0	
1	0	1	1	1	1	
1	0	0	1	1	1	
0	1	1	1	0	1	
0	1	0	1	0	0	
0	0	1	0	1	1	
0	0	0	0	1	1	

36 Exemples

On peut utiliser le principe de décomposition en sous-formule et l'application des tables de vérité pour obtenir la valeur de vérité de ϕ pour toutes les interprétations possibles des variables

x	y	z	$(x \vee y)$	$\neg y$	$(\neg y \vee z)$	$(x \vee y) \wedge (\neg y \vee z)$
1	1	1	1	0	1	1
1	1	0	1	0	0	0
1	0	1	1	1	1	1
1	0	0	1	1	1	1
0	1	1	1	0	1	1
0	1	0	1	0	0	0
0	0	1	0	1	1	0
0	0	0	0	1	1	0

37 Exemples

- $\neg x \vee y$ (qui se lit $(\neg x) \vee y$)

x	y	$\neg x$	$\neg x \vee y$
1	1	0	1
1	0	0	0
0	1	1	1
0	0	1	1

- Remarque : c'est la même table de vérité que pour l'implication :

x	y	$x \rightarrow y$
1	1	1
1	0	0
0	1	1
0	0	1

- On dira que les deux formules $\neg x \vee y$ et $x \rightarrow y$ sont **équivalentes**.

38 Remarques sur l'implication

- l'implication $\phi_1 \rightarrow \phi_2$ modélise le raisonnement *si-alors* :

  Si ϕ_1 est vraie, **ALORS** ϕ_2 est vraie aussi.
- le cas où ϕ_1 est faux ne nous intéresse pas dans l'implication.
- par conséquent, si ϕ_1 est faux, alors peu importe la valeur de vérité de ϕ_2, l'implication $\phi_1 \rightarrow \phi_2$ reste vraie.

39 Un dernier exemple : $\phi = \neg(x \wedge y) \leftrightarrow (\neg x \vee \neg y)$

- Calculons la table de vérité :

x	y	$x \wedge y$	$\neg(x \wedge y)$	$\neg x$	$\neg y$	$\neg x \vee \neg y$	ϕ
1	1	1	0	0	0	0	1
1	0	0	1	0	1	1	1
0	1	0	1	1	0	1	1
0	0	0	1	1	1	1	1

- pour toute les interprétations possibles des propositions, ϕ est vraie. On dira dans ce cas que ϕ est **valide**.
- En fait, on a montré ici que les deux formule $\neg(x \wedge y)$ et $\neg x \vee \neg y$ sont équivalentes. C'est une des lois de Morgan.

40 Exercices

Dire pour les interprétations V et les formules ϕ suivantes si $V \models \phi$:

1. $V(x) = 1$, $V(y) = 0$ et $\phi = (x \to y) \wedge (x \to \neg y)$
2. $V(x) = V(y) = 0$ et $\phi = ((\neg x \to y) \to (\neg y \to x)) \wedge (x \vee y)$
3. $V(x) = 0$ et $V(y) = 1$ et $\phi = ((\neg x \vee y) \to (y \wedge (x \leftrightarrow y))$
4. $V(x) = V(z) = 1$ et $V(y) = 0$ et
   $\phi = ((\neg z \to \neg x \wedge \neg y) \vee t) \leftrightarrow (x \vee y \to z \vee t)$
5. $V(x) = V(y) = 0$ et $V(z) = 1$ et
   $\phi = (x \wedge (y \to z)) \leftrightarrow ((\neg x \vee y) \to (x \wedge z))$.

41 Validité et Satisfaisabilité

Voici deux définitions importantes :

Définition (Satisfaisabilité)

Une formule propositionnelle ϕ est *satisfaisable* si et seulement si il existe une fonction d'interprétation V pour les propositions de ϕ, telle que $V \models \phi$.

Définition (Validité)

Une formule propositionnelle ϕ est *valide* si et seulement si pour toute fonction d'interprétation V pour les propositions de ϕ, on a $V \models \phi$.

42 Exemples

formule	satisfaisable	valide
$\top$	oui	oui
x	oui $V(x) = 1$	non $V(x) = 0$
$\neg x$	oui $V(x) = 0$	non $V(x) = 1$
$(x \vee y) \wedge (\neg y \vee z)$	oui voir table précédente	non voir table précédente
$x \wedge \neg x$	non	non
$x \wedge \neg y \wedge (\neg y \rightarrow \neg x)$	non	non
$x \vee \neg x$	oui	oui
$(x \rightarrow y) \vee (y \rightarrow x)$	oui	oui

43 Notion de conséquence logique

Définition

Soit $\phi_1, \ldots, \phi_n, \phi$ des formules. On dira que ϕ est une *conséquence logique* de $\phi_1, \ldots, \phi_n$, noté $\phi_1, \ldots, \phi_n \models \phi$, si $(\phi_1 \wedge \cdots \wedge \phi_n) \rightarrow \phi$ est valide.

Par exemple, $p, \neg p \models \bot$, et $\phi_1, \phi_1 \rightarrow \phi_2 \models \phi_2$.

44 Application importante de la notion de validité

Définition

Deux formules ϕ et ψ sont dites *équivalentes* si la formule $\phi \leftrightarrow \psi$ est valide. On notera $\phi \equiv \psi$ pour signifier que ϕ et ψ sont équivalentes.

- dans une formule, on peut substituer une sous-formule par une autre équivalente sans changer la sémantique de la formule de départ
- par exemple, dans la formule

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

on peut remplacer la sous-formule $\neg(x \wedge z)$ par $(\neg x \vee \neg z)$ et on obtient la formule suivante, qui est équivalente à γ :

$$\gamma' = (x \vee y) \wedge (\neg x \vee \neg z)$$

- l'application d'équivalences simples va nous permettre de simplifier des formules, et de les mettre sous des formes particulières

45 Quelques équivalences

Pour toutes formules ϕ_1, ϕ_2, ϕ_3, on a :

- $\neg\neg\phi_1 \equiv \phi_1$ (double négation)
- $\neg(\phi_1 \wedge \phi_2) \equiv (\neg\phi_1 \vee \neg\phi_2)$ (loi de De Morgan pour le 'et')
- $\neg(\phi_1 \vee \phi_2) \equiv (\neg\phi_1 \wedge \neg\phi_2)$ (loi de De Morgan pour le 'ou')
- $\phi_1 \wedge (\phi_2 \vee \phi_3) \equiv (\phi_1 \wedge \phi_2) \vee (\phi_1 \wedge \phi_3)$ (distributivité du 'et')
- $\phi_1 \vee (\phi_2 \wedge \phi_3) \equiv (\phi_1 \vee \phi_2) \wedge (\phi_1 \vee \phi_3$ (distributivité du 'ou')
- $\phi_1 \rightarrow \phi_2 \equiv \neg\phi_2 \rightarrow \neg\phi_1$ (contraposition)

46 Comment démontrer ces équivalences ? (1)

Prenons par exemple l'équivalence suivante :

$$\phi_1 \vee (\phi_2 \wedge \phi_3) \equiv (\phi_1 \vee \phi_2) \wedge (\phi_1 \vee \phi_3)$$

On doit démontrer que la formule $(\phi_1 \vee (\phi_2 \wedge \phi_3)) \leftrightarrow ((\phi_1 \vee \phi_2) \wedge (\phi_1 \vee \phi_3)$ est valide, c'est-à-dire que quelque soit la valeur de vérité des formules ϕ_1, ϕ_2, ϕ_3, les deux formules à droite et à gauche de l'implication ont la même sémantique. Pour cela, on pourrait faire une table de vérité :

ϕ_1	ϕ_2	ϕ_3	$\phi_2 \wedge \phi_3$	$\phi_1 \vee \phi_2$	$\phi_1 \vee \phi_3$	$\phi_1 \vee (\phi_2 \wedge \phi_3)$	$(\phi_1 \vee \phi_2) \wedge (\phi_1 \vee \phi_3)$
1	1	1	1	1	1	1	1
1	1	0	0	1	1	1	1
1	0	1	0	1	1	1	1
1	0	0	0	1	1	1	1
0	1	1	1	1	1	1	1
0	1	0	0	1	0	0	0
0	0	1	0	0	1	0	0
0	0	0	0	0	0	0	0

47 Comment démontrer ces équivalences ? (2)

On peut aller plus vite que faire la table de vérité. Nous devons démontrer que pour toute interprétation V, nous avons que $V \models \phi_1 \vee (\phi_2 \wedge \phi_3)$ si et seulement si $V \models (\phi_1 \vee \phi_2) \wedge (\phi_1 \vee \phi_3)$. Nous considérons deux cas :

1. si $V \models \phi_1$, alors on obtient par définition de la disjonction : $V \models \phi_1 \vee (\phi_2 \wedge \phi_3)$, $V \models (\phi_1 \vee \phi_2)$, et $V \models (\phi_1 \vee \phi_3)$. Donc $V \models (\phi_1 \vee \phi_2) \wedge (\phi_1 \vee \phi_3)$.
2. si $V \not\models \phi_1$. Alors $V \models \phi_1 \vee (\phi_2 \wedge \phi_3)$ ssi $V \models \phi_2 \wedge \phi_3$ ssi $V \models \phi_1$ et $V \models \phi_2$, ssi $V \models \phi_1 \vee \phi_2$ et $V \models \phi_1 \vee \phi_3$, ssi $V \models (\phi_1 \vee \phi_2) \wedge (\phi_1 \vee \phi_3)$.

48 Négation de l'implication

Quand est-ce que la négation d'une implication $A \to B$ est vraie ? Dès lors que l'hypothèse (A) est vraie, mais pas la conclusion (B). Autrement dit, lorsqu'on a $A \wedge \neg B$.
On peut le démontrer formellement avec les règles d'équivalences :

$$\neg(A \to B) \equiv \neg(\neg A \vee B) \equiv \neg\neg A \wedge \neg B \equiv A \wedge \neg B$$

49 Lien entre satisfaisibilité et validité

Théorème

Une formule propositionnelle ϕ est valide ssi sa négation $\neg\phi$ n'est pas satisfaisable.

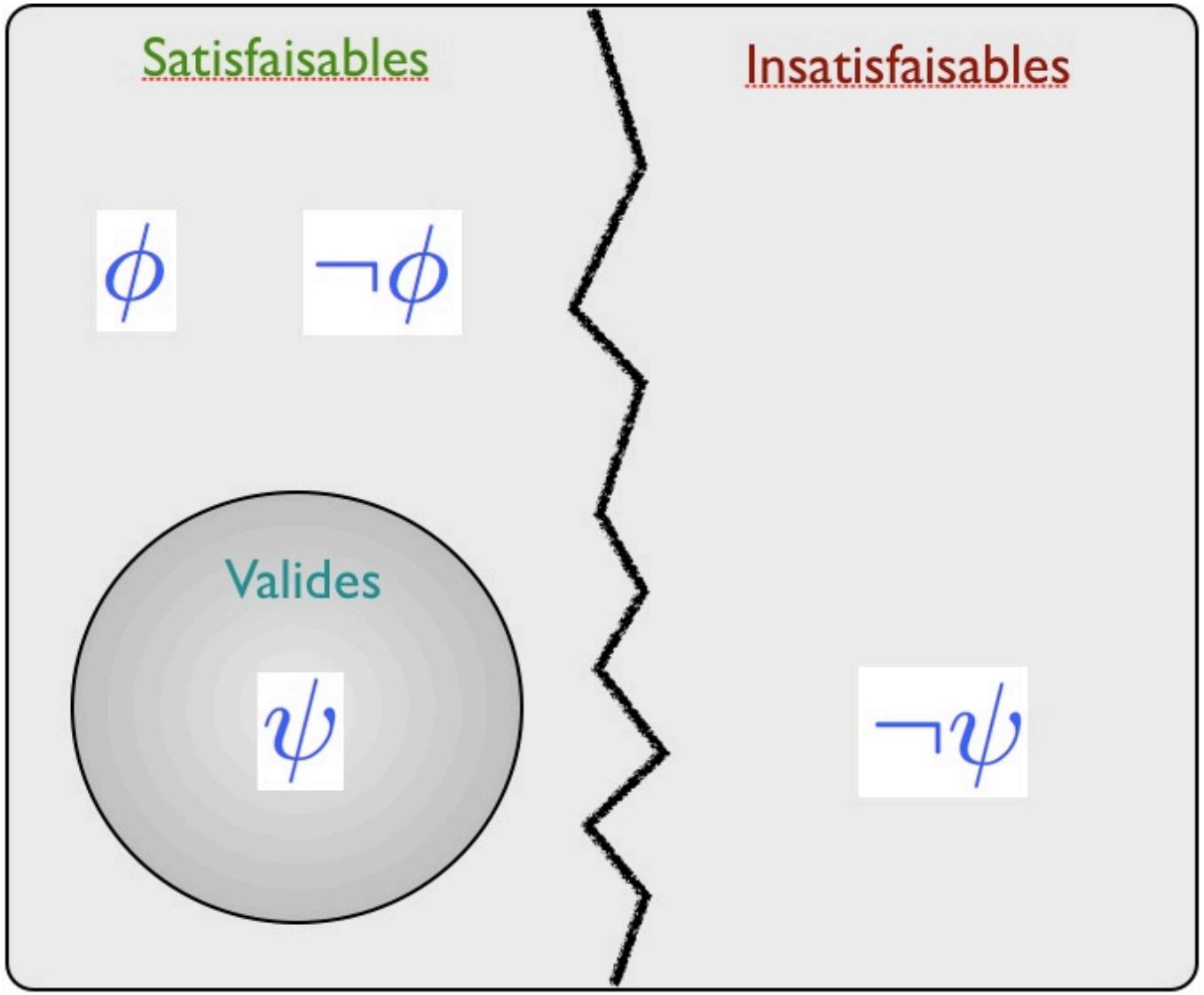
Par conséquent, si on dispose d'un algorithme qui décide si une formule est satisfaisable ou non, on obtient un algorithme qui décide si la formule est valide, il suffit de tester la (non-)satisfaisabilité de sa négation.
Remarque : ce théorème permet de réduire le problème de validité (décider si oui ou non une formule est valide) en un problème d'insatisfaisabilité (décider si oui ou non une formule est insatisfaisable). On peut d'ailleurs concevoir un algorithme qui fait cette réduction : il doit simplement prendre la négation de la formule.

50 Démonstration du théorème

Nous devons démontrer une équivalence (un si et seulement si). On peut donc supposer ce qui est à gauche de l'équivalence et en déduire ce qui est à droite, et réciproquement.

1. supposons que ϕ est valide. Alors pour tout interprétation V, on a $V \models \phi$. Donc $V \not\models \neg\phi$. Comme c'est pour toute interprétation quelconque V, cela signifie que $\neg\phi$ n'est pas satisfaisable.
2. réciproquement, supposons que $\neg\phi$ ne soit pas satisfaisable. Alors pour tout interprétation V quelconque, $V \not\models \neg\phi$, donc $V \models \phi$. De nouveau, comme c'est pour tout interprétation quelconque V, on en déduit que ϕ est valide.

51 Diagramme



52 Comment tester la satisfaisabilité d'une formule ?

52 Comment tester la satisfaisabilité d'une formule ?

- on a vu la méthode des tables de vérité : tester toutes les interprétations de propositions jusqu'à ce qu'on en trouve une qui satisfait la formule
- **problème :** quelle est la complexité d'un tel algorithme ? combien d'interprétations faut-il tester si la formule contient n propositions ?

52 Comment tester la satisfaisabilité d'une formule ?

- on a vu la méthode des tables de vérité : tester toutes les interprétations de propositions jusqu'à ce qu'on en trouve une qui satisfait la formule
- **problème :** quelle est la complexité d'un tel algorithme ? combien d'interprétations faut-il tester si la formule contient n propositions ?
- il faut essayer, dans le pire des cas (c'est à dire le cas où la formule n'est pas satisfaisable), 2^n interprétations : cet algorithme a donc une complexité exponentielle dans le nombre de propositions
- ce n'est pas raisonnable pour les applications que nous allons aborder, car nous allons générer des formules contenant plusieurs centaines de propositions.
- nous allons maintenant étudier un algorithme "plus intelligent" : la méthode des tableaux sémantiques.

53 Tableaux sémantiques

C'est un algorithme pour établir la satisfaisabilité de formules de la logique propositionnelle.
On a besoin d'une nouvelle définition :

Définition

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

54 Tableaux sémantiques : exemple

Considérons la formule $\phi = x \wedge (\neg y \vee \neg x)$
Essayons de construire systématiquement une fonction d'interprétation V telle que

$$[\![\phi]\!]_V = 1$$

Par définition on a

$$[\![\phi]\!]_V = 1$$
$$\underline{\text{ssi}}$$
$$[\![x]\!]_V = 1 \quad \underline{\text{et}} \quad [\![\neg y \vee \neg x]\!]_V = 1$$

et

$$[\![\neg y \vee \neg x]\!]v = 1$$
$$\underline{\text{ssi}}$$
$$[\![\neg y]\!]v = 1 \ \ \underline{\text{ou}} \ \ [\![\neg x]\!]v = 1$$

et donc,

$$[\![\phi]\!]v = 1$$
$$\underline{\text{ssi}}$$
$$[\![x]\!]v = 1 \ \underline{\text{et}} \ [\![\neg y]\!]v = 1$$
$$\underline{\text{ou}} \ [\![x]\!]v = 1 \ \underline{\text{et}} \ [\![\neg x]\!]v = 1$$

Pour ϕ, on construit la fonction d'interprétation V de la façon suivante :

- $V(x) = 1$;
- $V(y) = 0$.

et on a bien que $V \models \phi$.

56 Tableaux sémantiques

- Un ensemble S de littéraux est satisfaisable ssi il ne contient pas une paire de *littéraux complémentaires* . Par exemple, $\{x, \neg y\}$ est satisfaisable alors que $\{x, \neg x\}$ ne l'est pas.
- Nous avons réduit le problème de satisfaction de ϕ à un problème de satisfaction d'ensembles de littéraux : ϕ est satisfaisable ssi $\{x, \neg y\}$ est satisfaisable ou $\{x, \neg x\}$ est satisfaisable.

57 Tableaux sémantiques

Un autre exemple : $\phi = (x \vee y) \wedge (\neg x \wedge \neg y)$.

Par définition, on a

$$[\![\phi]\!]v = 1$$
$$\underline{\text{ssi}}$$
$$[\![x \vee y]\!]v = 1 \ \underline{\text{et}} \ \ [\![\neg x \wedge \neg y]\!]v = 1$$

et donc,

$$[\![\phi]\!]v = 1$$
$$\underline{\text{ssi}}$$
$$[\![x \vee y]\!]v = 1 \ \underline{\text{et}} \ \ [\![\neg x]\!]v = 1 \ \underline{\text{et}} \ \ [\![\neg y]\!]v = 1$$

et donc,

$$[\![\phi]\!]v = 1$$
$$\underline{\text{ssi}}$$
$$\text{soit } [\![x]\!]v = 1 \ \underline{\text{et}} \ \ [\![\neg x]\!]v = 1 \ \underline{\text{et}} \ \ [\![\neg y]\!]v = 1$$
$$\underline{\text{ou}} \ [\![y]\!]v = 1 \ \underline{\text{et}} \ \ [\![\neg x]\!]v = 1 \ \underline{\text{et}} \ \ [\![\neg y]\!]v = 1$$

58 Tableaux sémantiques

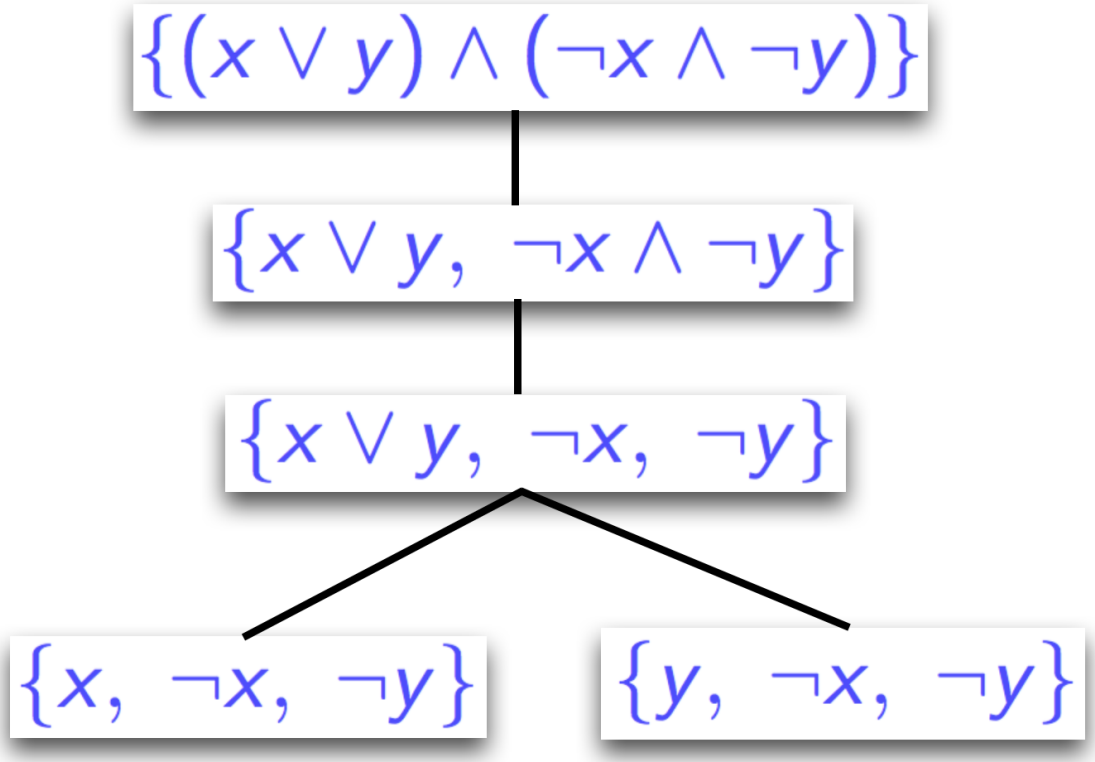
et donc,

$$\phi \text{ est satisfaisable}$$
$$\underline{\text{ssi}}$$
$$\{x, \neg x, \neg y\} \text{ est satisfaisable}$$
$$\underline{\text{ou}}\{y, \neg x, \neg y\} \text{ est satisfaisable.}$$

Vu que les deux ensembles de littéraux contiennent chacun une proposition et sa négation, la formule ϕ n'est pas satisfaisable.

59 Tableaux sémantiques

L'information présente dans les développements précédents est plus facilement représentée sous forme d'un arbre :



60 Tableaux sémantiques – Remarques

- quand on applique le "pas de simplification" à un noeud où on sélectionne une conjonction, alors on obtient un seul fils ; on appelle ces règles, des $\wedge$-règles ;
- quand on applique le "pas de simplification" à un noeud où on sélectionnne une disjonction, alors on obtient deux fils ; on appelle ces règles, des $\vee$-règles.

61 Règles de simplification : $\wedge$-règles

Etant donné un ensemble S contenant une formule α, on crée un ensemble fils égal $S \backslash \{\alpha\}$ auquel on ajoute α_1 et α_2.

α	α_1	α_2
$\neg\neg\phi$	ϕ	
$\phi_1 \wedge \phi_2$	ϕ_1	ϕ_2
$\neg(\phi_1 \vee \phi_2)$	$\neg\phi_1$	$\neg\phi_2$
$\neg(\phi_1 \rightarrow \phi_2)$	ϕ_1	$\neg\phi_2$
$\phi_1 \leftrightarrow \phi_2$	$\phi_1 \rightarrow \phi_2$	$\phi_2 \rightarrow \phi_1$

$\wedge$-règles .

Remarque : toutes les formules α peuvent être considérées comme équivalentes à des conjonctions. Par exemple :

$\neg(\phi_1 \vee \phi_2)$ est équivalent à $\neg\phi_1 \wedge \neg\phi_2$

62 Règles de simplification : $\vee$-règles

Etant donné un ensemble S contenant une formule β, on crée deux ensembles fils, l'un étant $(S \cup \{\beta_1\})\backslash\beta$, et l'autre $(S \cup \{\beta_2\})\backslash\beta$.

β	β_1	β_2
$\phi_1 \vee \phi_2$	ϕ_1	ϕ_2
$\neg(\phi_1 \wedge \phi_2)$	$\neg\phi_1$	$\neg\phi_2$
$\phi_1 \rightarrow \phi_2$	$\neg\phi_1$	ϕ_2
$\neg(\phi_1 \leftrightarrow \phi_2)$	$\neg(\phi_1 \rightarrow \phi_2)$	$\neg(\phi_2 \rightarrow \phi_1)$

$\vee$-règles .

Remarque : toutes les formules β peuvent être considérées comme équivalentes à des disjonctions. Par exemple :

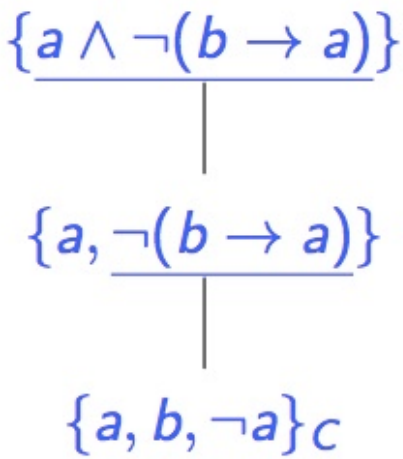
$\neg(\phi_1 \wedge \phi_2)$ est équivalent à $\neg\phi_1 \vee \neg\phi_2$

63 Algorithme

L'algorithme construit à partir d'une formule ϕ un arbre noté T_ϕ dont les noeuds sont des ensembles de formules.

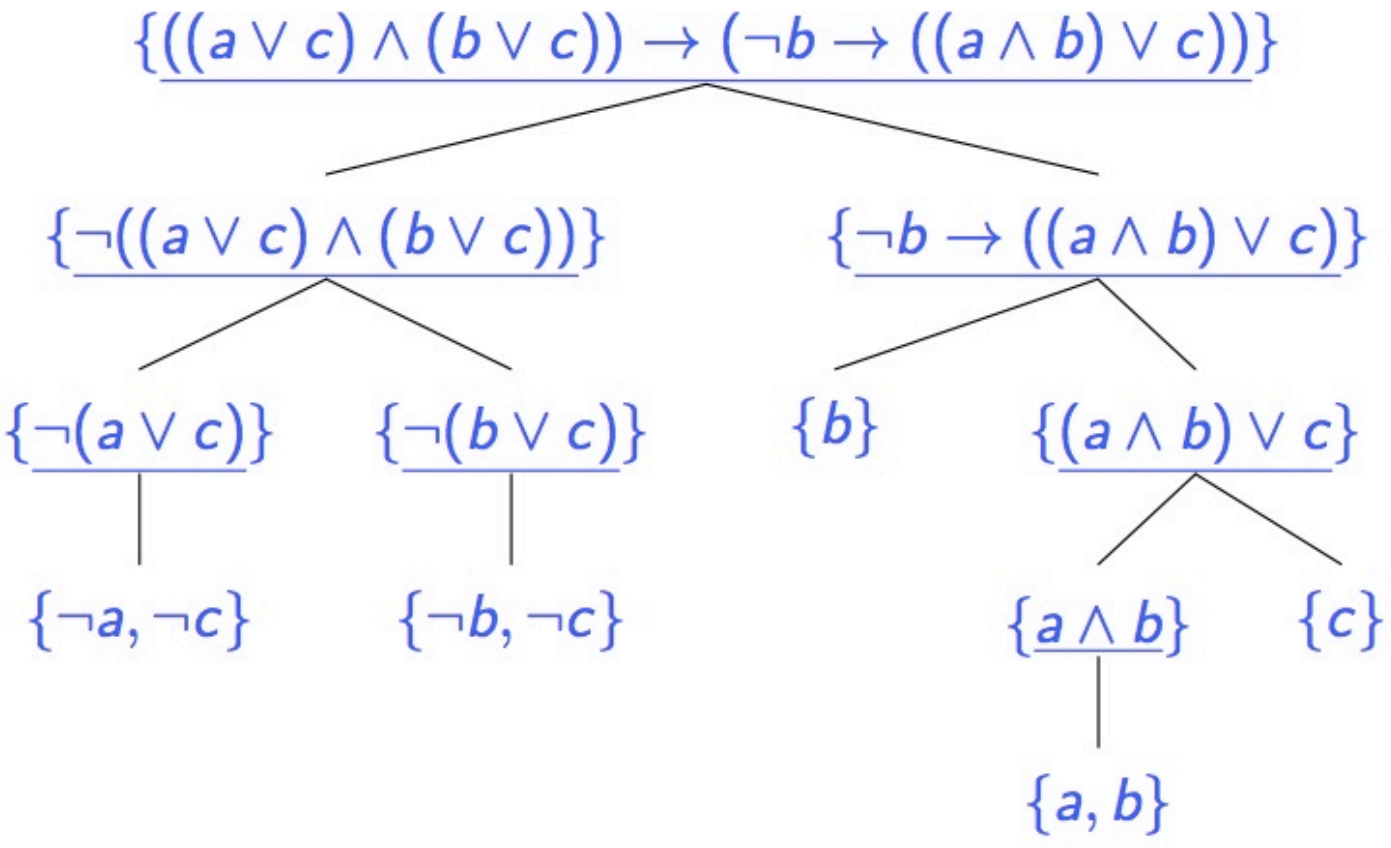
- **Initialisation** : au départ, l'arbre ne contient qu'un seul noeud : l'ensemble $\{\phi\}$;
- **Tant qu**'il existe une feuille de l'arbre F qui contient une formule ψ qui est simplifiable
  - **si** ϕ est simplifiable par une $\wedge$-règle, alors on crée un fils F' à F où F' est obtenu supprimant ϕ de F et en ajoutant la formule (dans le cas d'une double négation) ou les deux formules (pour les autres cas) obtenues par la simplification, à l'ensemble F'
  - **si** ϕ est simplifiable par une $\vee$-règle, alors on crée deux fils F_1 et F_2, chacun étant obtenu en supprimant ϕ de F et en ajoutant respectivement les formules obtenues par la simplification.
- **si** toutes les feuilles de l'arbre contiennent une paire de littéraux complémentaires, **alors retourner** non-satisfaisable, **sinon retourner** satisfaisable.

64 Exemples



Toutes les feuilles (ici une seule) contiennent une paire de littéraux complémentaires (a et $\neg a$), donc la formule n'est pas satisfaisable.

65 Exemples



Il existe plusieurs feuilles ne contenant pas de paire de littéraux complémentaires, la formule est donc satisfaisable.

66 Questions

Soit ϕ une formule et T_ϕ un arbre construit avec l'algorithme précédent.

1. Si toutes les feuilles de T_ϕ sont satisfaisables, est-ce que cela signifie que ϕ est valide ?

66 Questions

Soit ϕ une formule et T_ϕ un arbre construit avec l'algorithme précédent.

1. Si toutes les feuilles de T_ϕ sont satisfaisables, est-ce que cela signifie que ϕ est valide ? Non : prendre par exemple $\phi = x$.
2. quelle est la taille maximale de T_ϕ (nombre de noeuds) en fonction du nombre de symboles de ϕ ?

66 Questions

Soit ϕ une formule et T_ϕ un arbre construit avec l'algorithme précédent.

1. Si toutes les feuilles de T_ϕ sont satisfaisables, est-ce que cela signifie que ϕ est valide ? Non : prendre par exemple $\phi = x$.
2. quelle est la taille maximale de T_ϕ (nombre de noeuds) en fonction du nombre de symboles de ϕ ? T_ϕ peut avoir un nombre exponentiel de noeuds. Par exemple, prenons $n \geq 0$ et $\phi = (x_1 \vee y_1) \wedge (x_2 \vee y_2) \cdots \wedge (x_n \vee y_n)$, alors T_ϕ a au moins $2^{n+1} + n - 2$ noeuds, et exactement 2^n feuilles. L'algorithme a donc une complexité exponentielle en temps dans le pire cas.

67 Tableaux Sémantiques : Résumé

- la méthode des tableaux sémantiques est un algorithme pour tester la satisfaisabilité d'une formule.
- elle est basée sur la construction d'un arbre par applications successives de règles de simplifications de formules (tableaux)
- les formules équivalentes à une disjonction (disjonction, implication, négation d'une conjonction, négation d'une équivalence) sont satisfaites si un des deux membres de la disjonction est satisfait : les deux choix sont représentés par du branchement (deux fils) dans l'arbre.
- les formules équivalentes à une conjonction (conjonction, équivalence, négation d'une implication et d'une disjonction) sont satisfaites si les deux membres de la conjonction sont satisfaits : un seul fils est créer dans l'arbre.
- les feuilles de l'arbres sont des ensembles de littéraux (donc non simplifiables), pour lesquels la satisfaisabilité est facile à tester (absence de littéraux complémentaires).
- dans le pire cas, l'arbre créé est exponentiellement plus grand que la formule
- pour tester la validité d'une formule, on peut tester la non satisfaisabilité de se négation par la méthode des tableaux sémantiques.