

Modèles de calcul

Année 2011–2012¹

M1, Univ. Bordeaux
<http://www.labri.fr/perso/anca/MC.html>

18 novembre 2011

1. Version de Anca Muscholl, basée sur le poly de Marc Zeitoun.

1/131

Modalités du cours

- ▶ 12 cours, 4 (5 ?) groupes de TD (débutent la semaine du 12/9).
- ▶ Y. Oualhadj, M. Sylvestre, A. Zvonkine.
- ▶ Contrôle continu (CC) obligatoire (sauf dispense). Un DM et un DS.
- ▶ Note finale session 1 :
 $2/3$ Examen (3h) + $1/3$ CC.
- ▶ Note finale session 2 :
 $\max(\text{Examen}, 2/3 \text{ Examen (3h)} + 1/3 \text{ CC})$.

2/131






Objectifs (fiche UE)

Définir, **indépendamment de la technologie** :

- ▶ ce qui est **calculable** et ce qui ne l'est pas (théorie de la calculabilité) ;
- ▶ ce qui est calculable **efficacement** et ce qui ne l'est pas (théorie de la complexité).



3/131

Bibliographie

-  J.E. Hopcroft, R. Motwani, J. D. Ullman.
Introduction to Automata Theory, Languages & Computation.
Addison-Wesley, 2005.
-  M. Sipser.
Introduction to the Theory of Computation.
PWS publishing Company, 1997.
-  D. Kozen.
Automata and Computability.
Springer Verlag, 1997.
-  O. Carton.
Langages formels, Calculabilité et Complexité.
Vuibert, 2008.
-  J.M. Autebert.
Calculabilité et Décidabilité.
Masson, 1992.

4/131

Bibliographie complémentaire

-  Ch. Papadimitriou.
Computational complexity.
Addison-Wesley, 1995.
-  M. Garey, D. Johnson.
Computers and intractability.
W.H. Freeman & Co, 1979.
-  J. E. Savage.
Models of computation.
Addison-Wesley, 1998.

5/131

Plan

Présentation, bref historique

Ensembles dénombrables (rappels)

Modèles de calcul

Problèmes indécidables

Problèmes et réductions

La classe NP

Machines de Turing, hiérarchie de Chomsky

7/131

Plan du cours

Présentation, bref historique

Ensembles dénombrables (rappels)

Modèles de calcul

Problèmes indécidables

Problèmes et réductions

La classe NP

Machines de Turing, hiérarchie de Chomsky

6/131

Questions abordées dans ce cours

- ▶ Quels problèmes peut-on résoudre avec une machine (indépendamment de puissance de calcul des machines) ?
- ▶ Comment formaliser
 - ▶ ce qu'est un problème ?
 - ▶ ce qu'est une machine ?
- ▶ Quelles fonctions peut-on calculer avec une machine ?
- ▶ Comment comparer la complexité des problèmes ?
Y a-t-il des problèmes inhéremment difficiles ?

8/131

Bref historique (1)

- 1900 Hilbert, 10^{ème} problème : peut-on décider, de façon mécanique, si une équation Diophantienne a une solution ?
- 1931 Gödel publie le théorème d'incomplétude.
- 1935 Turing formalise une définition de machine et de calcul.
- 1936 Church exhibe un problème non résoluble par machine.
- 1938 Kleene prouve l'équivalence entre machines de Turing, λ -calcul, fonctions récursives.
- 1947 Post & Markov prouvent qu'un problème posé par Thue en 1914 n'est pas résoluble mécaniquement.
- 1970 Matiyasevich répond négativement au 10^{ème} problème de Hilbert.
- 1971 Cook & Levin formalisent la notion de problème NP-complet.

9/131

Bref historique (2)

- 1936 la calculabilité émerge de la logique (Russel, Hilbert, Turing, Church sont logiciens)
- 1941 Zuse invente le premier ordinateur (Z3) ; suit ENIAC 1945 ; modèle de von Neumann
- 1950 premiers langages de programmation (Fortran, Cobol, Lisp)
- 1960 développement de la théorie des langages formels et des modèles de calcul (hiérarchie de Chomsky)
- 1970 développement de la théorie de la complexité, question P vs. NP.
- 70/80 algorithmique et structures de données
- 80/90 algorithmique parallèle, distribuée ; cryptographie

10/131

Pourquoi ce cours ?

- ▶ pour comprendre les limites de la programmation, mais aussi...
- ▶ ... pour l'esthétique : la théorie apporte un point de vue plus simple et plus élégant,
- ▶ ... pour la pérennité : la technologie évolue rapidement, mais les concepts de base (la théorie) restent valides,
- ▶ ... pour développer l'abilité d'analyser et/ou décrire un problème avec clarté.

11/131

Formalismes

- ▶ On utilisera soit un style informel et des langages de haut niveau, voir des pseudo-langages pour décrire des algorithmes...
- ▶ ... ou on utilisera des formalismes plus précis, comme les machines de Turing, que l'on peut comparer avec les langages d'assembleur - syntaxe simple, mais plus lourds à utiliser.

Pourquoi le style informel ne suffit-il pas ? Comme vous le savez déjà, c'est plus difficile d'analyser par exemple le temps de calcul d'un algorithme décrit en pseudo-langage que celui d'un programme C ou Java. Plus généralement, pour prouver des choses, il est nécessaire d'adopter des descriptions plus précises, et donc plus formelles.

12/131

Qu'est-ce que c'est "calculable" ?

- ▶ Calcul = programme, algorithme, ...
- ▶ Une fonction $f : \mathbb{N} \rightarrow \mathbb{N}$ est **calculable** s'il existe un programme/algorithme qui calcule f .

Cette pseudo-définition est très peu satisfaisante : est-ce qu'il existe des fonctions "C-calculables", mais pas "Java-calculables" ? est-ce qu'on doit faire une preuve de calculabilité à chaque fois qu'on change de langage ? ou d'ordinateur ?

- ▶ Heureusement non : en calculabilité on admet la thèse de **Church-Turing**, qui dit que tous les modèles "raisonnables" de calcul sont équivalents.

13/131

Exemples de problèmes (de décision)

- Problème 1** **Donnée** Un nombre entier positif n en base 2.
Question n est-il pair ?
- Problème 2** **D.** Un nombre entier positif n en base 10.
Q. n est-il premier ?
- Problème 3** **D.** Une séquence DNA s et un motif p .
Q. p apparaît-il dans s ?
- Problème 4** **D.** Un programme C.
Q. Le programme est-il syntaxiquement correct ?

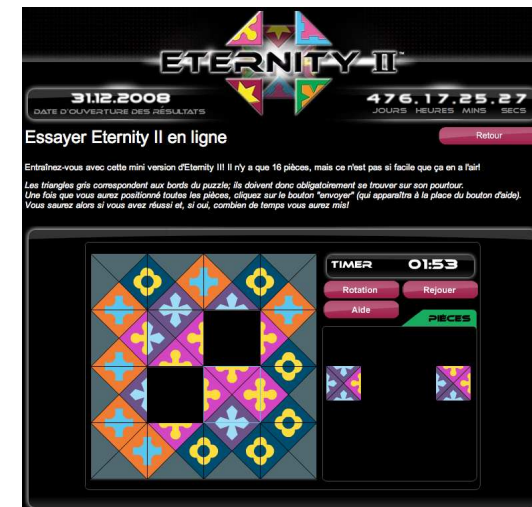
14/131

Exemples de problèmes de décision (2)

- Problème 5** **Donnée** Un graphe donné par une liste d'adjacence.
Question Le graphe est-il 3-coloriable ?
- Problème 6** **D.** Un puzzle Eternity <http://fr.eternityii.com>.
Q. Le puzzle a-t-il une solution ?
- Problème 7** **D.** Un programme C.
Q. Le programme s'arrête-t-il sur au moins l'une de ses entrées ?
- Problème 8** **D.** Un programme C.
Q. Le programme s'arrête-t-il sur toute entrée ?
- Problème 9** **D.** Des couples de mots $(u_1, v_1), \dots, (u_n, v_n)$.
Q. Existe-t-il des entiers i_1, \dots, i_k tels que $u_{i_1} \dots u_{i_k} = v_{i_1} \dots v_{i_k}$?

15/131

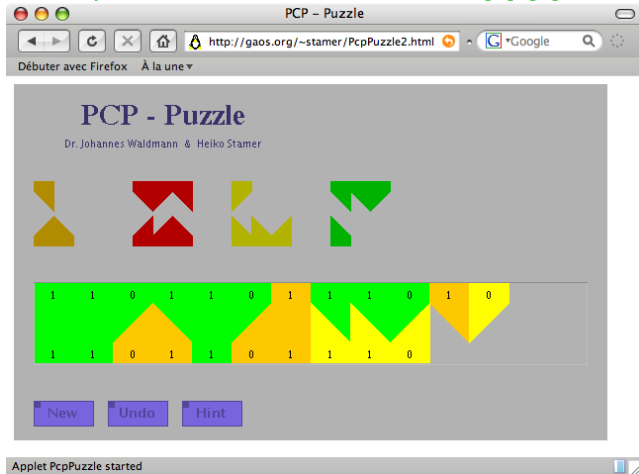
Problème 6 : Eternity II



16/131

Problème 9 : PCP, Problème de correspondance de Post

http://www.theory.informatik.uni-kassel.de/~stamer/pcp/pcpcontest_en.html



17/131

Codages

- ▶ En informatique, on raisonne en **binaire** : tout peut être codé sur l'alphabet binaire $\Sigma = \{0, 1\}$.
- ▶ Formellement, **coder** un domaine D de valeurs en binaire signifie trouver une fonction de **codage** $f : D \rightarrow \{0, 1\}^*$, c-à-d. une fonction (calculable) t.q. $f(d) \neq f(d')$ pour tout $d \neq d'$ dans D .
- ▶ Exemple : on peut coder un graphe G avec ensemble de sommets $V = \{1, 2, \dots, n\}$ et ensemble d'arêtes $E \subseteq V \times V$ de plusieurs façons :
 - ▶ par sa matrice d'adjacence $M \in \{0, 1\}^{n \times n}$,
 - ▶ par des listes d'adjacence,
 - ▶ ...
- ▶ On distingue la **taille de l'entrée** de la **taille du codage**.
Exemple : la taille d'un graphe $G = (V, E)$ où $|V| = n$, $|E| = m$ est $n + m$. Son codage par matrice est de taille n^2 , et celui par listes d'adjacence (en binaire) est de taille $O(n + m \log(n))$.

19/131

Un peu de vocabulaire

Un **problème de décision** est

- ▶ une **question**...
- ▶ ... portant sur un **ensemble** de données (= entrées) ...
- ▶ ... et dont la réponse est **OUI** ou **NON**.

Ne pas confondre **problème** et **algorithme** le résolvant.

Une **instance** du problème est la question posée sur une donnée/entrée particulière.

- ▶ On s'intéresse aussi aux **problèmes calculatoires**, dont la réponse n'est pas nécessairement binaire (OUI/NON).
- ▶ Exemple : **décider** si un graphe possède un 3-coloriage vs. **calculer** un 3-coloriage.

18/131

Des fonctions non-calculables, ça existe !

Un peu de comptage :

- ▶ Combien de mots binaires y a-t-il ? autant que des entiers positifs. On dit que $\{0, 1\}^*$ est **dénombrable**, voir chapitre suivant. Informellement, les mots binaires peuvent être énumérés en ordre lexicographique.
- ▶ Combien de programmes/algorithme existe-t-il ?
On peut coder chaque programme P par un mot en binaire $w_P \in \{0, 1\}^*$, il suffit de choisir son codage préféré (ASCII par exemple). Ensuite, on peut énumérer les programmes, en énumérant les mots $w \in \{0, 1\}^*$ représentant des programmes. L'ensemble des programmes est donc **dénombrable**.
- ▶ Combien de fonctions $f : \mathbb{N} \rightarrow \mathbb{N}$ (ou $f : \{0, 1\}^* \rightarrow \{0, 1\}^*$) existe-t-il ? Autant que des réels (en particulier **non-dénombrable**). **Conséquence** : il existe des fonctions $f : \mathbb{N} \rightarrow \mathbb{N}$ non-calculables.

20/131

Un exemple

Si vous vous souvenez comment on montre que l'ensemble des réels n'est **pas dénombrable**, vous pouvez imaginer comment on peut construire une fonction concrète qui n'est pas calculable. La technique s'appelle **diagonalisation** (Cantor).

- ▶ Enumérons tous les programmes, en fixant une fonction de codage qui code P par l'entier $w_P \in \mathbb{N}$.
- ▶ On définit la fonction $F : \mathbb{N} \rightarrow \mathbb{N}$ par

$$F(n) = \begin{cases} P(w_P) + 1 & \text{si } n = w_P \text{ et } P \text{ termine sur } w_P \\ 0 & \text{sinon} \end{cases}$$

Evidemment, la fonction F diffère de toute fonction calculable.

21/131

Plan

Présentation, bref historique

Ensembles dénombrables (rappels)

Modèles de calcul

Problèmes indécidables

Problèmes et réductions

La classe NP

Machines de Turing, hiérarchie de Chomsky

22/131

Ensembles dénombrables

- ▶ On note $\mathbb{N} = \{0, 1, 2, \dots\}$ l'ensemble des entiers naturels.
- ▶ Un ensemble E est dit **dénombrable** s'il est en bijection avec \mathbb{N} .
- ▶ **Exemples.** Les ensembles suivants sont dénombrables :
 1. $\mathbb{N} = \{0, 1, \dots\}$,
 2. \mathbb{Z} : les entiers,
 3. $\mathbb{N} \times \mathbb{N}$: paires d'entiers positifs ; \mathbb{N}^k ($k > 2$) : les k -uplets,
 4. \mathbb{Q} : les rationnels,
 5. l'ensemble des matrices avec entrées entières,
 6. l'ensemble Σ^* des mots sur un alphabet (fini) Σ ,
 7. l'ensemble des programmes C ,
 8. l'ensemble des arbres orientés,
 9. l'ensemble des graphes.

23/131

Quelques ensembles non dénombrables

Exemples Les ensembles suivants ne sont pas dénombrables :

1. l'ensemble des nombres réels,
2. l'ensemble des suites infinies d'entiers,
3. l'ensemble des parties de \mathbb{N} ;
4. l'ensemble des fonctions de \mathbb{N} dans $\{0, 1\}$.

Note Les trois derniers ensembles sont en bijection.

À une partie X de \mathbb{N} , on associe la suite $x = (x_n)_{n \geq 0}$ définie par

$$x_n = \begin{cases} 1 & \text{si } n \in X \\ 0 & \text{si } n \notin X \end{cases}$$

De même, à toute suite $x = (x_n)_{n \geq 0}$ à valeurs dans $\{0, 1\}$, on associe bijectivement la fonction $f_x : \mathbb{N} \rightarrow \{0, 1\}$ définie par $f_x(n) = x_n$.

24/131

Quelques ensembles non dénombrables

L'argument ci-dessous, dû à Cantor, permet de montrer que l'ensemble $E = \{0, 1\}^{\mathbb{N}}$ des suites infinies de 0 ou 1 est non dénombrable.

- ▶ Supposons par contradiction qu'on peut énumérer $E : e^1, e^2, \dots$
- ▶ Soit $x = (x_n)_{n \geq 0}$ la suite infinie de 0 ou 1 définie par

$$x_n = 1 - (e^n)_n$$

- ▶ Puisque $x_n \neq (e^n)_n$, on a $x \neq e^n$, et ce, pour tout $n \in \mathbb{N}$.
- ▶ Comme x n'est pas de la forme e^n , on déduit que $x \notin E$, contradiction.

25/131

Plan

Présentation, bref historique

Ensembles dénombrables (rappels)

Modèles de calcul

Problèmes indécidables

Problèmes et réductions

La classe NP

Machines de Turing, hiérarchie de Chomsky

26/131

LOOP, WHILE, GOTO

- ▶ On commence par quelques "langages de programmation" basiques : **LOOP**, **WHILE** et **GOTO**.
- ▶ A chacun de ces langages on associe une classe de fonctions calculables par des programmes de ce langage.
- ▶ On va montrer qu'une fonction est **WHILE**-calculable ssi elle est **GOTO**-calculable. Par contre, la classe des fonctions **LOOP**-calculables est strictement plus petite.
- ▶ On va présenter ensuite d'autres modèles de calcul, les **fonctions récursives** (plus tard, les **machines de Turing**), et montrer qu'ils définissent la même classe de fonctions calculables que les programmes **WHILE**.

27/131

Programmes LOOP

- ▶ Exemple 1 : addition

```
x := y;
LOOP (z) DO x := x+1 OD;
Calcule x = y + z.
```

- ▶ Exemple 2 : multiplication

```
x := 0;
LOOP (z) DO
  LOOP (y) DO x := x+1 OD OD
Calcule x = y · z.
```

Syntaxe :

- ▶ variables res, x, y, z, \dots (valuées dans \mathbb{N})
- ▶ constantes $0, 1, \dots$
- ▶ opérations $+, -$
- ▶ instructions $x := y \pm c$, $x := c$, **LOOP** (x) **DO** P **OD**

L'effet de $x := y \pm c$ est d'affecter à x la valeur $\max(y \pm c, 0)$.

L'effet de "**LOOP** (x) **DO** P **OD**" est d'itérer x fois le programme P .

28/131

LOOP-calculable

- ▶ Soit P un programme LOOP utilisant les variables $x_0 = \text{res}, x_1, \dots, x_\ell$.
 - ▶ L'entrée de P est un k -uplet $\vec{n} = (n_1, \dots, n_k) \in \mathbb{N}^k$ ($k < \ell$).
 - ▶ L'effet de P sur $\vec{x} = x_0, \dots, x_\ell$ (défini inductivement) est noté $P(\vec{x}) \in \mathbb{N}^{\ell+1}$.
 - ▶ La sortie de P est la valeur de la variable res. La fonction calculée par P est notée f_P .
- ▶ Une fonction $f : \mathbb{N}^k \rightarrow \mathbb{N}$ est **LOOP-calculable**, s'il existe un programme LOOP P t.q. $f(\vec{n}) = f_P(0, \vec{n}, 0, \dots, 0)$.
- ▶ Rq : Un programme LOOP **termine toujours**, donc les fonctions LOOP-calculables sont **totales** (c-à-d, définies partout).

Exercice : montrez que l'instruction (IF $(x = 0)$ THEN P ELSE Q FI) est LOOP-calculable.

29/131

Programmes WHILE

- ▶ Les programmes WHILE sont définis à partir des programmes LOOP, en rajoutant l'instruction "while" :
 $WHILE (x \neq 0) DO P OD$
- ▶ Une fonction $f : \mathbb{N}^k \rightarrow \mathbb{N}$ est **WHILE-calculable** s'il existe un programme WHILE P t.q. $f = f_P$.
- ▶ Par définition, toute fonction LOOP-calculable est aussi WHILE-calculable.
- ▶ **Attention** : Les fonctions WHILE-calculables **ne sont pas totales**.
Exemple :
 $x := 1;$
 $WHILE (x \neq 0) DO (x := x + 1) OD$

30/131

Programmes GOTO

Syntaxe : un programme $P = I_1; \dots; I_m$ est une séquence (numérotée) d'instructions I_j .

- ▶ Variables $x, y, \dots \in \mathbb{N}$, constantes $0, 1, \dots$
- ▶ Instructions :
 - ▶ $x := y \pm c, x := c$
 - ▶ IF $(x = 0)$ THEN GOTO ℓ FI (saut conditionnel)
 - ▶ GOTO ℓ (saut non-conditionnel)
 - ▶ HALT

Exemple (addition) :

Sémantique des programmes GOTO :

- | | |
|----------------------------------|---|
| (1) $x := y;$ | ▶ L'instruction HALT est la dernière. |
| (2) IF $(z = 0)$ THEN GOTO 5 FI; | ▶ Saut conditionnel : si x est zéro, |
| (3) $x := x + 1; z := z - 1;$ | continuer avec l'instruction I_ℓ , |
| (4) GOTO 2; | sinon avec l'instruction suivante |
| (5) HALT | (sauf si HALT). |

31/131

Prop.

Pour toute fonction $f : \mathbb{N} \rightarrow \mathbb{N}$:

f est WHILE-calculable ssi² f est GOTO-calculable

Rq : tout programme WHILE peut être réécrit en un programme utilisant **une seule boucle WHILE**.

Prop. (voir plus loin)

Il existe des fonctions WHILE-calculables, qui ne sont pas LOOP-calculables.

Exemple : fonction d'Ackermann.

2. ssi = "si et seulement si"

32/131

La classe des fonctions primitives récursives

On introduit maintenant un modèle de calcul plus mathématique, basé sur l'arithmétique : **les fonction primitives récursives et les fonction récursives**.

Les fonctions primitives récursives (p.r.) sont construites à partir de :

- ▶ fonctions constantes zéro $c_0^k : \mathbb{N}^k \rightarrow \mathbb{N}$ (pour tout $k \geq 0$) :
 $c_0^k(x) = 0$ pour tout $x \in \mathbb{N}^k$,
- ▶ la fonction successeur $\text{Succ} : \mathbb{N} \rightarrow \mathbb{N} : \text{Succ}(x) = x + 1$,
- ▶ la projection $\pi_i^k : \mathbb{N}^k \rightarrow \mathbb{N} : \pi_i^k(x_1, \dots, x_k) = x_i$,

en utilisant la **composition** et la **réursion**.

33/131

Fonctions primitives-récurives (p.r.), cont.

- ▶ La **composition** : si $f_1, \dots, f_k : \mathbb{N}^m \rightarrow \mathbb{N}$ et $g : \mathbb{N}^k \rightarrow \mathbb{N}$ sont p.r., alors $h : \mathbb{N}^m \rightarrow \mathbb{N}$ l'est aussi :

$$h(x_1, \dots, x_m) = g(f_1(x_1, \dots, x_m), \dots, f_k(x_1, \dots, x_m))$$

- ▶ La **réursion** : si $f : \mathbb{N}^k \rightarrow \mathbb{N}$ et $g : \mathbb{N}^{k+2} \rightarrow \mathbb{N}$ sont p.r., alors $h : \mathbb{N}^{k+1} \rightarrow \mathbb{N}$ l'est aussi :

$$h(0, x_1, \dots, x_k) = f(x_1, \dots, x_k)$$

$$h(n + 1, x_1, \dots, x_k) = g(n, h(n, x_1, \dots, x_k), x_1, \dots, x_k)$$

34/131

Exemples de fonctions primitives récursives

Exemples : Les fonctions suivantes sont primitives récursives.

- ▶ L'addition, la multiplication de deux entiers.
- ▶ La fonction sign : $\mathbb{N} \rightarrow \mathbb{N}$ définie par $\text{sign}(0) = 0$, et $\text{sign}(x) = 1$ si $x > 0$.
- ▶ La fonction prédécesseur : $\text{pred}(0) = 0$, $\text{pred}(x) = x - 1$ si $x > 0$.
- ▶ La différence tronquée : maximum de 0 et de la différence de deux entiers.

$$x \dot{-} y \equiv \max\{0, x - y\}$$

- ▶ Le minimum de deux entiers.

35/131

Propriétés des fonctions p.r.

- ▶ La somme, le produit, la différence tronquée de fonctions p.r. est aussi p.r.
- ▶ Si $f : \mathbb{N}^{p+1} \rightarrow \mathbb{N}$ est p.r., alors

$$g(n, x_1, \dots, x_p) = \sum_{i=0}^n f(i, x_1, \dots, x_p)$$

et

$$h(n, x_1, \dots, x_p) = \prod_{i=0}^n f(i, x_1, \dots, x_p)$$

sont p.r.

36/131

Fonctions définies par cas

- ▶ Une fonction dont les valeurs sont 0 ou 1 est appelée un **prédicat**.
- ▶ On interprète 0 comme **faux** et 1 comme **vrai**.
- ▶ Les opérateurs Booléens classiques, appliqués à des prédicats p.r. fournissent des prédicats p.r.
- ▶ Si $P(x, x_1, \dots, x_p)$ est un prédicat p.r., il en est de même de
 - ▶ $A_p(n, x_1, \dots, x_p) \equiv \forall x \leq n P(x, x_1, \dots, x_p)$
 - ▶ $E_p(n, x_1, \dots, x_p) \equiv \exists x \leq n P(x, x_1, \dots, x_p)$
- ▶ ... car $A_p(n, x_1, \dots, x_p) = \prod_{i=0}^n P(i, x_1, \dots, x_p)$, et $E_p = 1 \dot{-} A_{1 \dot{-} p}$.

37/131

Fonctions définies par cas

- ▶ Si P_1, \dots, P_k sont des prédicats p.r., et f_1, \dots, f_{k+1} des fonctions p.r., il en est de même de

$$g(x_1, \dots, x_n) = \begin{cases} f_1(x_1, \dots, x_n) & \text{si } P_1(x_1, \dots, x_n), \\ f_2(x_1, \dots, x_n) & \text{sinon, et si } P_2(x_1, \dots, x_n), \\ \dots & \\ f_k(x_1, \dots, x_n) & \text{sinon, et si } P_k(x_1, \dots, x_n), \\ f_{k+1}(x_1, \dots, x_n) & \text{sinon.} \end{cases}$$

38/131

Minimisation bornée

- ▶ Si $P(x, x_1, \dots, x_p)$ est un prédicat p.r., la fonction

$$\min_B P(n, x_1, \dots, x_k) \equiv \begin{cases} \min \{ x \leq n \mid P(x, x_1, \dots, x_k) = 1 \} & \text{si tel } x \text{ existe} \\ n + 1 & \text{sinon} \end{cases}$$

est aussi p.r.

- ▶ On a $\min_B P(n, x_1, \dots, x_k) = \sum_{j=0}^n \prod_{i=0}^j (1 \dot{-} P(i, x_1, \dots, x_k))$.
- ▶ Rq : On ne peut pas se passer de **borner** cette minimisation (sinon, la fonction n'est plus nécessairement p.r.).

39/131

Primitif récursif = LOOP-calculable

Théorème

Une fonction : $\mathbb{N}^k \rightarrow \mathbb{N}$ est primitive récursive ssi elle est LOOP-calculable.

On montre seulement "p.r. \Rightarrow LOOP-calculable". La preuve se fait selon la définition inductive des fonctions p.r.

Récursion : si $f : \mathbb{N}^k \rightarrow \mathbb{N}$ et $g : \mathbb{N}^{k+2} \rightarrow \mathbb{N}$ sont p.r., alors $h : \mathbb{N}^{k+1} \rightarrow \mathbb{N}$ l'est aussi :

$$\begin{aligned} h(0, x_1, \dots, x_k) &= f(x_1, \dots, x_k) \\ h(n + 1, x_1, \dots, x_k) &= g(n, h(n, x_1, \dots, x_k), x_1, \dots, x_k) \end{aligned}$$

Soient P_f et P_g des programmes LOOP calculant f et g , resp. On construit un programme LOOP calculant $h(n, x_1, \dots, x_k)$:

40/131

p.r. ⇒ LOOP-calculable

```

n' := 0; x'_1 := x_1; ... x'_k := x_k;
P_f(res, x'_1, ..., x'_k);
z := res;
LOOP (n) DO
  x'_1 := x_1; ... x'_k := x_k;
  P_g(res, n', z, x'_1, ..., x'_k);
  z := res;
  n' := n' + 1;
OD
res := z;
  
```

41/131

Récurrance simultanée

- ▶ On note $\vec{x} = x_1, \dots, x_p$.
- ▶ **Propriété** Si $f_1, \dots, f_k : \mathbb{N}^p \rightarrow \mathbb{N}$ et $g_1, \dots, g_k : \mathbb{N}^{k+p+1} \rightarrow \mathbb{N}$ sont des fonctions p.r., alors les fonctions $h_1, \dots, h_k : \mathbb{N}^{p+1} \rightarrow \mathbb{N}$ définies comme suit le sont aussi.

$$h_i(\mathbf{0}, \vec{x}) = f_i(\vec{x})$$

$$h_i(n+1, \vec{x}) = g_i(n, h_1(n, \vec{x}), \dots, h_k(n, \vec{x}), \vec{x})$$

- ▶ **Preuve.** Pour $k = 2$, on utilise le codage $c : \mathbb{N}^2 \rightarrow \mathbb{N}$ et ses décodages d_1 et d_2 pour exprimer $h = c(h_1, h_2)$ à l'aide du schéma de récurrence.

43/131

Bijection $\mathbb{N}^2 \rightarrow \mathbb{N}$ p.r.

- ▶ La bijection $c : \mathbb{N}^2 \rightarrow \mathbb{N}$ définie par

$$c(i, j) = \frac{(i+j)(i+j+1)}{2} + i$$

est p.r.
 (montrer que $n \mapsto n/2$ est p.r.).

- ▶ Les fonctions $d_1, d_2 : \mathbb{N} \rightarrow \mathbb{N}$ telles que $c^{-1} = (d_1, d_2)$ sont aussi p.r.. Par exemple, pour d_1 , utiliser

$$d_1(x+1) = \begin{cases} 0 & \text{si } \exists y \leq x, \quad x = \frac{y(y+1)}{2}, \\ d_1(x) + 1 & \text{sinon.} \end{cases}$$

42/131

Fonctions WHILE-calculables, non p.r.

- ▶ Ackermann et Sudan ont trouvé en 1927-28 des fonctions non p.r., mais WHILE-calculables.

$$A(m, x) = \begin{cases} x + 1 & \text{si } m = 0 \\ A(m-1, 1) & \text{si } m > 0, x = 0 \\ A(m-1, A(m, x-1)) & \text{si } m, x > 0. \end{cases}$$

- ▶ Variante [D. Kozen] : $B(m, x) = B_m(x)$, où

$$B_0(x) = x + 1$$

$$B_{m+1}(x) = \underbrace{B_m \circ \dots \circ B_m}_{x \text{ fois}}(x)$$

44/131

Fonctions calculables, non primitives récursives

- ▶ On vérifie que les récurrences **terminent**.
- ▶ **B croît trop vite** pour être p.r. (idem pour A).

$$B_0(x) = x + 1, B_1(x) = 2x, B_2(x) \geq 2^x$$

$$B_3(x) \geq \underbrace{2^{2^{\dots^2}}}_x = 2 \uparrow x$$

$$B_4(x) \geq 2 \uparrow \uparrow x,$$

$$B_4(2) \geq 2^{2048} \dots$$

où $2 \uparrow (x + 1) = 2^{2^x}$, et $2 \uparrow \uparrow (x + 1) = 2 \uparrow (2 \uparrow \uparrow x)$, et plus généralement $B_{m+2}(x) \geq 2 \uparrow \dots \uparrow_m \text{ fois}(x)$, où

$$2 \uparrow \dots \uparrow (x + 1) = 2 \underbrace{\uparrow \dots \uparrow}_{m-1 \text{ fois}} \underbrace{(2 \uparrow \dots \uparrow x)}_{m \text{ fois}}$$

On retrouve la définition d'Ackermann.

La fonction d'Ackermann est WHILE-calculable, mais n'est pas p.r.

- ▶ On démontre que si $f : \mathbb{N}^p \rightarrow \mathbb{N}$ est p.r., alors il existe k tel que

$$f(x_1, x_2, \dots, x_p) < B_k(\max(x_1, \dots, x_p)). \quad (1)$$

- ▶ Or, la fonction B ne vérifie pas (1).
- ▶ La fonction B n'est donc pas p.r.
- ▶ Idem pour A.
- ▶ Mais : chaque B_k est p.r.
- ▶ A et B sont WHILE-calculables.

Fonctions récursives

- ▶ Soit $P(y, \vec{x}) : \mathbb{N}^{p+1} \rightarrow \{0, 1\}$ un prédicat **non nécessairement total**, (c'est-à-dire, non nécessairement défini sur tout \mathbb{N}^{p+1}).
- ▶ On écrit $P(y, \vec{x}) = a$ si P est **défini** sur (y, \vec{x}) et vaut a .
- ▶ On définit la fonction $\text{Min } P : \mathbb{N}^p \rightarrow \mathbb{N}$:

$$\text{Min } P(\vec{x}) = \begin{cases} y & \text{si } P(y, \vec{x}) = 1 \text{ et} \\ & P(z, \vec{x}) = 0 \text{ pour tout } 0 \leq z < y \\ \text{indéfini} & \text{sinon.} \end{cases}$$

- ▶ **Attention** : l'opérateur Min défini précédemment prend en argument une fonction (prédicat P) et retourne une autre fonction ($\text{Min } P$). Ceci n'est pas à confondre avec l'opération $\text{min} : \mathbb{N}^2 \rightarrow \mathbb{N}$, qui calcule le minimum de deux entiers naturels.

Récurif = WHILE-calculable

Si $P(y, \vec{x})$ est un prédicat WHILE-calculable, $\text{Min } P(\vec{x})$ est WHILE-calculable :

```
y := 0;
WHILE (P(y, x̄) = 0) DO y := y + 1 OD;
res = y;
```

Rq : Le calcul peut ne pas terminer

- ▶ soit parce que l'un des appels $P(y, \vec{x})$ ne termine pas,
- ▶ soit parce que $P(y, \vec{x})$ vaut toujours 0.

La classe des fonctions récursives

Plus petite classe de fonctions $\mathbb{N}^k \rightarrow \mathbb{N}$

- ▶ contenant les **constantes zéro**, la fonction **successeur**, la **projection**,
- ▶ fermée par
 - ▶ **composition**,
 - ▶ **réursion**,
 - ▶ **minimisation** : si P est un prédicat récursif, alors $\text{Min } P$ l'est aussi.

49/131

Fonctions récursives et fonctions WHILE-calculables

Théorème

- ▶ Les fonctions récursives sont exactement les fonctions WHILE-calculables.
- ▶ Il existe une fonction récursive, définie sur \mathbb{N} tout entier, qui n'est pas primitive récursive.
- ▶ Il existe des fonctions non récursives.

50/131

Plan

Présentation, bref historique

Ensembles dénombrables (rappels)

Modèles de calcul

Problèmes indécidables

Problèmes et réductions

La classe NP

Machines de Turing, hiérarchie de Chomsky

51/131

Rappels et définitions

- ▶ Problèmes de **décision** versus problèmes de **calcul** : réponse OUI/NON pour les premiers, résultat pour les autres. La première catégorie est juste un cas spécial de la deuxième.
- ▶ Comme pour les fonctions, on peut parler des programmes WHILE qui donnent une solution à un problème de décision.

- ▶ Un problème est dit **semi-décidable** s'il existe un programme WHILE qui le résout. **Attention** : On **ne demande pas** que le programme WHILE termine toujours !
- ▶ Un problème est dit **décidable** s'il existe un programme WHILE qui le résout **et qui termine sur toute entrée**.

52/131

Vocabulaire

- ▶ Dans la littérature on dit soit semi-décidable ou **récurivement énumérable**. La raison est que un problème est semi-décidable ssi l'ensemble des instances positives est énumérable (voir TD).
- ▶ On dit aussi soit décidable ou **récurif**. Raison : un problème est décidable ssi l'ensemble des instances positives est énumérable en ordre lexicographique (TD).
- ▶ **Indécidable** = pas décidable.

53/131

RE et R

On note RE la classe des problèmes semi-décidables, et R celle des problèmes décidables.

- ▶ On a clairement $R \subseteq RE$.
- ▶ La classe R est fermée par complémentaire (le complémentaire d'un problème signifie d'inverser les réponses OUI/NON).
- ▶ Si un problème P et son complémentaire P^{co} sont semi-décidables, alors ils sont tous les deux décidables.
 - ▶ On met ensemble le programme WHILE π qui résout P et le programme WHILE π' qui résout P^{co} : le programme WHILE composé simule un pas de calcul de π , ensuite un pas de π' , ensuite un pas de π , etc.
 - ▶ Le programme s'arrête si π ou π' s'arrête, et retourne OUI si π répond OUI, et NON si π' répond OUI.
 - ▶ Une instance est soit positive pour P ou pour P^{co} , donc le programme construit **s'arrête toujours**.

54/131

Rappel : problèmes qui ne sont pas semi-décidables (voir page 20)

- ▶ On peut coder chaque programme WHILE d'abord en binaire, ensuite par un entier.
- ▶ On s'intéresse aux programmes WHILE qui reçoivent un entier en entrée. On note $Acc(P) \subseteq \mathbb{N}$ l'ensemble des entiers n sur lesquels P termine et retourne 1.
- ▶ On note P_n le programme WHILE codé par l'entier n (si n ne code aucun programme, alors P_n est le programme vide).
- ▶ Le problème DIAG est défini par :
 1. L'entrée de DIAG est un entier n .
 2. n est une instance positive si $n \notin Acc(P_n)$.
- ▶ **Proposition** Le problème DIAG n'est **pas semi-décidable**.

55/131

Un problème semi-décidable, mais pas décidable

Le complémentaire $DIAG^{co}$ de DIAG est le problème suivant, noté aussi UNIV ("langage universel") :

- ▶ L'entrée est un entier n .
- ▶ n est une instance positive si $n \in Acc(P_n)$.

Le problème UNIV est **semi-décidable** : il suffit de **simuler** le programme P_n sur l'entrée n . (On peut construire un programme WHILE qui, à partir de l'entrée n , récupère le programme P_n et le simule.)

Par contre, UNIV est **indécidable** - car sinon, DIAG et UNIV seraient décidables tous les deux.

56/131

Réductions

- ▶ Soient P et P' deux problèmes.
- ▶ On note $X \subseteq D$ l'ensemble des instances positives de P .
- ▶ On note $X' \subseteq D'$ l'ensemble des instances positives de P' .
- ▶ Une **réduction** de P vers P' est une **fonction calculable** $f : D \rightarrow D'$ telle que

$$x \in X \iff f(x) \in X'.$$

- ▶ On note $P \leq P'$ (lit : P se réduit à P')
- ▶ L'existence d'une réduction de P vers P' assure que
 - ▶ si P' est décidable, P l'est aussi,
 - ▶ si P est indécidable, P' l'est aussi.

57/131

Remarques

- ▶ " P se réduit à P' " ne signifie **PAS** que P' est plus facile que P . Cela signifie plutôt que la recherche d'une solution pour P sur une instance x donnée peut être ramenée à la recherche d'une solution pour P' sur l'instance $f(x)$.
- ▶ La notion de réduction est **symétrique** : x est une instance positive de P **SI ET SEULEMENT SI** $f(x)$ est une instance positive de P' .
Mais : $P \leq P'$ n'implique pas $P' \leq P$.
- ▶ Les réductions sont transitives : si $P_1 \leq P_2$ et $P_2 \leq P_3$, alors $P_1 \leq P_3$.

Exemple de réduction : $UNIV \leq UNIV_0$, où $UNIV_0$ est le problème suivant :

- ▶ Données : entiers n, m .
- ▶ Question : est-ce que $m \in Acc(P_n)$?

58/131

Problème de l'arrêt

Les problèmes suivants sont **indécidables** :

- ▶ $HALT$: étant donné un programme WHILE P et un entier n , est-ce que P s'arrête sur n ?
- ▶ $HALT_0$: étant donné un programme WHILE P , est-ce que P s'arrête sur 0?
- ▶ étant donné un programme WHILE P et une instruction I , P utilise-t-il l'instruction I sur l'entrée 0?
- ▶ $HALT_{\exists}$: étant donné un programme WHILE P , est-ce que P s'arrête sur au moins une entrée?
- ▶ $HALT_{\forall}$: étant donné un programme WHILE P , est-ce que P s'arrête sur toute entrée?
- ▶ étant donné deux programmes WHILE P_1, P_2 , est-ce que $Acc(P_1) = Acc(P_2)$?

Rq : Les 3 premières questions sont semi-décidables, les 2 dernières ne le sont même pas.

59/131

L'indécidabilité hors du monde des entiers : le PCP

- ▶ Problème de correspondance de Post (1946).
- ▶ **Donnée** : n paires de mots $(u_1, v_1), \dots, (u_n, v_n)$.
- ▶ **Question** : existe-il une suite finie i_1, \dots, i_k telle que

$$u_{i_1} \cdots u_{i_k} = v_{i_1} \cdots v_{i_k}$$

[A noter : les suites d'indices sont les mêmes.]

- ▶ \rightsquigarrow On peut voir les couples de mots comme des dominos.

a	aa	ba	bab
ab	a	aa	abba

- ▶ Une solution : $a.bab.ba.aa.aa = ab.abba.aa.a.a$
- ▶ Suite d'indices : 1, 4, 3, 2, 2.

60/131

Le PCP modifié (PCPM)

- ▶ Problème de correspondance de Post (1946).
- ▶ Donnée : n paires de mots $(u_1, v_1), \dots, (u_n, v_n)$.
- ▶ Question : existe-il une suite finie i_1, \dots, i_k telle que $i_1 = 1$ et

$$u_{i_1} \cdots u_{i_k} = v_{i_1} \cdots v_{i_k}$$

A noter : les suites d'indices sont les mêmes, ...
... et le premier indice est 1.

61/131

Indécidabilité du PCP et PCPM

- ▶ On montre que

$$\text{UNIV}_0 \leq \text{PCPM} \leq \text{PCP}.$$

- ▶ Comme UNIV_0 est indécidable, il en est de même de PCPM et de PCP.
- ▶ Accessoirement, on peut montrer que $\text{PCP} \leq \text{PCPM}$.

62/131

PCP \leq PCPM

- ▶ Si on a un algorithme pour résoudre PCPM, on a un algorithme pour résoudre PCP.
- ▶ Il suffit de résoudre n PCPM différents, selon le mot avec lequel on commence.

63/131

PCPM \leq PCP : plus difficile

- ▶ Supposons donné un algorithme pour résoudre le PCP.
- ▶ On introduit une nouvelle lettre \$, et pour $a_1 \cdots a_k \in A^+$, soient $p(a_1 \cdots a_k) = \$a_1 \cdots \a_k et $s(a_1 \cdots a_k) = a_1 \$ \cdots a_k \$$.
- ▶ Soit $(u_1, v_1), \dots, (u_n, v_n)$ une instance de PCPM.
- ▶ Soient les $2n + 1$ mots suivants :

$$\begin{aligned} x_i &= p(u_i), & y_i &= s(v_i) \\ x_{n+i} &= p(u_i)\$, & y_{n+i} &= s(v_i) \\ x_{2n+1} &= p(u_1), & y_{2n+1} &= \$s(v_1). \end{aligned}$$

- ▶ Le PCPM sur l'instance $((u_\ell, v_\ell))_{1 \leq \ell \leq n}$ a une solution si et seulement si le PCP sur l'instance $((x_\ell, y_\ell))_{1 \leq \ell \leq 2n+1}$ en a une.

64/131

Indécidabilité de PCPM

- ▶ Idée générale : On simule un calcul de programme WHILE avec les dominos de PCP. Pour simplifier, on va simuler un programme GOTO P avec 2 variables x, y , dont les instructions arithmétiques sont $x := x \pm 1$ et $y := y \pm 1$ (ceci n'est pas une restriction) sur l'entrée 0. Soit p le nombre d'instructions de P .
- ▶ Une configuration de P est un triplet : instruction actuelle k + valeurs actuelles m, n des variables x, y . La configuration initiale sera le triplet $(1, 0, 0)$, n étant l'entrée de P .
- ▶ On code une configuration par un mot sur l'alphabet $\{1, \dots, p\} \cup \{0, 1, 0, 1\}$. Prenons comme exemple la configuration $(3, 4, 6)$. Elle sera codée par le mot :

3 0 3 0 3 1 3 0 3 1 3 1

en représentant 4 et 6 en binaire (bit le plus significatif à droite). Dans ce codage, on a intercalé le numéro de l'instruction après chaque bit.

65/131

Indécidabilité de PCPM

Idée générale : On définit l'instance de PCPM de telle façon que la seule solution sera la suite des configurations de P sur n , qui commence par la configuration initiale et finit par la configuration finale avec $x = y = 0$.

- ▶ Le premier domino représente la configuration initiale :

$$\begin{bmatrix} 1 & 0 & 1 & 0 \end{bmatrix}$$

- ▶ La partie du bas est en retard d'une configuration. On rattrape avec les dominos et on génère en même temps la configuration suivante.

66/131

Exemple : on veut incrémenter le nombre 11, écrit en binaire comme 1101 (bit le moins significatif à gauche). Le résultat de l'incrémement est 0011. D'abord on va marquer le premier 0 (de gauche), pour arriver à 1101.

L'instance suivante de PCPM réalise d'abord ce marquage :

$$\begin{bmatrix} \#1101 \\ \#1 \end{bmatrix} \begin{bmatrix} \$1 \\ \$0 \end{bmatrix} \begin{bmatrix} 11 \\ 11 \end{bmatrix} \begin{bmatrix} 10 \\ 10 \end{bmatrix} \begin{bmatrix} 0 \\ 0 \end{bmatrix} \begin{bmatrix} 1 \\ 1 \end{bmatrix} \begin{bmatrix} \$ \\ \$ \end{bmatrix}$$

En commençant par le premier domino $(\#1101, \epsilon)$ il y a une seule façon d'utiliser les dominos, et à la fin on obtient $(w \#1101, w)$, donc la partie haute est en avance par $\$1101$:

#1101 \$1 1 0 1 \$ 11 0 1 \$ 1 10 1
#1 1 0 1 \$ 11 0 1 \$ 1 10 1

Si on rajoute les dominos $\begin{bmatrix} \#1 \\ \$0 \end{bmatrix}$, $\begin{bmatrix} 11 \\ 10 \end{bmatrix}$, $\begin{bmatrix} 10 \\ 11 \end{bmatrix}$, $\begin{bmatrix} \#0 \\ \$1 \end{bmatrix}$, alors on obtient une avancée de la partie haute de $\#0011$, donc le résultat de l'incrémement.

67/131

Quelques autres problèmes indécidables

Les problèmes suivants sont indécidables :

- ▶ Étant donné un jeu fini de tuiles carrées, avec conditions de compatibilité entre côtés, déterminer si on peut paver le $1/4$ de plan.
- ▶ Étant donnée une grammaire, déterminer si elle est ambiguë.
- ▶ Étant donné un nombre fini de matrices 3×3 à coefficients entiers, déterminer si un produit permet d'annuler la composante $(3,2)$.
- ▶ Étant donnée une suite calculable d'entiers, déterminer si elle converge.

68/131

Simple-PCP

- ▶ Dans **Simple-PCP** on a les restrictions suivantes :
 1. u_i et v_i ont la **même longueur**, pour tout $2 \leq i \leq n - 1$ (on suppose $n \neq 1$).
 2. La solution de PCP doit commencer par l'indice 1 et terminer par l'indice n . De plus, ces indices ne peuvent pas être utilisés au milieu.
- ▶ Exemple : $(u_1, v_1) = (ab, a)$, $(u_2, v_2) = (aa, ba)$ et $(u_3, v_3) = (a, aa)$. La séquence 1,2,3 est une solution de **Simple-PCP**.
- ▶ Rq. 1 : pour qu'une solution existe, il faut que $|u_1| - |v_1| = |v_n| - |u_n|$. Soit donc $k := |u_1| - |v_1| = |v_n| - |u_n|$ et supposons que $k > 0$.
- ▶ Rq. 2 : Pour chaque séquence $1 = i_1, i_2, \dots, i_k$ (où $i_2, \dots, i_k \in \{2, \dots, n - 1\}$) on a :

$$|u_{i_1} \dots u_{i_k}| - |v_{i_1} \dots v_{i_k}| = k$$

69/131

Réduction de **Simple-PCP** au problème d'accessibilité

- ▶ On peut chercher une solution pour une instance Π de **Simple-PCP** dans un graphe orienté *fini* G_Π : les sommets sont les mots de longueur k ; on a un arc de u vers v s'il existe un couple (u_j, v_j) tel que $u u_j = v_j v$.
Le sommet de départ est le mot w tel que $u_1 = v_1 w$ et le sommet d'arrivée est w' tel que $w' u_n = v_n$. L'instance Π de **Simple-PCP** a une solution **si et seulement si** il existe un chemin dans G_Π de w à w' .
- ▶ On a donc réduit **Simple-PCP** au problème d'accessibilité dans les graphes orientés. Comme ce dernier problème est décidable, **Simple-PCP** l'est aussi.
- ▶ C'est possible de réduire aussi dans le sens inverse, du problème d'accessibilité à **Simple-PCP**.

70/131

Arrêt borné

Problème de l'arrêt borné :

- ▶ **Données** : programme (WHILE) P , entrée $n \in \mathbb{N}$ et entier $k \in \mathbb{N}$.
- ▶ **Question** : Est-ce que P termine sur n en moins de k pas ?

L'arrêt borné est décidable, il suffit de rajouter à P une horloge et de s'arrêter quand elle atteint k .

Problème des valeurs bornées :

- ▶ **Données** : programme (WHILE) P , entrée $n \in \mathbb{N}$ et entier $k > n$.
- ▶ **Question** : Est-ce que P termine sur n avant que ses variables dépassent la valeur k ?

Ce problème est également décidable (pourquoi ?)

71/131

D'autres problèmes indécidables

Le problème suivant :

- ▶ Entrée : programme (WHILE) P .
- ▶ Sortie : OUI si $Acc(P) \neq \emptyset$.

est semi-décidable, mais pas décidable :

- ▶ Semi-décidabilité : on énumère les paires $(n, k) \in \mathbb{N}^2$ et on simule k pas de P sur n . Si la simulation s'arrête et $res = 1$, on accepte. Si non, on passe au couple suivant.
- ▶ Non décidabilité : réduction à partir de $UNIV_0$.

Il s'en suit que son complémentaire

- ▶ Entrée : programme (WHILE) P .
- ▶ Sortie : OUI si $Acc(P) = \emptyset$.

n'est pas semi-décidable.

Le théorème suivant (Rice) généralise cet exemple. Il parle de

72/131

Théorème de Rice

- ▶ Soit \mathcal{E} l'ensemble des $X \subseteq \mathbb{N}$ tels que $X = \text{Acc}(P)$ pour un programme P .
- ▶ Une **propriété d'ensembles semi-décidables** est un sous-ensemble \mathcal{P} de \mathcal{E} .
- ▶ Une propriété $\mathcal{P} \subseteq \mathcal{E}$ est **triviale** si $\mathcal{P} = \emptyset$ ou $\mathcal{P} = \mathcal{E}$.
- ▶ **Attention** Ne pas confondre $\mathcal{P} = \emptyset$ (\mathcal{P} ne contient aucun ensemble) et $\mathcal{P} = \{\emptyset\}$ (\mathcal{P} ne contient que l'ensemble vide).
- ▶ **Attention** Un algorithme qui décide une propriété d'ensembles semi-décidables reçoit en entrée (le codage de) un programme P . Si l'algorithme répond OUI sur P , alors il doit répondre OUI sur tout P' tel que $\text{Acc}(P) = \text{Acc}(P')$.

73/131

Plan

Présentation, bref historique

Ensembles dénombrables (rappels)

Modèles de calcul

Problèmes indécidables

Problèmes et réductions

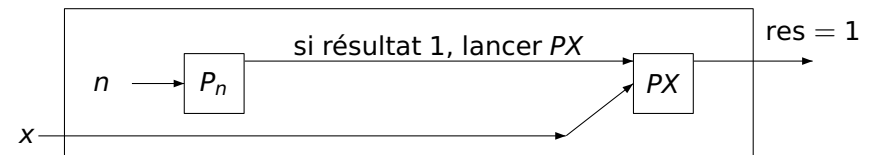
La classe NP

Machines de Turing, hiérarchie de Chomsky

75/131

Théorème de Rice

- ▶ **Toute propriété non triviale \mathcal{P} d'ensembles semi-décidables est indécidable.**
- ▶ **Re-attention** : il s'agit d'une propriété **d'ensembles**, et pas de programmes.
- ▶ **Preuve** A nouveau une réduction à partir de UNIV.
- ▶ Quitte à changer \mathcal{P} et $\mathcal{E} \setminus \mathcal{P}$, on peut supposer $\emptyset \notin \mathcal{P}$.
- ▶ Comme $\mathcal{P} \neq \emptyset$, il existe $X \in \mathcal{P}$. Soit PX un programme tel que $\text{Acc}(PX) = X$.
- ▶ À partir de n on construit (de façon effective) le programme R suivant :



- ▶ Soit $X_0 = \text{Acc}(R)$. On a $X_0 = \emptyset \notin \mathcal{P}$ si $n \notin \text{UNIV}$, et $X \in \mathcal{P}$ sinon.
- ▶ Donc si \mathcal{P} était décidable, UNIV le serait aussi. Contradiction.

74/131

Temps de calcul

Le temps de calcul d'un algorithme (programme) P se mesure en fonction de la taille de l'entrée. Soit l une entrée de P .

- ▶ $t_P(l)$ désigne le temps de calcul de P sur l . Le temps de calcul est le nombre d'instructions élémentaires exécutées (par exemple, instructions arithmétiques pour les programmes WHILE).
- ▶ La fonction $T_P : \mathbb{N} \rightarrow \mathbb{N}$ est définie par :

$$T_P(n) = \max\{t_P(l) \mid \text{taille}(l) = n\}$$

Il s'agit d'un temps de calcul **au pire des cas**.

- ▶ Un algorithme P est **polynomial** s'il existe un polynôme $p(n)$ tel que $T_P(n) \leq p(n)$, pour tout n . On parle aussi d'algorithmes quadratiques ($p(n) = c \cdot n^2$), cubiques, etc.
- ▶ Un algorithme P est **exponentiel** s'il existe un polynôme $p(n)$ tel que $T_P(n) \leq 2^{p(n)}$, pour tout n .

76/131

Comparaison de problèmes

Spectre large des niveaux de difficulté :

- ▶ Problèmes **faciles** (sur les graphes) :
 1. Accessibilité : étant donné un graphe et 2 sommets s, t , est-ce qu'il y a un chemin de s à t ? (solution : DFS/BFS, Dijkstra, Floyd-Warshall)
 2. Connexité : est-ce qu'un graphe donné est connexe? (solution : accessibilité)
 3. Graphes eulériens : est-ce qu'un graphe possède un circuit qui passe par chaque arc exactement une fois? (solution : connexité)
 4. 2-colorabilité : est-ce qu'on peut colorier un graphe avec 2 couleurs t.q. pour chaque arête uv , les 2 sommets u, v ont des couleurs différentes? (solution : DFS)
- ▶ **Facile** veut dire qu'on connaît des algorithmes **polynomiaux** pour résoudre ces problèmes.

77/131

Comparaison de problèmes

- ▶ Problèmes **difficiles** :
 1. SAT : savoir si une formule booléenne (sans quantificateurs) a une valuation satisfaisante (voir <http://www.dwheeler.com/essays/minisat-user-guide.html>).
 2. Graphes hamiltoniens : est-ce qu'un graphe possède un circuit qui passe par chaque sommet exactement une fois (voir <http://www.tsp.gatech.edu/index.html>).
 3. 3-colorabilité
- ▶ Problèmes encore **plus difficiles** :
 1. Certains jeux à 2 joueurs.
 2. QSAT : savoir si une formule booléenne quantifiée est valide.
 3. Savoir si l'intersection de n automates finis est non-vide.
- ▶ **Difficile** veut dire qu'on ne connaît pas d'algorithmes polynomiaux pour ces problèmes. La théorie de la **complexité** classe les problèmes selon les ressources (en temps et/ou mémoire) **nécessaires** pour les résoudre.

78/131

Littéraux, clauses, et formules CNF

Étant données des variables x_1, x_2, \dots :

- ▶ un littéral est soit une variable x_i , soit la négation d'une variable $\neg x_i$.
- ▶ Une **clause** est une disjonction de littéraux.
Exemple : $x_1 \vee \neg x_2 \vee \neg x_4 \vee x_5$.
- ▶ Une **3-clause** est une clause avec 3 littéraux différents.
Exemple : $x_1 \vee \neg x_2 \vee \neg x_4$.
- ▶ Une formule CNF est une conjonction de clauses.
- ▶ Une formule 3-CNF est une conjonction de 3-clauses.
Exemple : $(x_1 \vee \neg x_2 \vee \neg x_4) \wedge (\neg x_1 \vee x_2 \vee x_3)$.

79/131

SAT

Le problème **SAT** est le suivant :

- ▶ **Donnée** : une formule **CNF** sur des variables x_1, x_2, \dots, x_n .
- ▶ **Question** : existe-t-il une **valuation** des variables $\sigma : \{x_i \mid 1 \leq i \leq n\} \rightarrow \{tt, ff\}$ qui rend la formule vraie?

80/131

3-SAT

Le problème **3-SAT** est le suivant :

- ▶ **Donnée** : une formule **3-CNF** sur des variables x_1, x_2, \dots, x_n .
- ▶ **Question** : existe-t-il une **valuation** des variables $\sigma : \{x_i \mid 1 \leq i \leq n\} \rightarrow \{0, 1\}$ qui rend la formule vraie ?

Le problème **3-SAT** est donc moins général que **SAT**.

81/131

Réduction SAT vers 3-SAT

- ▶ À toute instance φ de SAT, on associe une instance $\tilde{\varphi}$ de 3-SAT tq.

$$\varphi \text{ est satisfaisable} \iff \tilde{\varphi} \text{ est satisfaisable.}$$

Remarque. On va construire $\tilde{\varphi}$ en temps polynomial par rapport à $|\varphi|$.

On parlera d'une **réduction polynomiale** de SAT vers 3-SAT.

82/131

Réduction SAT vers 3-SAT

Si $\varphi = c_1 \wedge \dots \wedge c_k$ où chaque c_i est une clause, on construit

$$\tilde{\varphi} = \varphi_1 \wedge \dots \wedge \varphi_k, \text{ où}$$

- ▶ Chaque φ_i est une conjonction de 3-clauses,
- ▶ φ_i utilise les variables de c_i , + éventuellement de nouvelles variables.
- ▶ Si une affectation des variables rend c_i vraie, on peut la compléter pour rendre φ_i vraie.
- ▶ Inversement, si une affectation des variables rend φ_i vraie, sa restriction aux variables de c_i rend c_i vraie.

83/131

Réduction SAT vers 3-SAT : construction de φ_i

- ▶ Si $c_i = l_1$ (un littéral), on ajoute 2 variables y_i, z_i et

$$\varphi_i = (l_1 \vee y_i \vee z_i) \wedge (l_1 \vee \neg y_i \vee z_i) \wedge (l_1 \vee y_i \vee \neg z_i) \wedge (l_1 \vee \neg y_i \vee \neg z_i).$$

- ▶ Si $c_i = l_1 \vee l_2$ (2 littéraux), on ajoute 1 variable y_i et

$$\varphi_i = (y_i \vee l_1 \vee l_2) \wedge (\neg y_i \vee l_1 \vee l_2).$$

- ▶ Si c_i est une 3-clause : $\varphi_i = c_i$.
- ▶ Si $c_i = l_1 \vee \dots \vee l_k$ avec $k \geq 4$, on ajoute $k - 3$ variables $t_{i,1}, \dots, t_{i,k-3}$

$$\varphi_i = (t_{i,1} \vee l_1 \vee l_2) \wedge (\neg t_{i,1} \vee l_3 \vee t_{i,2}) \wedge (\neg t_{i,2} \vee l_4 \vee t_{i,3}) \wedge \dots \wedge (\neg t_{i,k-3} \vee l_{k-1} \vee l_k)$$

84/131

Réduction SAT vers 3-SAT : exemple

► $\varphi = (x_1 \vee \neg x_2 \vee x_3 \vee \neg x_4 \vee x_5) \wedge (x_1 \vee \neg x_2) \wedge (\neg x_1 \vee x_2 \vee x_4)$,
alors

► La construction donne

$$\begin{aligned} \tilde{\varphi} = & (t_{1,1} \vee x_1 \vee \neg x_2) \wedge (\neg t_{1,1} \vee x_3 \vee t_{1,2}) \wedge (\neg t_{1,2} \vee \neg x_4 \vee x_5) \\ & \wedge (y_2 \vee x_1 \vee \neg x_2) \wedge (\neg y_2 \vee x_1 \vee \neg x_2) \\ & \wedge (\neg x_1 \vee x_2 \vee x_4) \end{aligned}$$

85/131

Réduction SAT vers 3-SAT

On vérifie qu'avec la construction précédente :

- Si une affectation des variables rend chaque c_i vraie, on la complète facilement pour rendre chaque φ_i vraie.
- Inversement, si une affectation des variables rend chaque φ_i vraie, la restriction de cette affectation aux variables de c_i rend c_i vraie. Donc

c_i est satisfaisable

\iff

φ_i est satisfaisable avec les mêmes valeurs pour les variables de c_i .

- Comme les variables ajoutées dans φ_i n'apparaissent que dans φ_i :

φ est satisfaisable $\iff \tilde{\varphi}$ est satisfaisable.

86/131

Réduction SAT vers 3-SAT

Récapitulatif. A partir de φ CNF, on a construit $\tilde{\varphi}$ 3-CNF telle que

φ est satisfaisable $\iff \tilde{\varphi}$ est satisfaisable.

On a donc

SAT \leq 3-SAT.

Inversement, comme 3-SAT est un problème moins général que SAT :

3-SAT \leq SAT.

87/131

3-coloration

Le problème **3-coloration** est le suivant :

- **Donnée** : un graphe non orienté G .
- **Question** : existe-t-il une 3-coloration de G ?

88/131

Réduction 3-coloration vers 3-SAT

- ▶ À toute instance $G = (V, E)$ de 3-coloration on associe une formule φ_G tq.

$$G \text{ est 3-coloriable} \iff \varphi_G \text{ est satisfaisable}$$

- ▶ Littéraux : $\{v_i \mid v \in V, i \in \{1, 2, 3\}\}$
 (les couleurs sont 1,2,3 ; v_i vrai signifiera que le sommet v est colorié par la couleur i)

- ▶ Clauses :

1. Chaque sommet est colorié par une (et une seule) couleur :

$$\bigwedge_{v \in V} ((v_1 \vee v_2 \vee v_3) \wedge \bigwedge_{i \neq j} (\overline{v_i} \vee \overline{v_j}))$$

2. Deux sommets voisins n'ont pas la même couleur :

$$\bigwedge_{uv \in E, i \in \{1,2,3\}} (\overline{u_i} \vee \overline{v_i})$$

89/131

Réduction 3-SAT vers 3-coloration

- ▶ À toute instance φ de 3-SAT, on associe une instance G_φ de 3-coloration tq.

$$\varphi \text{ est satisfaisable} \iff G_\varphi \text{ admet une 3-coloration.}$$

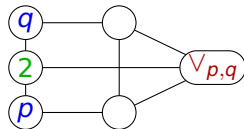
On utilise des sous-graphes (appelés *gadgets*) pour coder

- ▶ les littéraux vrais dans une évaluation qui satisfait φ ,
- ▶ les opérateurs logiques \wedge et \vee .

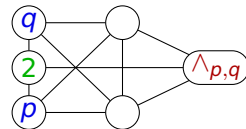
90/131

3-SAT \leq 3-coloration

- ▶ On utilise 3 sommets particuliers 0, 1, 2 reliés entre eux, qu'on peut supposer, quitte à renommer les couleurs, coloriés par 0, 1, 2.
- ▶ Pour chaque variable x_i : 2 sommets x_i et $\neg x_i$ reliés entre eux et à 2.
- ▶ Opérateurs : OU $p \vee q$ codé par :



ET $p \wedge q$ codé par :



en reliant $\vee_{p,q}$ et $\wedge_{p,q}$ au sommet 2 (p et q le sont inductivement).

- ▶ $\vee_{p,q}$ coloriable par 1 si et seulement si p OU q sont coloriés 1.
- ▶ $\wedge_{p,q}$ coloriable par 1 si et seulement si p ET q sont coloriés 1.
- ▶ Sommet « résultat » relié à 0 (et 2 par la construction précédente).

91/131

Clique et ensemble indépendant

Dans un graphe G non orienté

- ▶ Une **clique** pour G est un ensemble de sommets tous reliés 2 à 2.

Le problème **Clique** est le suivant :

- ▶ **Donnée** : un graphe G non orienté et un entier $K > 0$.
- ▶ **Question** : existe-t-il une clique de G de taille K ?

92/131

Réduction 3-SAT vers clique

- ▶ À toute instance φ de 3-SAT, on associe une instance G_φ, K_φ de **Clique** tq.
 φ est satisfaisable $\iff G_\varphi$ a une clique de taille K_φ .
et tq. on peut construire G_φ, K_φ en temps polynomial par rapport à $|\varphi|$.

93/131

- ▶ Une réduction **polynomiale** d'un problème P_1 vers un problème P_2 est une fonction de réduction de P_1 vers P_2 , qui est calculable en temps polynomial.
On note $P_1 \leq_P P_2$ s'il existe une réduction polynomiale de P_1 vers P_2 .
- ▶ Les réductions polynomiales sont **transitives** :
 $si P_1 \leq_P P_2 \text{ et } P_2 \leq_P P_3, \text{ alors } P_1 \leq_P P_3.$
- ▶ Si $P_1 \leq_P P_2$ et P_2 possède un algorithme polynomial, alors P_1 en a un, aussi. (On dit que la classe des problèmes résolubles en temps polynomial est fermée par réductions polynomiales.)

95/131

Réduction 3-SAT vers clique

- ▶ Soit $\varphi = (l_0 \vee l_1 \vee l_2) \wedge \dots \wedge (l_{3k-3} \vee l_{3k-2} \vee l_{3k-1})$.
- ▶ Le graphe G_φ a $3k$ sommets l_0, \dots, l_{3k-1} .
- ▶ Deux sommets l_i, l_j sont reliés si
 - ▶ ils ne proviennent pas de la même clause ($i/3 \neq j/3$), et si
 - ▶ ils ne sont pas de la forme $l, \neg l$.
- ▶ On choisit l'entier K_φ égal à k .
- ▶ On vérifie que G_φ a une clique de taille K_φ ssi φ est satisfaisable.

94/131

Plan

Présentation, bref historique

Ensembles dénombrables (rappels)

Modèles de calcul

Problèmes indécidables

Problèmes et réductions

La classe NP

Machines de Turing, hiérarchie de Chomsky

96/131

Classe NP

- ▶ Soit P un problème. Un **vérificateur** pour P est un algorithme V qui satisfait :
Pour toute instance x de P : x est une instance positive si et seulement si il existe un y tel que V accepte $\langle x, y \rangle$.
- ▶ Un vérificateur V pour P est dit **polynomial**,
 1. s'il existe un polynôme $p(\cdot)$ tel que l'algorithme V prend en entrée des paires $\langle x, y \rangle$ où la taille de y est bornée par $p(\text{taille}(x))$;
 2. l'algorithme V est polynomial dans la **taille de x** .
- ▶ Exemples : Un vérificateur pour **SAT** prend en entrée une formule CNF et une valuation de ses variables, et décide si la valuation satisfait la formule.

97/131

Classe NP

- ▶ Un vérificateur pour **3-coloration** prend en entrée un graphe et une coloration des sommets par $\{1, 2, 3\}$, et vérifie que deux sommets adjacents ont des couleurs différentes.
- ▶ Un vérificateur pour **chemin hamiltonien** prend en entrée un graphe $G = (V, E)$ et une permutation $v_{i_1}, v_{i_2}, \dots, v_{i_n}$ des sommets dans V , et vérifie qu'il s'agit d'un chemin dans G : $(v_{i_j}, v_{i_{j+1}}) \in E$ pour tout j .

Remarque : pour tous les problèmes ci-dessus, la difficulté n'est pas de **vérifier** une solution, mais d'en **trouver** une.

La classe **NP** est la classe des problèmes qui possèdent des vérificateurs polynomiaux.

98/131

P vs. NP

- ▶ La classe **P** contient tous les problèmes qui ont des algorithmes polynomiaux.
- ▶ Evidemment, $\mathbf{P} \subseteq \mathbf{NP}$.
- ▶ La question $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$ est **ouverte** et représente une des questions majeures de la théorie de la complexité.

Le début d'une réponse à $\mathbf{P} \stackrel{?}{=} \mathbf{NP}$ est la notion de **problème NP-complet** (Karp, 1972)



99/131

Problèmes NP-complets

- ▶ Un problème P est **NP-complet** si
 1. P appartient à **NP**, et
 2. tout problème P' appartenant à **NP** se réduit à P par une réduction **polynomiale** : $P' \leq_p P$.
On dit aussi que P est **NP-difficile**.
- ▶ On va montrer qu'il existe des problèmes **NP-complets**.
- ▶ *Attention* : Ne pas confondre les termes **NP** et **NP-complet**. Tout problème $P \in \mathbf{P}$ est aussi dans **NP**.
- ▶ Si on trouve un algorithme polynomial pour un problème **NP-complet**, alors $\mathbf{NP} \subseteq \mathbf{P}$, donc $\mathbf{P} = \mathbf{NP}$ (et on remporte un prix de 1 Mio.\$) :
Supposons que P est **NP-complet** et que $P \in \mathbf{P}$. Alors, pour tout problème $P' \in \mathbf{NP} : P' \leq_p P$ (car P est **NP-difficile**), donc $P' \in \mathbf{P}$ (car \mathbf{P} est clos par les réductions polynomiales).
- ▶ *Attention* : Un problème **NP-difficile** peut être même indécidable.

100/131

Problèmes de décision / calcul de solutions / optimisation

- ▶ Les problèmes vus jusqu'à présent (SAT, 3-coloration, etc) sont des problèmes de **décision** : réponse OUI/NON.
- ▶ Problèmes de **calcul de solution** : si une formule est satisfaisable, alors calculer une valuation satisfaisante. Si un graphe est 3-coloriable, alors calculer un coloriage à 3 couleurs...
- ▶ Problèmes d'**optimisation** : ayant un graphe pondéré sur les arcs, calculer un cycle hamiltonien de poids minimal (TSP).

101/131

Problèmes de décision / calcul de solutions / optimisation

Considérons le problème SAT et supposons qu'il ait un **algorithme polynomial A** pour le résoudre. Alors on peut construire l'algorithme suivant, qui calcule pour une formule donnée φ une valuation satisfaisante σ (s'il en existe une) :

- ▶ La formule donnée : $\varphi(x_1, \dots, x_n)$.
- ▶ Si $A(\varphi)$ retourne "non", alors retourner "non-satisfaisable".
- ▶ Pour $i = 1$ jusqu'à n faire :
 - ▶ Si $A(\varphi(b_1, \dots, b_{i-1}, 1, x_{i+1}, \dots, x_n))$ retourne "oui", alors $b_i := 1$, sinon $b_i = 0$.
- ▶ Retourner (b_1, \dots, b_n) .

102/131

Réductions de Turing

On remarque que l'algorithme précédent est en fait une **réduction** (du problème de calcul de solution au problème de décision) différente des réductions \leq vues jusqu'à maintenant. Ce type de réduction s'appelle de type **Turing (polynomiale)**. Comme les réductions \leq (appelées "many-one") celles de type Turing préservent la décidabilité, et **P** est fermée par les réductions de Turing polynomiales :

- ▶ $A \leq_T B$ signifie que A se réduit à B par une réduction de type Turing : il existe un algorithme pour A , qui utilise un algorithme pour B de façon "boîte noire".
On écrit $A \leq_T^p B$ si A se réduit à B par une réduction Turing polynomiale : l'algorithme pour A - sans compter le temps d'exécution des appels de B - est polynomial.
- ▶ Si $A \leq_T B$ et B est décidable, alors A est décidable.
- ▶ Si $A \leq_T^p B$ est $B \in \mathbf{P}$, alors $A \in \mathbf{P}$.

103/131

Théorème de Cook-Levin

- ▶ **Théorème (Cook, Levin)** SAT est **NP**-complet.



- ▶ **Conséquence.** D'après les réductions polynomiales vues précédemment, les problèmes 3-SAT, 3-COLORATION, et CLIQUE sont **NP**-complets.

104/131

Un autre problème **NP**-complet : Somme d'entiers

Le problème **Somme d'entiers** est le suivant :

- ▶ **Donnée** : des entiers $x_1, \dots, x_k > 0$ et un entier s .
- ▶ **Question** : existe-t-il $1 \leq i_1 < i_2 < \dots < i_p \leq k$ tels que

$$x_{i_1} + \dots + x_{i_p} = s.$$

C'est clairement dans **NP** : on devine i_1, \dots, i_p et on teste.

On montre que c'est **NP**-complet par une réduction 3-SAT \leq Somme d'entiers.

105/131

Partition

Le problème **Partition** est le suivant :

- ▶ **Donnée** : des entiers $x_1, \dots, x_k > 0$.
- ▶ **Question** : existe-t-il $X \subseteq \{1, \dots, k\}$ tel que

$$\sum_{i \in X} x_i = \sum_{i \notin X} x_i.$$

C'est clairement dans **NP** : on devine X et on teste.

106/131

Réduction Somme d'entiers vers Partition

Soit x_1, \dots, x_k, s une instance de **Somme d'entiers**. Soit $x = \sum x_i$.
On construit (en temps polynomial) l'instance $x_1, \dots, x_k, x - 2s$ de **Partition**.

- ▶ Si **Somme d'entiers** a une solution sur x_1, \dots, x_k, s ,
Partition a une solution sur $x_1, \dots, x_k, x - 2s$.
- ▶ Inversement, si **Partition** a une solution sur $x_1, \dots, x_k, x - 2s$,
Somme d'entiers a une solution sur x_1, \dots, x_k, s .

107/131

Réduction 3-SAT vers Somme d'entiers

- ▶ On construit à partir d'une **formule 3-CNF**
 $\varphi = C_1 \wedge C_2 \wedge \dots \wedge C_m$ avec variables x_1, \dots, x_n une instance de **Somme d'entiers** (x_1, \dots, x_k, s) t.q. φ est satisfaisable ssi (x_1, \dots, x_k, s) a une solution.
- ▶ Idée : on code les littéraux et les clauses de φ par de (très) grands entiers en base 10. Le codage est défini de telle façon que quand on fait des sommes, il n'y a pas de retenue (c-à-d, les "bits" sont décodables).
- ▶ A chaque variable x_i correspondent les 2 entiers y_i, z_i . Les entiers y_i, z_i sont de longueur $m + i$ et commencent chacun par 10^{i-1} . Les m derniers "bits" codent les clauses : pour y_i le j dernier "bit" est 1 si x_i apparaît dans C_j , et 0 sinon ; pour z_i le j dernier "bit" est 1 si \bar{x}_i apparaît dans C_j , et 0 sinon.

108/131

Réduction 3-SAT vers Somme d'entiers

- ▶ Exemple : Pour $\varphi = (x_1 \vee \overline{x_3}) \wedge (\overline{x_1} \vee x_2 \vee x_3)$ on a $y_1 = 110$, $z_1 = 101$, $y_2 = 1001$, $z_2 = 1000$, etc.
- ▶ A chaque clause C_j correspondent les 2 entiers $t_j = w_j = 10^{m-j}$.
- ▶ L'entier $s = \underbrace{1 \cdots 1}_n \underbrace{3 \cdots 3}_m$.
- ▶ Pour produire le bloc 1^n dans s il faut choisir exactement un de y_i, z_i (pour tout $1 \leq i \leq n$). Ceci revient à choisir une valuation des variables – y_i signifie x_i vrai, et z_i signifie x_i faux.
- ▶ Pour produire le bloc 3^m dans s il faut que pour chaque clause, au moins un des littéraux soit vrai. On complète jusqu'à 3 en utilisant les entiers t_j, w_j .
- ▶ L'instance de "Somme d'entiers" peut se calculer en temps polynomial.

109/131

Théorème Cook-Levin : idée de preuve (1/4)

- ▶ On part d'un problème **NP**-complet P , et d'une instance I de P .
- ▶ P étant **NP**-complet, il a un vérificateur polynomial V . Plus précisément, il existe des polynômes $p, t : \mathbb{N} \rightarrow \mathbb{N}$ tels que :
 1. V prend en entrée des paires $\langle u, v \rangle$ où la taille de v est bornée par $p(\text{taille}(u))$;
 2. le temps de calcul de V est borné par $t(\text{taille}(u))$.
- ▶ Pour la preuve, on va considérer que V est un programme WHILE, qui prend en entrée deux entiers u, v , stockés dans les variables x_1 et x_2 . La **taille** d'un entier est le nombre de bits dans la représentation binaire. Le **temps de calcul** de V est égal aux nombres d'instructions élémentaires (qui sont : $x := y \pm c$, $x := c$, $x > 0$).

110/131

Théorème Cook-Levin : idée de preuve (2/4)

- ▶ Supposons que V utilise ℓ variables $x_0 = \text{res}, x_1, x_2, \dots, x_{\ell-1}$ et a k instructions, la dernière étant

$$\text{res} := 1$$

Au début, $x_1 = u, x_2 = v$ et $x_i = 0$ pour tout $i \neq 1, 2$.

- ▶ **Remarque** : les valeurs des variables après un temps de calcul t augmentent au plus par le facteur $c_{max} \cdot 2^t$ (c_{max} étant la plus grande constante du programme). Donc, les tailles des nombres stockés dans les variables seront bornées à la fin par $N := p(\text{taille}(u)) + \text{taille}(u) + t(\text{taille}(u)) + \log(c_{max})$.
- ▶ On construira une formule booléenne $\varphi(V, u, v)$ qui est satisfaisable si et seulement si V termine avec $\text{res} = 1$ sur l'entrée $x_1 = u, x_2 = v$ après exactement $M = t(\text{taille}(u))$ pas de calcul.

111/131

Théorème Cook-Levin : idée de preuve (3/4)

- ▶ Les variables de $\varphi(V, u, v)$ sont de deux types :
 1. $L_{i,r}$ pour $0 \leq i \leq k, 0 \leq r < M$. La variable $L_{i,r}$ sera vraie ssi à l'instant r , l'instruction à exécuter est la i -ième instruction de V .
 2. $B_{i,j,r}$ pour $0 \leq i \leq \ell, 0 \leq j < N, 0 \leq r < M$. La variable $B_{i,j,r}$ sera vraie ssi à l'instant r , le bit j de la valeur de la variable x_i de V , est 1.
- ▶ Les **clauses** de $\varphi(V, u, v)$ expriment plusieurs contraintes :
 1. Pour chaque r , il y a un seul i tel que $L_{i,r}$ est vrai.
 2. Début : les variables $B_{1,*,0}$ représentent le nombre u , et les variables $B_{2,*,0}$ le nombre v . Pour tout $i \neq 1, 2$, les variables $B_{i,*,0}$ sont toutes fausses (représentant 0). Et $L_{1,0}$ est vraie.
 3. Fin : $L_{k,M-1}$ est vraie (V termine avec l'instruction $\text{res} = 1$).
 4. Pour chaque $r < M$, les valeurs des $L_{*,r}$ et $B_{*,*,r}$ sont déterminées par les valeurs des $L_{*,r-1}$ et $B_{*,*,r-1}$.

112/131

Théorème Cook-Levin : idée de preuve (4/4)

- ▶ Remarque : si on enlève la contrainte " $L_{k,M-1}$ est vraie", la formule $\varphi(V, u)$ est toujours satisfaisable avec une unique valuation satisfaisante.
- ▶ Pour conclure, on définit une formule $\varphi(V, u)$ qui est satisfaisable si et seulement si u est instance positive de P , en enlevant la contrainte "les variables $B_{2,*,0}$ représentent le nombre v ".
- ▶ $\varphi(V, u)$ se calcule en temps polynomial, ce qui donne une réduction du problème P à SAT.

113/131

Retour aux modèles de calcul : grammaires et réécriture

Rappels :

- ▶ Pour un ensemble fini X (alphabet) on note par X^* l'ensemble des suites finies sur X , appelés aussi **mots**.
- ▶ Sur X^* on a une opération binaires et associative : la **concaténation**. On note uv la concaténation des mots $u, v \in X^*$, et ϵ le mot vide.

Une grammaire est une tuplet $G = \langle V, \Sigma, \mathcal{P}, S_0 \rangle$, où :

- ▶ V est un ensemble fini de **variables** (ou non-terminaux),
- ▶ Σ est un alphabet (fini), $\Sigma \cap V = \emptyset$; soit $\Gamma = V \cup \Sigma$,
- ▶ $\mathcal{P} \subseteq \Gamma^* V \Gamma^* \times \Gamma^*$ est un ensemble fini de **règles**,
- ▶ $S_0 \in V$ est l'**axiome**.

115/131

Plan

Présentation, bref historique

Ensembles dénombrables (rappels)

Modèles de calcul

Problèmes indécidables

Problèmes et réductions

La classe NP

Machines de Turing, hiérarchie de Chomsky

114/131

Grammaires

En partant de l'axiome on peut générer des mots sur l'alphabet $\Gamma = V \cup \Sigma$ à l'aide des règles :

- ▶ **Pas de réécriture** : si $u, v \in \Gamma^*$ tels que

$$u = xly, \quad v = xry$$

pour $(l, r) \in \mathcal{P}$ et $x, y \in \Gamma^*$ on écrit :

$$u \rightarrow_{\mathcal{P}} v.$$

- ▶ On écrit $u \rightarrow_{\mathcal{P}}^* v$ s'il existe $n \geq 0$ et des mots $w_0 = u, w_1, \dots, w_n = v$ tels que $w_i \rightarrow_{\mathcal{P}} w_{i+1}$ pour tout $0 \leq i < n$.

On écrit $L(G)$ pour l'ensemble des mots terminaux générés par G , appelé aussi **langage** de G :

$$L(G) = \{w \in \Sigma^* \mid S_0 \rightarrow_{\mathcal{P}}^* w\}$$

116/131

Hiérarchie de Chomsky

Chomsky définit dans les années '60 une hiérarchie de grammaires, selon la forme des règles $\mathcal{P} \subseteq \Gamma^*V\Gamma^* \times \Gamma^*$:



- ▶ type 0 : pas de restriction
- ▶ type 1 : règles de la forme $xAy \rightarrow xuy$, où $x, y \in \Gamma^*$, $A \in V$, $u \in \Gamma^+$ (grammaires contextuelles)
- ▶ type 2 : règles de la forme $A \rightarrow u$, où $A \in V$, $u \in \Gamma^*$ (grammaires non-contextuelles / algébriques)
- ▶ 3 : règles de la forme $A \rightarrow Ba$ ou $A \rightarrow a$, où $A \in V$, $a \in \Sigma \cup \{\epsilon\}$ (grammaires régulières)

117/131

Hiérarchie de Chomsky

Concernant les langages générés, chaque niveau de la hiérarchie de Chomsky correspond à un type d'automate :

type	automates	langages
0	machines de Turing	semi-décidables
1	automates linéairement bornés	NSPACE(n)
2	automates à pile	hors-contexte
3	automates finis	réguliers

118/131

Machines de Turing : modèle abstrait d'un ordinateur

Turing (1936) propose un modèle de calcul simple et élégant : la **machine de Turing**.



- ▶ Le nombre d'états d'une machine de Turing est **fini** (par comparaison, un ordinateur a un nombre fini de registres et les programmes sont finis).
- ▶ La bande représente la mémoire de la machine. Elle est **infinie** : sur un ordinateur, on peut ajouter des périphériques mémoire (disques. . .) de façon quasi-illimitée.
- ▶ L'accès à la mémoire est **séquentiel** : la machine peut bouger sa tête à droite ou à gauche d'une case à chaque étape.

119/131

Machines de Turing : formalisation

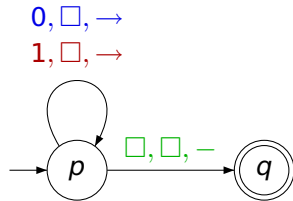
Une **machine de Turing** (MT) à une bande $M = (Q, q_0, F, \Sigma, \Gamma, \delta)$ est donnée par

- ▶ Q : ensemble **fini** d'états.
- ▶ q_0 : état initial.
- ▶ $F \subseteq Q$: ensemble d'états finaux (ou acceptants).
- ▶ Γ : alphabet fini de la bande, avec $\square \in \Gamma$.
- ▶ Σ : alphabet d'entrée, avec $\Sigma \subseteq \Gamma \setminus \{\square\}$.
- ▶ δ : ensemble de transitions. Une transition est de la forme (p, a, q, b, d) , notée $p \xrightarrow{a,b,d} q$, avec
 - ▶ $p, q \in Q$,
 - ▶ $a, b \in \Gamma$,
 - ▶ $d \in \{\leftarrow, \rightarrow\}$.
- ▶ On supposera qu'aucune transition ne part d'un état de F .

120/131

Machines de Turing : représentation graphique

- ▶ On représente souvent une MT comme un automate.
- ▶ Seules changent les étiquettes des transitions.
- ▶ Exemple, avec $\Gamma = \{0, 1, \square\}$ et $\Sigma = \{0, 1\}$:



représente la machine avec $Q = \{p, q\}$, $q_0 = p$, $F = \{q\}$,
 $\delta = \{(p, 0, \square, \rightarrow, p), (p, 1, \square, \rightarrow, p), (p, \square, \square, -, q)\}$

121/131

Fonctionnement d'une MT

- ▶ Initialement, un mot w est écrit sur la bande entouré de \square .
- ▶ Un calcul d'une MT sur w est une suite de **pas de calcul**.
- ▶ Cette suite peut être **finie** ou **infinie**.
- ▶ Le calcul commence
 - ▶ avec la tête de lecture-écriture sur la première lettre de w ,
 - ▶ dans l'état q_0 .
- ▶ Chaque pas de calcul consiste à appliquer une transition, si possible (il peut y avoir des choix : **non-déterminisme**).
- ▶ Le calcul ne s'arrête que si aucune transition n'est applicable.

122/131

Fonctionnement d'une MT

- ▶ Chaque pas consiste à appliquer une transition.
- ▶ Une transition de la forme $p \xrightarrow{a,b,d} q$ est possible seulement si
 1. la machine se trouve dans l'état p , et
 2. la lettre se trouvant sous la tête de lecture-écriture est a .
- ▶ Dans ce cas, l'application de la transition consiste à
 - ▶ changer l'état de contrôle qui devient q ,
 - ▶ remplacer le contenu de la case sous la tête de lecture-écriture par b ,
 - ▶ bouger la tête d'une case à gauche si $d = \leftarrow$, ou
 - ▶ bouger la tête d'une case à droite si $d = \rightarrow$, ou
 - ▶ ne pas bouger la tête si $d = -$.

123/131

Configurations et calculs

- ▶ Une **configuration** représente un instantané du calcul.
- ▶ La configuration uqv signifie que
 - ▶ L'état de contrôle est q
 - ▶ Le mot écrit sur la bande est uv , entouré par des \square ,
 - ▶ La tête de lecture est sur la première lettre de v .
- ▶ La configuration initiale sur w est donc q_0w .
- ▶ Pour 2 configurations C, C' , on écrit $C \vdash C'$ lorsqu'on obtient C' par application d'une transition à partir de C .

Un **calcul** d'une machine de Turing est une suite de configurations.

$$C_0 \vdash C_1 \vdash C_2 \vdash \dots$$

124/131

Calculs acceptants

Un calcul d'une machine de Turing est une suite de configurations.

$$C_0 \vdash C_1 \vdash C_2 \vdash \dots$$

3 cas possibles

- ▶ Le calcul est infini,
- ▶ Le calcul s'arrête sur un état final (de F),
- ▶ Le calcul s'arrête sur un état non final (pas de F).

125/131

Langages acceptés

On peut utiliser une machine pour **accepter** des mots.

- ▶ Le langage $\mathcal{L}(M) \subseteq \Sigma^*$ des mots acceptés par une MT M est l'ensemble des mots w sur lesquels **il existe** un calcul **fini**

$$C_0 \vdash C_1 \vdash C_2 \vdash \dots \vdash C_n$$

avec $C_0 = q_0 w$ (w est le mot **d'entrée**) et $C_n \in \Gamma^* F \Gamma^*$.

- ▶ 3 cas exclusifs : un calcul peut
 - ▶ soit s'arrêter sur un état acceptant,
 - ▶ soit s'arrêter sur un état non acceptant,
 - ▶ soit ne pas s'arrêter.
- ▶ On dit qu'une machine est **déterministe** si, pour tout $(p, a) \in Q \times \Gamma$, il existe **au plus** une transition de la forme $p \xrightarrow{a,b,d} q$.
- ▶ Si M est déterministe, elle n'admet qu'un calcul par entrée.

126/131

Grammaires de type 0 et machines de Turing

On associe une grammaire G_M de type 0 à une MT $M = (Q, q_0, F, \Sigma, \Gamma, \delta)$. On suppose que quand M atteint un état final, la bande est vide (c-à-d, le contenu est dans \square^*). La grammaire $G_M = (V, \Sigma \cup \{\#\}, \mathcal{P}, S_0)$ est définie par $V = \{S_0\} \cup (\Gamma \setminus \Sigma) \cup Q$ et l'ensemble suivant de règles :

- ▶ $S_0 \rightarrow \#f\square\#$, pour tout $f \in F$,
- ▶ A chaque transition $(p, A, q, B, d) \in \delta$ de M on associe des règles (avec $C \in \Gamma$) :

$(p, A, q, B, \rightarrow)$	$Bq \rightarrow pA$ et $Bq\square\# \rightarrow pA\#$,
$(p, A, q, B, -)$	$qB \rightarrow pA$,
(p, A, q, B, \leftarrow)	$qCB \rightarrow CpA$ et $\#q\square B \rightarrow \#pA$.
- ▶ $\#q_0 \rightarrow \epsilon$

On obtient $L(G) = \{w\# \mid M \text{ accepte } w\}$.

127/131

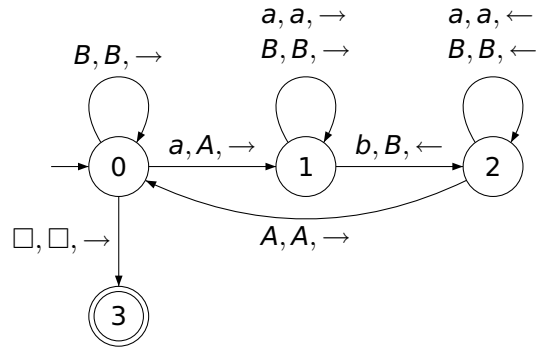
Exemples de machines de Turing

- ▶ Machine qui effectue `while(true)` ;
- ▶ Machine qui efface son entrée et s'arrête.
- ▶ Machine qui accepte 0^*1^* (**automate fini**).
- ▶ Machine qui accepte $\{a^n b^n \mid n \geq 0\}$ (**automate à pile**).
- ▶ Machine qui accepte $\{a^{2^n} \mid n \geq 0\}$ (**automate linéairement borné**).
- ▶ Machine qui accepte $\{a^n b^n c^n \mid n \geq 0\}$ (**automate linéairement borné**).
- ▶ Machine qui accepte $\{ww \mid w \in \{0, 1\}^*\}$ (**automate linéairement borné**).

128/131

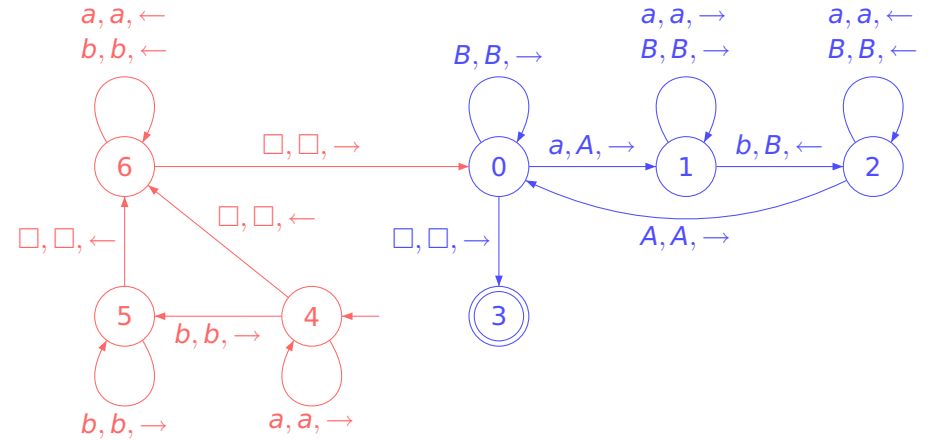
MT acceptant $(\{a^n b^n \mid n \geq 0\})^*$

Idée : marquer le 1^{er} a et le 1^{er} b, et recommencer.



MT acceptant $\{a^n b^n \mid n \geq 0\}$

Idée : idem en vérifiant qu'on est dans a^*b^* .



MT acceptant $\{a^{2^n} \mid n \geq 0\}$

Idée : marquer un a sur 2.

