Complexité et calculabilité

Anca Muscholl, LaBRI

Année 2018/19 [2]

Master 1, Département ST, Université Bordeaux
http://www.labri.fr/perso/anca/MC.html

5 septembre 2018

2. Version issue du polycopié de Marc Zeitoun.

Modalités du cours

- 12 cours, 5 groupes de TD (débutent la semaine prochaine).
- Louis-Marie Dando, Philippe Duchon, Lamine Lamali, Frédéric Mazoit, Vincent Penelle.
- Contrôle continu (CC) obligatoire (sauf dispense). DS + DM sur machine
- DS : mercredi 24/10/18, 10h15-12h15.
- Note finale session 1 :

  1/2 Examen (3h) + 1/2 CC.

- Note finale session 2 :

  max(Examen,   1/2 Examen (2h) + 1/2 CC).

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).

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.

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.

Plan du cours

Plan

Présentation, bref historique

Complexité

- Problèmes et réductions
- **P** vs. **NP**

Calculabilité

- Ensembles dénombrables (rappels)
- Programme WHILE, machines de Turing
- Décidabilité

Questions abordées dans ce cours

- Qu'est-ce que c'est un problème, un algorithme . . . ?
- Calculabilité : comment on formalise la notion de calcul ? quels problèmes peut-on résoudre (algorithmiquement) avec un ordinateur ?
- Complexité : Comment comparer la complexité de deux problèmes ?
  Y a-t-il des problèmes plus difficiles que d'autres ? Peut-on parler d'algorithmes optimaux ?

Histoire brève de la calculabilité

Wilhelm Schickard (1592 – 1635), professeur à l'Université de Tübingen (Allemagne), invente la première machine à calculer (mécanique).

Blaise Pascal (1623 – 1662), mathématicien et philosophe, construit à l'age de 19 ans la *Pascaline*, première machine à calculer opérationnelle du XVIIè siècle.

Gottfried Wilhelm Leibniz (1646 – 1716), mathématicien et philosophe, développe aussi une machine à calculer. Il préconise des idées très modernes : la machine de calcul universelle, le schéma "entrée-calcul-sortie", la base 2 pour la représentation des nombres.

Histoire brève de la calculabilité

Le métier à tisser de Joseph Marie Jacquard (1752 – 1834) est basé sur l'utilisation de cartes perforées, et est à l'origine des premiers programmes de calcul.

Charles Babbage (1791 – 1871), professeur à Cambridge, construit la machine différentielle et la machine analytique. La dernière peut être considérée comme précurseur des ordinateurs modernes, consistant d'une unité de contrôle, une unité de calcul, une mémoire, ainsi que l'entrée-sortie.

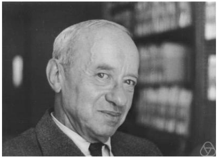
Histoire brève de la calculabilité

Ada Lovelace (1815 – 1852) travaille avec Babbage et préconise l'utilisation de la machine analytique pour la résolution de problèmes mathématiques. Elle est considérée comme premier programmeur du monde.

David Hilbert (1862 – 1943), professeur à Göttingen, présente en 1920 un programme de recherche visant à clarifier les fondaments des mathématiques : "tout enoncé mathématique peut être soit prouvé ou refuté". Plus tard il enonce le "Entscheidungsproblem" : montrer de façon "méchanique" si un enoncé mathématique est vrai ou faux.

Histoire brève de la calculabilité

Kurt Gödel (1906 – 1978), un des logiciens les plus fameux de l'histoire, répond 1931 négativement quand au programme proposé par Hilbert, en montrant que tout système formel suffisamment puissant est soit incomplet ou incohérent. Il montre ceci en construisant une formule qui exprime le fait qu'elle n'est pas démontrable ("codage de Gödel", "diagonalisation").



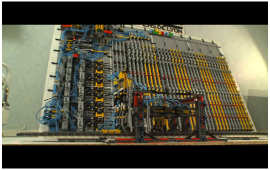
Alfred Tarski (1901 – 1983), autre logicien très connu, axiomatise la géométrie euclidienne et montre la décidabilité de la théorie du premier ordre des réels en 1931.

Histoire brève de la calculabilité

Alan Turing (1912 – 1954) et Alonzo Church (1903 – 1995) montrent indépendamment, en 1936, l'indécidabilité de l'Entscheidungsproblem. Turing propose la machine de Turing comme modèle formel de calcul, et Church le lambda-calcul. Ils enoncent le principe selon lequel tout ce qui est calculable peut être calculé sur un de ces deux modèles (“thèse de Church-Turing”).

Histoire brève de la calculabilité

2012 a commémoré le centenaire de la naissance de Turing. Une réalisation en LEGO de la machine de Turing : http ://videotheque.cnrs.fr/doc=3001



- Emil Post (1897 – 1954) invente en 1936 un modèle de calcul proche de la machine de Turing.
- Steven Kleene (1909 – 1994) montre en 1938 l'équivalence entre les machines de Turing, le λ-calcul, et les fonctions récursives.
- Emil Post et Andrei A. Markov (1903 – 1979) prouvent qu'un problème posé par Axel Thue en 1914 n'est pas résoluble algorithmiquement (problème du mot pour les semigroupes).
- Yuri Matiyasevich (1947 –) montre qu'il n'y a pas d'algorithme qui résout le $10^{\text{ème}}$ problème de Hilbert ("peut-on trouver une solution entière pour une équation diophantienne donnée ?").

Origines de la théorie de la complexité

La théorie de la complexité débute par la fameuse question **P** $=_?$ **NP**, qui est une des questions ouvertes majeures de l'informatique. La question **P** $=_?$ **NP** fait partie des 7 “Clay Millennium Problems”, et pour leur solution il y a un prix de 1 Mio.$.
Un seul de ces problèmes a été résolu (la conjecture de Poincaré).

Que signifie la question **P** $=_?$ **NP** ?

En gros, on se limite à des problèmes qui possèdent un espace *exponentiel* de solutions potentielles et on se demande s'il existe un algorithme *polynomial* qui permet de trouver une solution (s'il en existe une).

De nombreux problèmes pratiques sont caracterisés par un nombre *exponentiel* de solutions potentielles : des puzzles, des problèmes d'ordonnancement, de cryptographie, etc.

Origines de la théorie de la complexité

John Nash (1928-2015), connu pour ses contributions en théorie des jeux économiques, a reconnu l'importance de la question **P** $=_?$ **NP** pour la cryptographie dès 1955 :

> *Now my general conjecture is as follows : for almost all sufficiently complex types of enciphering, especially where the instructions given by different portions of the key interact complexly with each other in the determination of their ultimate effects on the enciphering, the mean key computation length increases exponentially with the length of the key, or in other words, the information content of the key ... The nature of this conjecture is such that I cannot prove it, even for a special type of ciphers. Nor do I expect it to be proven.*

Origines de la théorie de la complexité

En 1956, Kurt Gödel écrit à John von Neumann en posant la question **P** $=_?$ **NP** pour la première fois :

> *If there actually were a machine with [running time] $\sim Kn$ (or even only with $\sim Kn^2$) [for some constant K independent of n], this would have consequences of the greatest magnitude. That is to say, it would clearly indicate that, despite the unsolvability of the Entscheidungsproblem, the mental effort of the mathematician in the case of yes-or-no questions could be completely [added in a footnote : apart from the postulation of axioms] replaced by machines. One would indeed have to simply select an n so large that, if the machine yields no result, there would then also be no reason to think further about the problem.*

Plan

Vocabulaire

Avant d'introduire formellement la question **P** $=_?$ **NP** on va se familiariser avec les notions centrales de *problème* et de *réduction* entre problèmes.

1. Un problème de décision A est
   - une question portant sur un ensemble de données (= entrées)
   - dont la réponse est OUI ou NON.

   Rq : Ne pas confondre problème et algorithme le résolvant.
2. Une instance du problème A est la question posée sur une donnée/entrée particulière de A.

Exemple:

- Problème : Savoir si un graphe non-orienté est connexe.
- Instance : $V = \{1, 2, 3, 4, 5\}$, $E = \{(1, 2), (2, 3), (3, 1), (4, 5)\}$
- Algorithme : Depth-first-search.

On s'intéresse aussi aux problèmes calculatoires, dont la réponse n'est pas nécessairement binaire (OUI/NON).

Exemple:

- Calculer les composantes connexes d'un graphe non-orienté.
- Calculer les facteurs premiers d'un entier.

On s'intéresse aussi aux problèmes d'optimisation.

Exemple:

- Calculer un cycle de longueur minimale dans un graphe.
- Calculer un cycle de longueur maximale et sans sommet répété dans un graphe.
- Calculer le plus petit facteur premier d'un entier.

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 apparait-il dans s ?

Problème 4
- D. Un programme C.
- Q. Le programme est-il syntaxiquement correct ?

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://www.mathpuzzle.com/eternity.html.
- Q. Le puzzle a-t-il une solution ?

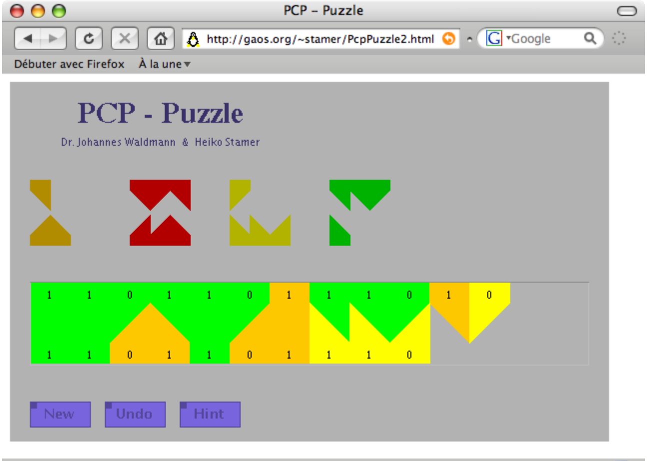
Problème 7
- D. Un programme C.
- Q. Le programme s'arrête-t-il sur l'entrée donnée ?

Problème 8
- D. Un programme C.
- Q. Le programme s'arrête-t-il toujours ?

Problème 9
- D. Des couples de mots $(u_1, v_1), \ldots, (u_n, v_n)$.
- Q. Existe-t-il des entiers $i_i, \ldots, i_k$ tels que $u_{i_1} \ldots u_{i_k} = v_{i_1} \ldots v_{i_k}$?

Problème 9 : PCP, Problème de correspondance de Post

http://webdocs.cs.ualberta.ca/~games/PCP/



Temps de calcul

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

- $t_P(I)$ désigne le temps de calcul de P sur I. Le temps de calcul est le nombre d'instructions élémentaires exécutées (par exemple, instructions arithmétiques, comparaisons, affectations de valeur, etc).
- La fonction $T_P : \mathbb{N} \to \mathbb{N}$ est définie par :

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

  Il s'agit d'un temps de calcul au pire des cas.
- Un algorithme P est polynomial s'il existe un polynome $p(n)$ tel que $T_P(n) \leq p(n)$, pour tout n. On parle par exemple d'algorithmes quadratiques ($p(n) = c \cdot n^2$), cubiques, etc.
- Un algorithme P est exponentiel s'il existe un polynome $p(n)$ tel que $T_P(n) \leq 2^{p(n)}$, pour tout n.

Problèmes et leur complexité

- 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)

Rq : Facile veut dire qu'on connaît des algorithmes polynomiaux pour résoudre ces problèmes.

Problèmes et leur complexité

- 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 formules booléene quantifiée est valide.
  3. Savoir si l'intersection de n automates finis est non-vide.
- Problèmes qui ne sont même pas résolubles :
  1. Terminaison de programme
  2. PCP
  3. Eternity

Comment comparer 2 problèmes ?

La notion essentielle de *réduction* permet de comparer deux problèmes (de décision A, B) : de manière informelle : le problème A est plus facile que le problème B si l'on peut se servir d'un algorithme pour le problème B afin de résoudre le problème A.

Logique propositionnelle

Étant données des variables booléennes $x_1, x_2, \ldots$:

- un littéral est soit une variable x_i, soit la négation d'une variable $\neg x_i$ (on écrit aussi $\overline{x_i}$).
- 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 : $(x_1 \vee \neg x_2 \vee \neg x_4) \ \wedge \ (\neg x_1 \vee x_2 \vee x_3)$.

SAT

Le problème SAT est le suivant :

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

$1 =$ vrai, $0 =$ faux

3-SAT

Le problème 3-SAT est le suivant :

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

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

On peut bien-sûr utiliser un algorithme pour SAT afin de résoudre 3-SAT, mais est-ce que l'inverse est possible aussi ?

Réduction SAT vers 3-SAT

- À toute instance φ de SAT, on va associer une instance $\widetilde{\varphi}$ de 3-SAT tq.

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

Remarque.

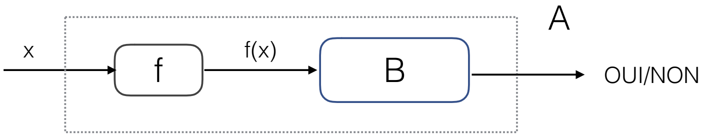
1. On va construire $\widetilde{\varphi}$ à partir de φ en temps polynomial par rapport à la taille $|\varphi|$ de φ.
   On parlera d'une réduction polynomiale de SAT vers 3-SAT.
2. La propriété ci-dessus garantit qu'on peut se servir d'un algorithme P pour 3-SAT afin de résoudre SAT : pour toute formule CNF φ on construit d'abord $\widetilde{\varphi}$ et on lance ensuite P sur $\widetilde{\varphi}$. Soit P' l'algorithme ainsi construit.
   Important : si P est un algorithme polynomial pour 3-SAT, alors P' est aussi polynomial.

Réductions

- Soient A et B deux problèmes.
- On note $X \subseteq D$ l'ensemble des instances positives de A, et $Y \subseteq D'$ l'ensemble des instances positives de B.
- Une réduction de A vers B est une fonction calculable $f : D \to D'$ telle que

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

- On note $A \leqslant B$ (lit : A se réduit à B)
- L'existence d'une réduction de A vers B assure qu'on on peut se servir d'un algorithme pour le problème B pour résoudre A :



Remarques

- "A se réduit à B" ne signifie PAS que B est plus facile que A. Plutôt : la recherche d'une solution pour A sur une instance x donnée peut être ramenée à la recherche d'une solution pour B sur l'instance $f(x)$.
- La notion de réduction est symétrique : x est une instance positive de A SI ET SEULEMENT SI $f(x)$ est une instance positive de B.
  Mais : $A \leq B$ n'implique pas $B \leq A$.
- Les réductions (polynomiales) sont transitives :
  - si $A_1 \leq A_2$ et $A_2 \leq A_3$, alors $A_1 \leq A_3$;
  - si $A_1 \leq_P A_2$ et $A_2 \leq_P A_3$, alors $A_1 \leq_P A_3$

Réduction SAT vers 3-SAT

Si $\varphi = c_1 \wedge \cdots \wedge c_k$ où chaque c_i est une clause, on construit $\widetilde{\varphi} = \varphi_1 \wedge \cdots \wedge \varphi_k$, 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.

Réduction SAT vers 3-SAT : construction de φ_i

- Si $c_i = \ell_1$ (un littéral), on ajoute 2 nouvelles variables y_i, z_i et

$$\varphi_i = (\ell_1 \vee y_i \vee z_i) \wedge (\ell_1 \vee \neg y_i \vee z_i) \wedge (\ell_1 \vee y_i \vee \neg z_i) \wedge (\ell_1 \vee \neg y_i \vee \neg z_i).$$

- Si $c_i = \ell_1 \vee \ell_2$ (2 littéraux), on ajoute 1 nouvelle variable y_i et

$$\varphi_i = (y_i \vee \ell_1 \vee \ell_2) \wedge (\neg y_i \vee \ell_1 \vee \ell_2).$$

- Si c_i est une 3-clause : $\varphi_i = c_i$.
- Si $c_i = \ell_1 \vee \cdots \vee \ell_k$ avec $k \geqslant 4$, on ajoute $k-3$ nouvelles variables $t_{i,1}, \ldots, t_{i,k-3}$ et

$$\begin{aligned}\varphi_i &= (t_{i,1} \vee \ell_1 \vee \ell_2) \wedge (\neg t_{i,1} \vee \ell_3 \vee t_{i,2}) \wedge (\neg t_{i,2} \vee \ell_4 \vee t_{i,3}) \wedge \cdots \\ &\quad \wedge (\neg t_{i,k-3} \vee \ell_{k-1} \vee \ell_k)\end{aligned}$$

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

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

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

$$c_i \text{ est satisfaisable} \iff \varphi_i \text{ est satisfaisable avec les mêmes valeurs pour les variables de } c_i.$$

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

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

Réduction SAT vers 3-SAT

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

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

On a donc

$$\text{SAT} \leqslant \text{3-SAT et la réduction est polynomiale}$$

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

$$\text{3-SAT} \leqslant \text{SAT.}$$

3-coloration

Le problème 3-coloration est le suivant :

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

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 :

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

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

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

Réduction 3-SAT vers 3-coloration

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

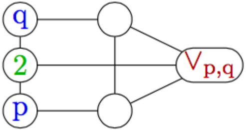
$$\varphi \text{ est satisfaisable} \iff G_\varphi \text{ admet une 3-coloration.}$$

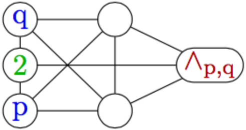
On utilise des sous-graphes (appelés *gadgets*) pour coder

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

3-SAT $\leqslant$ 3-coloration

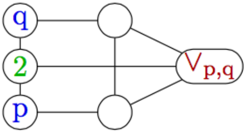
- On utilise 3 sommets particuliers 0, 1, 2 reliés entre eux, qu'on peut supposer, quitte à renommer les couleurs, coloriés par 0, 1, 2.
- Pour chaque variable x_i : 2 sommets x_i et $\neg x_i$ reliés entre eux et à 2.
- Opérateurs : OU $p \vee q$ codé par : ET $p \wedge q$ codé par :

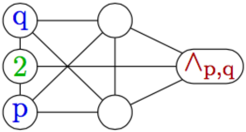




en reliant $\vee_{p,q}$ et $\wedge_{p,q}$ au sommet 2 (p et q le sont inductivement).

3-SAT $\leqslant$ 3-coloration





- $\vee_{p,q}$ 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).

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 ?

Réduction 3-SAT vers clique

- À toute instance φ de 3-SAT, on associe une instance G_φ, K_φ de Clique tq.

$$\varphi \text{ est satisfaisable} \iff G_\varphi \text{ a une clique de taille } K_\varphi.$$

et tq. on peut construire G_φ, K_φ en temps polynomial par rapport à $|\varphi|$.

Réduction 3-SAT vers clique

- Soit $\varphi = (\ell_0 \vee \ell_1 \vee \ell_2) \wedge \cdots \wedge (\ell_{3k-3} \vee \ell_{3k-2} \vee \ell_{3k-1})$.
- Le graphe G_φ a $3k$ sommets $\ell_0, \ldots, \ell_{3k-1}$.
- 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 $\ell, \neg\ell$.
- 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.

- Une réduction polynomiale d'un problème A_1 vers un problème A_2 est une fonction de réduction de A_1 vers A_2, qui est calculable en temps polynomial.
  On note $A_1 \leq_p A_2$ s'il existe une réduction polynomiale de A_1 vers A_2.
- Les réductions polynomiales sont transitives :

  si $A_1 \leq_p A_2$ et $A_2 \leq_p A_3$, alors $A_1 \leq_p A_3$.

- Si $A_1 \leq_p A_2$ et A_2 possède un algorithme polynomial, alors A_1 en a un, aussi. (On dit que la classe des problèmes résolubles en temps polynomial est fermée par réductions polynomiales.)

Classe **NP**

La classe **NP** est une classe de problèmes : informellement, un problème appartient à **NP** si

- on peut vérifier une solution donnée en temps polynomial
- s'il existe une solution, alors il en existe toujours une de taille polynomielle

Pourquoi les problèmes de la classe **NP** sont-ils difficiles ? Parce que le nombre de solutions est potentiellement exponentiel. Donc la recherche naïve peut demander un temps exponentiel.

Classe **NP** - définition

- Soit A un problème. Un vérificateur pour A est un algorithme V qui prend en entrée des paires $\langle x, y\rangle$, où x est une instance de A, et vérifie que y est bien une “preuve” pour le fait que x est instance positive.
- Le problème A possède un vérificateur polynomial V si :
  1. il existe un polynome $p(\cdot)$ tel que pour toute instance x de A : x est une instance positive si et seulement si il existe un y de **taille au plus** $p(\textbf{taille}(x))$ et tel que V accepte $\langle x, y\rangle$;
  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 rend la formule vraie.

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 $v_{i_1}, v_{i_2}, \ldots, v_{i_n}$ 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.

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 l'informatique.

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

Problèmes **NP**-complets

- Un problème A est **NP**-complet si
  1. A appartient à **NP**, et
  2. tout problème A' appartenant à **NP** se réduit à A par une réduction polynomiale : $A' \leqslant_P A$.
     On dit aussi que A 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 $A \in \mathbf{P}$ est aussi dans **NP**.
- Si on trouve un algorithme polynomial pour un problème **NP**-complet, alors $\mathbf{NP} \subseteq \mathbf{P}$, donc $\mathbf{P} = \mathbf{NP}$:
  Supposons que A est **NP**-complet et que $A \in \mathbf{P}$. Alors, pour tout problème $A' \in \mathbf{NP}$: $A' \leq_p A$ (car A est **NP**-difficile), donc $A' \in \mathbf{P}$ (car **P** est clos par les réductions polynomiales).
- *Attention :* Un problème **NP**-difficile peut être même indécidable.

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).

Problèmes de décision / calcul de solutions / optimisation

Considérons le problème SAT et supposons qu'il ait un algorithme polynomial P 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 : $\varphi(x_1, \ldots, x_n)$.
- Si $P(\varphi)$ retourne "non", alors retourner "non-satisfaisable".
- Pour $i = 1$ jusqu'à n faire :
  - Si $P(\varphi(b_1, \ldots, b_{i-1}, 1, x_{i+1}, \ldots, x_n))$ retourne "oui", alors $b_i := 1$, sinon $b_i = 0$.
- Retourner $(b_1, \ldots, b_n)$.

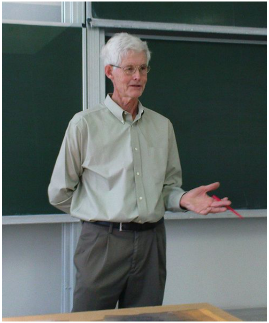
Réductions de Turing

On remarque que l'algorithme précedent 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 $A \in \mathbf{P}$.

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.

SAT est **NP**-complet - quelles conséquences ?

1. Si on réduit (par une réduction polynomiale) SAT à un problème *A*, alors *A* est **NP**-difficile.
   Il s'agit d'une *borne infèrieure de complexité* : le problème *A* est au moins aussi difficile que SAT (donc **NP**-difficile).

   > *Conséquence : en supposant* $\mathbf{P} \neq \mathbf{NP}$ *il n'existe pas d'algorithme polynomial pour A.*

2. Si un problème *A* se réduit (par une réduction polynomiale) à SAT, alors *A* appartient à **NP**.
   Il s'agit d'une *borne supérieure de complexité* : le problème *A* se trouve alors dans la classe **NP**.

   > *Conséquence : on peut utiliser la réduction à SAT + un SAT-solveur pour obtenir un algorithme pour A. Si A est un problème difficile, alors cet algorithme peut être bien plus efficace qu'un algorithme direct.*

Cook-Levin : preuve

On considère un problème $A \in \mathbf{NP}$ quelconque et on montre $A \leq_P \mathrm{SAT}$.

- $A \in \mathbf{NP}$ signifie qu'on a un vérificateur polynomial V pour le problème A :

  Pour toute instance positive x de A il existe y de taille polynomiale en x, tel que V accepte $\langle x, y \rangle$.

- Le problème SAT :

  φ est instance positive de SAT si il existe une valuation qui rend φ vraie.

Cook-Levin : preuve

- Pour tout algorithme polynomial P on peut construire (en temps polynomial) un circuit booléen C_P de taille polynomiale, dont les entrées $z_1, \ldots, z_n \in \{0, 1\}$ correspondent à l'entrée $z = z_1 \cdots z_n$ de P (codée en binaire), et tel que

  P accepte x si et seulement si C_P s'évalue à 1.

- Etant donné un circuit booléen C avec entrées $x_1, \ldots, x_k, y_1, \ldots, y_m$ et une valuation val : $\{x_1, \ldots, x_k\} \to \{0, 1\}$ on construit une formule booléenne φ_{val}, de taille polynomiale en taille(C), telle que :

  φ_{val} est satisfaisable si et seulement si il existe une valuation $\widetilde{val}$: $\{y_1, \ldots, y_m\} \to \{0, 1\}$ tel que C s'évalue à 1 sous la valuation $\langle val, \widetilde{val} \rangle$.

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

Le problème Somme d'entiers est le suivant :

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

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

C'est clairement dans **NP** : on peut vérifier pour $i_1, \ldots, i_p$ donné, si $x_{i_1} + \cdots + x_{i_p} = s$.
On montre que c'est **NP**-complet par une réduction 3-SAT $\leqslant$ Somme d'entiers.

Partition

Le problème Partition est le suivant :

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

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

C'est clairement dans **NP** : on peut vérifier si $X \subseteq \{1, \ldots, k\}$ est une solution.

Remarque Les problèmes Somme d'entiers et Partition sont *pseudo* **NP**-complets, c-à-d si les entiers sont codés en unaire (et donc polynomiaux dans la taille de l'entrée), ces problèmes peuvent être résolus en temps polynomial.

Réduction Somme d'entiers vers Partition

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

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

Réduction 3-SAT vers Somme d'entiers

- On construit à partir d'une formule 3-CNF $\varphi = C_1 \wedge C_2 \wedge \cdots \wedge C_m$ avec variables $x_1, \ldots, x_n$ une instance de Somme d'entiers $(x_1, \ldots, x_k, s)$ t.q. φ est satisfaisable ssi $(x_1, \ldots, x_k, s)$ 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 $\overline{x_i}$ apparaît dans C_j, et 0 sinon.

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.

Plan

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

- Calcul = programme, algorithme, ... (du latin "calculus" = petits cailloux)
- Une fonction $f : \mathbb{N} \to \mathbb{N}$ est calculable s'il existe un programme/algorithme qui calcule f.

Cette pseudo-définition est floue : 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 : on admettra la thèse de Church-Turing, qui dit que tous les modèles "raisonnables" de calcul sont équivalents.

Codages

- Sur l'ordinateur tout est codé en binaire (Unicode, ASCII, ...).
- Formellement, un codage binaire d'un ensemble D est une fonction injective (et calculable) $f : D \to \{0,1\}^*$, c-à-d. une fonction t.q. $f(d) \neq f(d')$ pour tout $d \neq d'$ dans D.
  Peut-on coder des ensembles quelconques ? Non, voir la notion d'ensemble dénombrable.
- Exemple : on peut coder un graphe G avec ensemble de sommets $V = \{1, 2 \ldots, n\}$ 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,
  - ...

Des fonctions non-calculables, ça existe !

- Combien de mots binaires y a-t-il ? autant que des entiers positifs ($\mathbb{N} = \{0, 1, 2, \ldots\}$). On dit que l'ensemble $\{0, 1\}^*$ est dénombrable. Les mots binaires peuvent être énumérés, par exemple : $0, 1, 00, 01, 10, 11, 000, \ldots$.
- Combien de programmes/algorithmes 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. Ensuite, on peut énumérer les programmes, en énumérant les mots $w \in \{0, 1\}^*$ représentant des programmes.
  L'ensemble des programmes est donc dénombrable également.
- Combien de fonctions $f : \mathbb{N} \to \mathbb{N}$ existe-t-il ? Autant que des nombres réels (ensemble non-dénombrable).
  Conséquence : il existe des fonctions $f : \mathbb{N} \to \mathbb{N}$ non-calculables.

Comme on vient de voir, l'existence de fonctions non-calculables (ou de problèmes non-résolubles) se montre par un argument de comptage.

On va donc rappeler dans la suite d'abord la notion de comptage (= ensembles dénombrables). Ensuite on va fixer une notion de "modèle de calcul" (= programmes WHILE). Finalement on va parler de la notion de (in)décidabilité et montrer que toute question non-triviale qu'on peut poser au sujet d'un programme quelconque, n'est pas résoluble algorithmiquement (= indécidable).

Ensembles dénombrables

- On note $\mathbb{N} = \{0, 1, 2, \ldots\}$ l'ensemble des entiers naturels.
- Un ensemble D est dit dénombrable s'il est en bijection avec $\mathbb{N}$.
- Un ensemble D fini ou dénombrable est dit au plus dénombrable. Equivalent : il existe une fonction injective $f : D \to \mathbb{N}$ (ou une fonction surjective $f : \mathbb{N} \to D$).
- Exemples. Les ensembles suivants sont dénombrables :
  1. $\mathbb{N} = \{0, 1, \ldots\}$,
  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.

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 Ces ensembles sont en bijection.

À une partie X de $\mathbb{N}$, on associe la suite $x = (x_n)_{n \geqslant 0}$ 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 $x = (x_n)_{n \geqslant 0}$ à valeurs dans $\{0, 1\}$, on associe bijectivement la fonction $f_x : \mathbb{N} \to \{0, 1\}$ définie par $f_x(n) = x_n$.

Diagonalisation

L'argument suivant, dû à Cantor et nommé diagonalisation, 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 : $e^1, e^2, \ldots$.
- Soit $x = (x_n)_{n \geqslant 0}$ 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.

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 que WHILE-calculable est équivalent à GOTO-calculable. Par contre, LOOP-calculable est plus restrictif.
- On va présenter un autre modèle de calcul, la machine de Turing), et montrer qu’elle définit la même classe de fonctions - qu’on va appeler la classe de fonctions calculables :

  WHILE-calculable = Turing-calculable

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 $x = y \cdot z$.

Syntaxe :

- variables res, $x, y, z, \ldots$ (valuées dans $\mathbb{N}$)
- constantes $0, 1, \ldots$
- opérations $+, -$
- instructions $x := y \pm c$, $x := c$, skip
- 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.

LOOP-calculable

- Soit P un programme LOOP utilisant les variables $x_0 = \text{res}, x_1, \ldots, x_\ell$.
  - L'entrée de P est un k-uplet $\vec{n} = (n_1, \ldots, n_k) \in \mathbb{N}^k$ ($k < \ell$), stocké dans les variables $x_1, \ldots, x_k$.
  - L'effet de P sur $\vec{x} = x_0, \ldots, x_\ell$ (défini inductivement) est noté $P(\vec{x}) \in \mathbb{N}^{\ell+1}$.
  - La sortie de P est la valeur de la variable res à la fin du calcul de P. La fonction calculée par P est notée f_P.
- Une fonction $f : \mathbb{N}^k \to \mathbb{N}$ est LOOP-calculable, s'il existe un programme LOOP P t.q. $f(\vec{n}) = f_P(0, \vec{n}, 0, \ldots, 0)$.
- 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.

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 $f : \mathbb{N}^k \to \mathbb{N}$ 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. Autrement dit : $f_P(0, \vec{n}, 0, \ldots, 0)$ n'est défini que si P a un calcul fini à partir des valeurs initiales $(0, \vec{n}, 0, \ldots, 0)$.
  Exemple :

  x := 1 ;
  WHILE $(x \neq 0)$ DO $(x := x + 1)$ OD

Programmes GOTO

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

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

Exemple (addition) :

(1) $x := y$;
(2) IF ($z = 0$) THEN GOTO 5 FI ;
(3) $x := x + 1$; $z := z - 1$;
(4) GOTO 2 ;
(5) HALT

Sémantique des programmes GOTO :

- L'instruction HALT est la dernière.
- Saut conditionnel : si x est zéro, continuer avec l'instruction I_ℓ, sinon avec l'instruction suivante (sauf si HALT).

Prop.

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

> *f est WHILE-calculable si et seulement si f est GOTO-calculable*

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

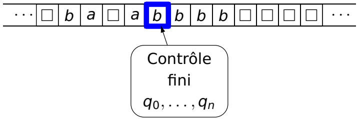
Prop. (voir plus loin)

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

Exemple : fonction d'Ackermann.

Machines de Turing

- Une machine de Turing comporte :
  - Une bande infinie à droite et à gauche faite de cases consécutives.
  - Dans chaque case se trouve un symbole, éventuellement blanc $\square$.
  - Une tête de lecture-écriture.
  - Un contrôle à nombre fini d'états.



Machines de Turing : intuition

- Le nombre d'états d'une machine de Turing est fini : un ordinateur a un nombre fini de registres.
- 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.

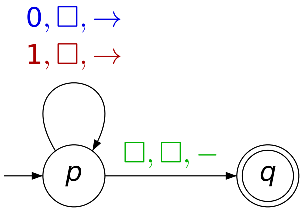
Machines de Turing : formalisation

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 $\Box \in \Gamma$.
- Σ : alphabet d'entrée, avec $\Sigma \subseteq \Gamma \setminus \{\Box\}$.
- 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.

Machines de Turing : représentation graphique

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



$$\delta = \{(p, 0, \Box, \rightarrow, p), (p, 1, \Box, \rightarrow, p), (p, \Box, \Box, -, q)\}$$

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.
- Le calcul ne s'arrête que si aucune transition n'est applicable.

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 $d = -$.

Configurations et calculs

- Une configuration représente un instantanné du calcul.
- La configuration uqv signifie que
  - L'état de contrôle est q
  - Le mot écrit sur la bande est uv, entouré par des $\Box$,
  - La tête de lecture est sur la première lettre de v.
- La configuration initiale sur w est donc $q_0 w$.
- 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.

$$C_0 \vdash C_1 \vdash C_2 \vdash \cdots$$

Calculs acceptants

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

$$C_0 \vdash C_1 \vdash C_2 \vdash \cdots$$

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).

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

$$C_0 \vdash C_1 \vdash C_2 \vdash \cdots C_n$$

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

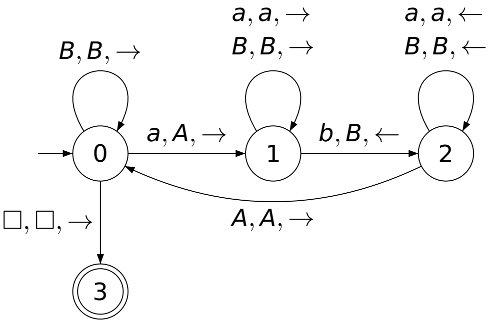
- 3 cas exclusifs : un calcul peut
  - soit s'arrêter sur un état acceptant,
  - soit s'arrêter sur un état non acceptant,
  - soit ne pas s'arrêter.
- On dit qu'une machine est déterministe si, pour tout $(p, a) \in Q \times \Gamma$, il existe au plus une transition de la forme $p \xrightarrow{a,b,d}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

Exemples de machines de Turing

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

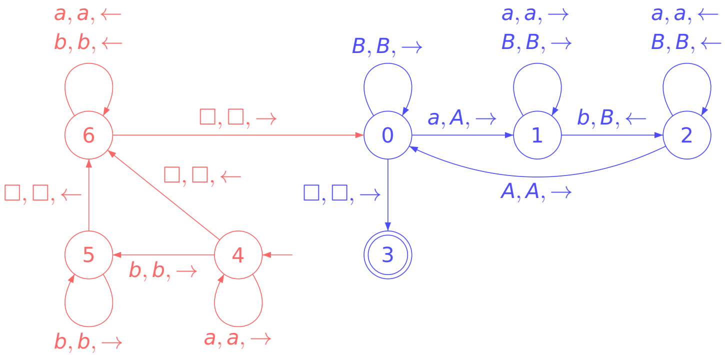
MT acceptant $(\{a^n b^n \mid n \geqslant 0\})^*$

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



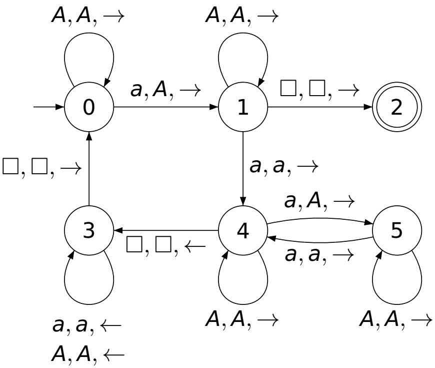
MT acceptant $\{a^n b^n \mid n \geqslant 0\}$

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



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

Idée : marquer un a sur 2.



Les machines de Turing peuvent calculer

- On peut utiliser les MT pour accepter des langages ou calculer.
- Une MT déterministe acceptant un langage L calcule la fonction caractéristique de L définie par

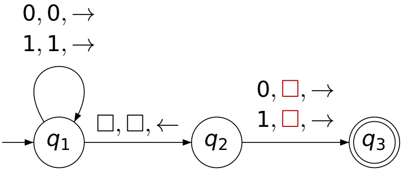
$$
\begin{aligned}
f : \Sigma^* &\to \{0,1\} \\
w &\mapsto \begin{cases} 0 & \text{si } w \notin L, \\ 1 & \text{si } w \in L. \end{cases}
\end{aligned}
$$

- Plus généralement, on peut associer à une MT déterministe M une fonction $f_M : \Sigma^* \to \Gamma^*$
  - On écrit la donnée $w \in \Sigma^*$ sur la bande,
  - Si la MT s'arrête avec sur la bande le mot $z \in \Gamma^*$, la fonction est définie par $f_M(w) = z$.

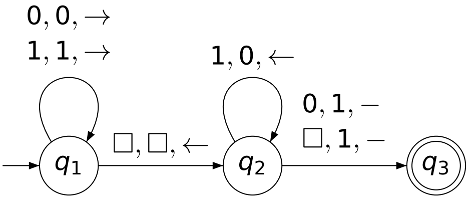
Exemples de machines de Turing

- Machine qui interprête son entrée comme un entier n, le remplace par $\lfloor n/2 \rfloor$ et s'arrête.
- Machine qui effectue l'incrément en binaire.
- Machine qui effectue l'addition de deux entiers en unaire.
- Machine qui effectue la multiplication de deux entiers en unaire.

Calcul de $\lfloor n/2 \rfloor$

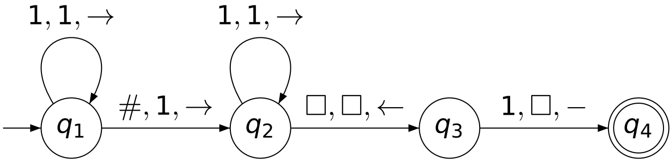


Incrément en binaire



Addition en unaire

- Le mot d'entrée est de la forme $1^n\#1^m$ interprété comme la donnée des entiers n et m.



“Thèse” de Church-Turing

La notion de “fonction calculable” ne dépend pas du modèle de calcul choisi (en supposant un modèle raisonnable).

Théorème

Pour toute fonction $f : \mathbb{N} \to \mathbb{N}$:
f est WHILE-calculable ssi f est Turing-calculable.

Machines de Turing et programmes WHILE

Simulations

Tout programme WHILE peut être simulé par une MT, et vice-versa.

- Simuler X par Y : coder les configurations de X par des configurations de Y de telle façon que l'effet d'un pas de calcul de X est réalisé par un ou plusieurs pas de calcul de Y.

Programme WHILE $\longrightarrow$ MT

Soit P un programme WHILE avec k variables. Une configuration de P est donc un tuplet $(i, n_0, \ldots n_{k-1}) \in [1 \ldots \ell] \times \mathbb{N}^k$. La MT M_P aura sur sa bande les valeurs $n_0, \ldots, n_{k-1}$ des variables (codées en binaire, séparées par $\#$), et le numéro de l'instruction actuelle i dans son contrôle fini. La simulation d'une opération élémentaire (addition par exemple) se fait par une séquence de pas de la MT.

Machines de Turing et programmes WHILE

MT $\longrightarrow$ programme WHILE

Soit M une MT qui utilise $k = |\Gamma|$ symboles sur sa bande. Une configuration uqv de M, $u, v \in \Gamma^*$, $q \in Q$ sera représentée par $(q, n_u, n_v) \in Q \times \mathbb{N} \times \mathbb{N}$, où n_w est l'entier dont la représentation en base k est le mot $w \in \Gamma^*$:

- le bit le moins significatif pour n_u est à droite, et pour n_v à gauche,
- $\square$ vaut 0, les autres symboles de $\Gamma \setminus \{\square\}$ sont codés par $1, \ldots, k-1$.

Machines de Turing et programmes WHILE

Exemple

Supposons que a représente le chiffre 1, et b le chiffre 2. Pour $\square b \square ba \; q \; \square bb \square$ on aura $k = 3$ et $n_v = 0 + 2 \cdot 3 + 2 \cdot 3^2 = 24$ et $n_u = 1 + 2 \cdot 3 + 2 \cdot 3^3 = 61$.

Simulation

Une transition de M à droite revient (en gros) à multiplier n_u par k et à diviser n_v par k. Et inversemment pour une transition à gauche.

Exemple

L'effet de la transition $(q, \square, p, a, \rightarrow)$ est de calculer $n'_v = 2 + 2 \cdot 3 = 8 = \lfloor n_v / 3 \rfloor$ et $n'_u = 3 \cdot n_u + 1 = 184$.

Dans cette partie on définit les notions de problème décidable (et indécidable), semi-décidable, ainsi que les notions d'algorithme et semi-algorithme.

On verra ensuite que le problème de l'arrêt de programme sur entrée donnée, ainsi que le problème PCP (correspondance de Post) sont indécidables (mais semi-décidables). On finira par le théorème de Rice, qui dit que toute propriété non-triviale de programme est indécidable.

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.
- Instance d'un problème A : une entrée de A. Une instance positive d'un problème de décision est une instance sur laquelle la réponse est OUI.
- De manière abstraite, un problème de décision A peut être interprété comme problème de langages :
  - On choisit un codage f des instances de A sur un alphabet Σ (ou $\mathbb{N}$).
  - L'ensemble des codages des instances positives de A définit un langage $L_A \subseteq \Sigma^*$ (ou $L_A \subseteq \mathbb{N}$).
  - La question si une instance I de A est positive revient à demander si $f(I) \in L_A$ (problème du mot).

Décidable et semi-décidable : définitions

Un programme WHILE P qui résout un problème de décision A est un programme qui s'arrête ET retourne 1 (= OUI) dans la variable *res* sur les instances positives de A. Sur les instances négatives, P retourne 0 (= NON) dans *res*, s'il termine.

- Un problème A est dit semi-décidable s'il existe un programme WHILE P qui le résout.

  Attention : On ne demande pas que le programme P termine toujours !

- Un problème A est dit décidable s'il existe un programme WHILE P qui le résout et qui termine sur toute entrée.

Vocabulaire

- Dans la littérature on emploie soit le terme “semi-décidable” ou “récursivement énumérable”.

  Rq : Un problème est semi-décidable ssi l’ensemble des instances positives est énumérable (d’où “récursivement énumérable”).
- De même, on emploie soit le terme “décidable” ou “récursif”.

  Rq : Un problème est décidable ssi l’ensemble des instances positives est énumérable en ordre lexicographique.
- Indécidable = pas décidable.

Problèmes décidables et semi-décidables

- Tout problème décidable est en particulier semi-décidable.
- Le complémentaire A^{co} d'un problème décidable est aussi décidable (complémentaire : inverser les réponses OUI/NON).
- Si un problème A et son complémentaire A^{co} sont semi-décidables, alors ils sont tous les deux décidables :
  - On met ensemble le programme WHILE P qui résout A et le programme WHILE P' qui résout A^{co} : le programme WHILE composé simule un pas de calcul de P, ensuite un pas de P', ensuite un pas de P, etc.
  - Le programmme s'arrête si si P ou P' s'arrête, et retourne OUI si P répond OUI, et NON si P' répond OUI.
  - Une instance est soit positive pour A ou pour A^{co}, donc le programme construit s'arrête toujours.

Un problème qui n'est pas semi-décidable : DIAG

Rappels :

- On peut coder chaque programme WHILE par un entier (rappel : l'ensemble des programmes WHILE est dénombrable).
- 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).
- On s'interesse aux programmes WHILE qui reçoivent un seul entier en entrée. On note $Acc(P) \subseteq \mathbb{N}$ l'ensemble des entiers n sur lesquels le programme P termine et retourne 1.

Le problème DIAG est défini par :

1. Entrée : entier n.
2. Question : Est-ce que $n \notin Acc(P_n)$?

Proposition. Le problème DIAG n'est pas semi-décidable.

Un problème semi-décidable, mais pas décidable : UNIV

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

- Entrée : entier n.
- Question : est-ce que $n \in Acc(P_n)$?

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

Conséquence : UNIV est indécidable. (Sinon, DIAG et UNIV seraient décidables tous les deux).

Codage/décodage de programmes WHILE

Un programme-WHILE $P : I_1; \ldots I_m$ est codé par $f(P) = (f(I_1), \ldots, f(I_m))$ de façon récursive :

- $f(x_i := x_j + c) = (0, i, j, c)$
- $f(\text{LOOP } (x_i) \text{ DO } P' \text{ OD}) = (1, i, f(P'))$
- $f(\text{WHILE } (x_i \neq 0) \text{ DO } P' \text{ OD}) = (2, i, f(P'))$

Donc, $f(P)$ est donc une liste de listes de ..., d'entiers.

Exemple : programme P

```
x0 := x1 ;
LOOP (x2) DO x1 := x1+1 OD ;
```

$f(P) = ((0, 0, 1, 0), (1, 2, (0, 1, 1, 1))$

Soit g une fonction qui code des listes d'entiers par un entier (par exemple $g(11, 7, 3) = 2^{11} \cdot 3^7 \cdot 5^3$). Alors le programme précédant est codé par $g(g(0, 0, 1, 0), g(1, 2, g(0, 1, 1, 1)))$. Ce codage (tout comme le décodage associé) est une fonction calculable.

Réductions (rappels)

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

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

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

Remarques

- Rappel : “A se réduit à A'” ne signifie PAS que A' est plus facile que A. Cela signifie que la recherche d’une solution pour A sur une instance x donnée peut être ramenée à la recherche d’une solution pour A' sur l’instance $f(x)$.
- La notion de réduction est symétrique : x est une instance positive de A SI ET SEULEMENT SI $f(x)$ est une instance positive de A'.
  Mais : $A \leq A'$ n’implique pas $A' \leq A$.
- Les réductions sont transitives : si $A_1 \leq A_2$ et $A_2 \leq A_3$, alors $A_1 \leq A_3$.

Exemple de réduction : UNIV $\leq$ UNIV$_0$, où UNIV$_0$ est le problème suivant :

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

Problème de l'arrêt

Les problèmes suivants sont indécidables :

- *HALT* : étant donnés 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 ?
- *UTILE* : étant donnés un programme WHILE P et une instruction I, est-ce que P utilise 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 ?
- *EQUIV* : étant donnés deux programmes WHILE P_1, P_2, est-ce que $Acc(P_1) = Acc(P_2)$?

Rq : Les 4 premières questions sont semi-décidables, les 2 dernières ne le sont pas.

L'indécidabilité hors du monde des entiers : le PCP

- Problème de correspondance de Post (1946).
- Donnée : n paires de mots (u_1, v_1),...,(u_n, v_n) sur un alphabet Σ.
- Question : existe-il une suite finie $i_1, \ldots, i_k$ ($k > 0$) 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.]

- $\rightsquigarrow$ les couples (u_i, v_i) peuvent être vus 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.

Le PCP modifié (PCPM)

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

Indécidabilité du PCP et PCPM

- On montre que

$$\mathrm{HALT}_0 \leqslant \mathrm{PCPM} \leqslant \mathrm{PCP}.$$

- Comme HALT_0 est indécidable, il en est de même de PCPM et de PCP.
- Accessoirement, on peut montrer que PCP $\leqslant$ PCPM.

PCP $\leqslant$ 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.

PCPM $\leqslant$ PCP

- On introduit une nouvelle lettre \$, et pour $a_1, \ldots, a_k \in A$, 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),...,(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 $((u_\ell, v_\ell))_{1 \leqslant \ell \leqslant n}$ a une solution si et seulement si le PCP sur l'instance $((x_\ell, y_\ell))_{1 \leqslant \ell \leqslant 2n+1}$ en a une.

Indécidabilité du PCPM

- On rappelle que la question UNIV : “est-ce que la MT M termine sur l’entrée w ?” est un problème indécidable.
- On montre une réduction UNIV $\leq$ PCPM.
- Étant donné une MT M et un mot w, on construit une instance $(u_\ell, v_\ell)_{1 \leqslant \ell \leqslant n}$ de PCPM telle que M termine sur w ssi l’instance $(u_\ell, v_\ell)_{1 \leqslant \ell \leqslant n}$ a une solution.
- On peut supposer que
  - M a un seul état d’arrêt : q_{OK},
  - M déplace sa tête à chaque transition.
- Idée : la seule solution sera la suite des configurations de M sur w, en partant de la configuration initiale et s’arrêtant dans une configuration terminale (dont l’état de contrôle est q_{OK}). La partie bleue sera en retard d’une configuration. Le retard est rattrapé à la fin, si l’état est q_{OK}.

La réduction

- Domino $(\#, \#q_0w\#)$. Les autres dominos :
- Dominos de copie : (a, a), $(\#, \#)$,
- Dominos de transitions :
- Pour chaque $p \xrightarrow{a,b,\rightarrow} q \in \delta$, il y a un domino (pa, bq).
- Pour chaque $p \xrightarrow{a,b,\leftarrow} q \in \delta$, il y a un domino (xpa, qxb) pour tout $x \in \Gamma$.
- Pour chaque $p \xrightarrow{\square,b,\rightarrow} q \in \delta$, il y a un domino $(p\#, bq\#)$.
- Pour chaque $p \xrightarrow{\square,b,\leftarrow} q \in \delta$, il y a un domino $(xp\#, qxb\#)$ pour tout $x \in \Gamma$.
- Dominos de synchronisation en fin de calcul : $(q_{OK}\#\#, \#)$ et pour chaque $a, b \in \Gamma$: (aq_{OK}, q_{OK}). $(q_{OK}b, q_{OK})$.

Quelques autres problèmes indécidables

Les problèmes suivants sont indécidables :

- Étant donné un jeu fini de tuiles carrées, avec conditions de compatibilité entre côtés (gauche/droite/haut/bas), déterminer si on peut paver le 1/4 de plan .
- Étant donnée une grammaire hors-contexte, 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.
- Étant donné un polynome à coefficients entiers, déterminer s'il a des racines entières (10ème problème de Hilbert).

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 $1 = i_1, i_2, \ldots, i_k$ (où $i_2, \ldots, i_k \in \{2, \ldots, n-1\}$) on a :

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

Réduction de Simple-PCP au problème d'accessibilité

- On peut chercher une solution pour une instance I de Simple-PCP dans un graphe orienté *fini* G_I : 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 I de Simple-PCP a une solution si et seulement si il existe un chemin dans G_I 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.

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 ?)

D'autres problèmes indécidables

Le problème suivant :

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

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.
- Indécidabilité : réduction à partir de HALT_0.

Il s'en suit que son complémentaire

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

n'est pas semi-décidable.

Théorème de Rice

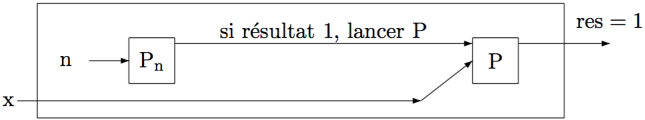
- Soit $\mathcal{E}$ l'ensemble des $X \subseteq \mathbb{N}$ tels que $X = Acc(P)$ pour un programme P.
- Une propriété d'ensembles semi-décidables est un sous-ensemble $\mathcal{P}$ de $\mathcal{E}$.
- Une propriété $\mathcal{P} \subseteq \mathcal{E}$ est triviale si $\mathcal{P} = \varnothing$ ou $\mathcal{P} = \mathcal{E}$.
- Attention Ne pas confondre $\mathcal{P} = \varnothing$ ($\mathcal{P}$ ne contient aucun ensemble) et $\mathcal{P} = \{\varnothing\}$ ($\mathcal{P}$ ne contient que l'ensemble vide).
- 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 $Acc(P) = Acc(P')$.

Théorème de Rice

- Toute propriété non triviale $\mathcal{P}$ d'ensembles semi-décidables est indécidable.
- Attention : il s'agit d'une propriété d'ensembles, et pas de programmes.
  Exemple : Pour le problème de l'arrêt il ne s'agit pas d'une propriété d'ensembles semi-décidables, mais d'une propriété de programmes.

Théorème de Rice : preuve

- Réduction à partir de UNIV (sur entrée n on demande si un programme accepte son propre code : $n \in Acc(P_n)$?)
- Quitte à changer $\mathcal{P}$ et $\mathcal{E} \setminus \mathcal{P}$, on suppose $\varnothing \notin \mathcal{P}$.
- Comme $\mathcal{P} \neq \varnothing$, il existe $X \in \mathcal{P}$. Soit **P** un programme WHILE tel que $Acc(\mathbf{P}) = X$.
- À partir de n on construit le programme R suivant :



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

Théorème de Rice : version fonctionelle

- Soit $\mathcal{F}$ l'ensemble des fonctions $f : \mathbb{N} \to \mathbb{N}$ calculables.
- Une proprieté de fonctions calculables est un sous-ensemble $\mathcal{P}$ de $\mathcal{F}$.
- Thm. de Rice : toute propriété non-triviale de fonctions calculables est indécidable.
- Exemples : on ne peut pas décider si une fonction calculable est croissante, constante, bornée etc.