Cours de calculabilité et complexité

Prof. Jean-François Raskin
Département d'Informatique
Faculté des Sciences
Université Libre de Bruxelles

Année académique 2009-2010

Organisation pratique du cours :

Références • **Introduction à la calculabilité, Pierre Wolper, InterEditions, 1991.**

- Computational Complexity, Christos H. Papadimitriou, Addison Wesley, 1994.

- Computers and Intractability - A Guide to the Theory of NP-Completeness, M. R. Garey and D. S. Johnson, Editor: W.H. Freeman and Compagny, 1979.

- Introduction to the theory of computation. Michael Sipser, PWS Publishing Compagny, 1997.

Il est **fortement recommandé** de se procurer la première référence, et de consulter les deux autres !!!

Matériel Les slides utilisés lors des cours seront disponibles sur la page web du cours.

Plan du cours

- Introduction - Motivations
- Notions de problèmes et d'algorithmes
- Cardinalité des ensembles - Diagonale de Cantor
- Modèles de calcul
  - les machines de Turing
  - les fonctions récursives
  - (le lambda calcul)
- Equivalences des modèles de calcul et thèse de Church

- Indécidabilité - Décidabilité

- Complexité
  - complexité en temps et en espace
  - réduction
  - intractabilité
  - P versus NP

- Complément de complexité (en option)
  - calculs parallèles
  - espace logarithmique
  - espace polynomiale

Introduction et motivations (I)

Dans différentes branches de la science, il y a des résultats à la vue desquels nous pouvons tout de suite conclure que certaines prétendues inventions sont impossibles :

- un moteur qui ne consomme pas d'énergie basé sur le principe du mouvement perpétuel (en contradiction avec les lois fondamentales de la physique);

- un chauffage révolutionaire fournissant de la chaleur et qui ne consomme pas d'énergie.

- ...

Il est intéressant de noter que pour réfuter de telles inventions, il est inutile d'étudier ces inventions en détails pour y trouver une faille. Nous pouvons les rejeter à priori.

Introduction et motivations (II)

En est-il de même pour certains projets en informatique. Y a-t-il des limites à ce que nous pouvons faire avec un ordinateur, des limites à ce que nous pouvons calculer ?

C'est en effet, le cas : tout n'est pas calculable ! Dans ce cours, nous allons essayer de comprendre les limites de l'informatique :

- nous verrons qu'il existe, en effet, des problèmes qui n'ont pas une solution algorithmique.

- nous verrons également que certains problèmes n'ont pas de solutions satisfaisantes en terme d'efficacité.

Introduction et motivations (III)

Quand un informaticien tente résoudre un problème et qu'il ne parvient pas à résoudre ce problème, il adopte généralement deux attitudes :

- (version optimiste) il se dit : "je pense que je suis proche de réussir, je vais corriger quelques détails et ajouter les derniers cas que mon algorithme ne considère pas" ;

- (version plus réaliste) il se dit : "je suis parti sur de mauvaises bases, je vais recommencer à zéro..."

Introduction et motivations (III)

Après avoir suivi ce cours, ce même informaticien se rendra peut-être compte que :

- le problème qu'il est censé résoudre est insoluble algorithmiquement : il n'existe aucun algorithme qui peut le résoudre et il n'en existera jamais !

Avouez qu'il est intéressant de se rendre compte de ce phénomène !

Introduction et motivations (IV)

Il y a une autre situation où le cours de calculabilité et complixité vous sera très utile:

- Patron : vu la hausse des prix du pétrole, nous ne pouvons plus gaspiller ! Je veux que tous nos représentants de commerce se déplacent optimalement : “plus 1km de trop !!!” Mais il ne faut pas non plus perdre du temps à la planification ! Faites-moi un programme qui calculera le circuit optimal pour nos représentants chaque matin !!!

- Vous : bien patron...

Introduction et motivations (IV)

Deux situations sont possibles : (i) vous avez malheureusement renoncé à suivre le cours de calculabilité et complexité, (ii) vous avez suivi le cours de calculabilité et complexité.

Introduction et motivations (IV)

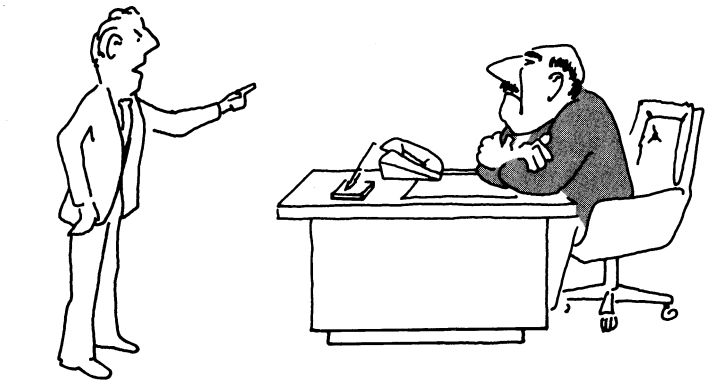
Situation (i) : après trois semaines de travail à la recherche d'un algorithme efficace pour résoudre le problème de votre patron, vous vous résignez à aller le trouver pour admettre que vous êtes certainement trop stupide et que vous n'arrivez pas à résoudre son problème de manière efficace (les seuls algorithmes que vous avez trouvés ne terminent pas en une journée même sur la machine la plus rapide de la compagnie...) Votre patron perd son calme et vous licencie !

Introduction et motivations (IV)

Situation (ii) : vous commencez à travailler sur le problème et vous vous rendez compte que les seules solutions que vous trouvez énumèrent chaque circuit possible (le problème c'est qu'il y en a un nombre exponentiel), aucune des heuristiques auquelles vous avez pensé ne marche pas dans tous les cas ! Et là, vous vous rappelez le bon vieux temps : quand vous êtiez étudiant..., et le cours de calculabilité et complexité que vous avez suivi !

Introduction et motivations (IV)

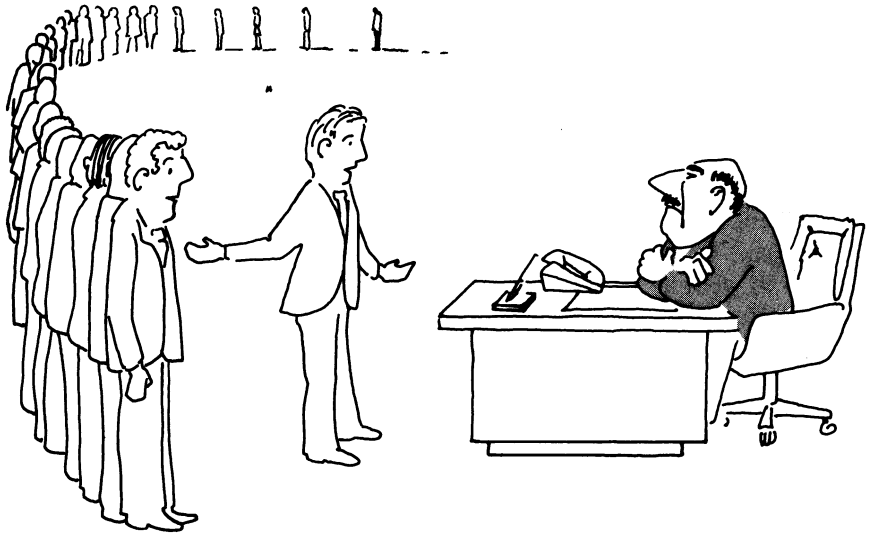
Confiant, vous allez dans le bureau de votre patron et vous lui dites que le problème n'est pas soluble efficacement et qu'il fait partie des problèmes NP-Complet, et donc que votre patron devra se contenter d'une approximation du circuit le plus court !



Normalement, votre patron vous félicitera (de ne pas avoir cherché en vain une solution efficace et exacte au problème, recherche qui à coups sûr aurait duré le reste de votre carrière) et vous obtiendrez une promotion !!!

Introduction et motivations (IV)

Si votre patron persiste dans son obstination, vous faites défiler dans son bureau tous les informaticiens qui ont déjà essayé de résoudre efficacement un problème NP-Complet et qui n'y sont jamais parvenus !



Notions de probleme et d'algorithme (I)
Qu'est-ce qu'un problème ?

Commençons par donner quelques exemples de problèmes :

- déterminer si un nombre naturel n est *pair* ou *impair*;

- calculer le PGCD de deux nombres naturels m et n;

- étant donné un graph G et deux sommets m et n de ce graphe, déterminer si il existe un chemin menant de m à n dans ce graphe;

Notions de probleme et d'algorithme (I)
Qu'est-ce qu'un problème ?

- étant donné une procédure Pascal, et un vecteur de valeurs pour les paramètres "input" de la procédure, déterminer si la procédure termine sur ces valeurs "input" (*problème de l'arrêt*).

- ...

Notions de problème et d'algorithme (II)
Qu'est-ce qu'un problème ?

Quelles sont les caractéristiques d'un problème?

- un problème est une question "générique", c'est-à-dire qu'un problème contient des paramètres ou variables libres. Lorsque l'on attribue une valeur à ces variables libres on obtient une *instance* du problème;

- un problème existe indépendement de toute solution ou de la notion de programme pour le résoudre;

- un problème peut avoir plusieurs solutions, plusieurs algorithmes différents peuvent résoudre le même problème.

Notions de problème et d'algorithme (II)
Qu'est-ce qu'un problème ?

Nous nous intéresserons à une classe spéciale de problèmes : les problèmes de **décision**.

Un problème est dit *de décision* si la réponse aux instances du problème est soit **oui** soit **non**.

Exemples :

- Déterminer si oui ou non un nombre entier n est pair est un problème de décision;

- Par contre déterminer la longueur minimale du circuit qui passe par tous les sommets d'un graphe n'est pas un problème de décision.

Notions de problème et d'algorithme (II)
Qu'est-ce qu'un problème ?

La classe des problèmes de décision est limitée mais suffisante pour illustrer les notions qui nous intéressent. Pour ne pas surcharger (inutilement) la formalisation, nous nous limiterons la plus part du temps aux problèmes de décision. Les techniques que nous allons étudier sont généralisables aux autres problèmes.

Nous verrons également que, par exemple, la plupart des problèmes d'optimisation sont "équivalents (pour une notion d'équivalence que nous préciserons plus tard) à des problèmes de décision.

Notions de problème et d'algorithme (II)
Qu'est-ce qu'un algorithme

Maintenant que nous avons une idée intuitive et précise de ce qu'est un problème essayons de préciser quelque peu la notion d'algorithme (ou encore de programme).

Derrière la notion d'algorithme se trouve la notion de **procédure effective** pour résoudre un problème (toutes les instances d'un problème).

Un programme Pascal est une procédure effective. Pourquoi ? Le code Pascal peut-être compilé et ensuite exécuté "**mécaniquement**" par le processeur. Une autre caractéristique essentielle d'une procédure effective est qu'elle contient exactement la marche à suivre pour résoudre le problème et qu'aucune décision supplémentaire ne doit être prise lors de l'exécution de la procédure.

Notions de problème et d'algorithme (II)
Qu'est-ce qu'un algorithme

Voici un exemple d'une procédure qui n'est pas effective pour résoudre le problème de l'arrêt :

> Déterminez si le programme n'a pas de boucle infinies ou d'appels récursifs infinis.

Il est clair que cette solution n'est pas effective : rien n'est dit au sujet de comment détecter les boucles infinies ou les appels récursifs infinis. Notons que nous établirons dans la suite de ce cours qu'il n'existe aucune procédure effective pour résoudre le problème de l'arrêt.

Notions de problème et d'algorithme (II)
Qu'est-ce qu'un algorithme

Dans la suite, nous aurions pu utiliser la notion d'algorithmes écrits dans un langage de programmation usuel pour formaliser la notion de procédure effective.

Nous ne choisirons pas cette option car elle serait trop lourde. En effet, pour qu'un programme écrit dans un langage usuel (comme Pascal) soit une procédure effective, il faut que le programme soit compilé.

Notions de problème et d'algorithme (II)
Qu'est-ce qu'un algorithme

Cette indirection complique la formalisation. Pour éviter cette étape de "compilation", nous adopterons des langages de programmation primitifs qui ont une forme tellement simple que leur procédure d'interprétation (comment les exécuter) est immédiate. Nous étudirons en particulier le formalisme de la **machine de Turing**.

Attention : pour qu'une procédure soit considérée comme effective pour résoudre un problème, il faut que celle-ci **se termine** sur toutes les instances du problème.

Notions de problème et d'algorithme (II)
Formalisation de la notion de problème

Si un problème doit être résolu par une procédure effective, il est naturel que les instances du problèmes soient accessibles à la procédure effective. Pour cela, nous avons besoin d'un **encodage** des instances du problème. Cet encodage doit pouvoir être manipulé par la procédure effective.

Si nous écrivions nos procédures effectives à l'aide d'un langage de programmation usuel, nous encoderions les instances de problèmes à l'aide d'entiers, de séquences d'entiers, de chaînes de caractères, ...

Notions de problème et d'algorithme (II)
Formalisation de la notion de problème

Ici, nous adopterons un point de vue un peu plus abstrait et considérerons seulement des chaînes de caractères sur un alphabet fini.

Remarque : Il est évident que cela est suffisant : en effet, de toutes façons tout est toujours ramené d'une façon ou d'une autre à des chaînes de 0 et de 1.

Notions de problème et d'algorithme (II)
Formalisation de la notion de problème
Quelques définitions

Un *alphabet* est un ensemble fini de symboles, dans la suite nous utiliserons $\Sigma, \Sigma_1, \Sigma_2, ...$ pour désigner des alphabets.

Quelques exemples d'alphabets : $\Sigma = \{a, b, c, d\}$, $\Sigma = \{0, 1\}$, $\Sigma = \{\clubsuit, \diamondsuit, \heartsuit, \spadesuit\}$, ...

Notons que la caractéristique importante d'un alphabet est son **nombre d'éléments** et non pas les symboles particuliers qui le composent (ces symboles n'ont aucune interprétation).

Notions de problème et d'algorithme (II)
Formalisation de la notion de problème
Quelques définitions

Un *mot* sur Σ est une séquence finie d'éléments de Σ.

Par exemple : $ababcdcdabcd$ est un mot sur $\Sigma = \{a, b, c, d\}$.

La *longueur* d'un mot est le nombre de symboles qu'il contient.

Par exemple : la longueur de la séquence $w = ababcdcdabcd$ est 12, on notera cela $long(w)$. On utilisera également la notation $w(i)$ pour dénoter le i^e symbole du mot w, et ϵ pour dénoter un mot vide (de longueur nulle)..

Notions de problème et d'algorithme (II)
Formalisation de la notion de problème

On appelera *fonction d'encodage* la fonction qui transforme une instance particulière d'un problème en son codage en terme de mot.

Dans la suite nous considérerons que chaque instance d'un problème est représentable par une séquence finie de symboles.

Notions de problème et d'algorithme (II)
Formalisation de la notion de problème

Soit un problème de décision dont les instances sont encodées par des mots définis sur un alphabet Σ. L'ensemble de tous les mots définis sur Σ peut être partitionné en trois sous-ensembles :

- les mots représentant des instances du problème pour lesquelles la réponse est oui, nous appelerons ces instances les *instances positives* du problème;

- les mots représentant des instances du problème pour lesquelles la réponse est non, ces sont les *instances négatives*;

- les mots qui ne représentent pas des instances du problème considéré.

Notions de problème et d'algorithme (II)
Formalisation de la notion de problème

Les deux dernières classes sont souvent regroupées et ainsi nous obtienons par chaque problème une partition de l'ensemble des mots en deux sous-ensembles :

- les mots qui représentent des instances positives du problème;

- les mots qui représentent des instances négatives du problème ou qui ne représentent pas d'instance du problème.

Notions de problème et d'algorithme (II)
Formalisation de la notion de problème

Un problème peut donc être caractérisé par l'ensemble des mots qui sont des instances positives du problème. Nous appelerons un ensemble de mots un langage.

Un *langage* est un ensemble de mots définis sur le même alphabet.

Souvent nous ne ferons pas la distinction entre:

- résoudre un problème;
- reconnaître de manière effective le langage des encodages des instances positives du problème.

Notations : $\emptyset$ dénote le langage vide, nous utiliserons $\mathcal{L}, \mathcal{L}_1, \mathcal{L}_2, \ldots$ pour désigner des langages.

Notions de problème et d'algorithme (II)
Formalisation de la notion de problème

Quelques exemples de langages :

- pour l'alphabet $\Sigma = \{a, b\}$,

  $$\mathcal{L} = \{aabbaabab, bbabba, bbaaaabbba\}$$

  est un langage;

- l'ensemble des programmes Pascal syntaxiquement corrects;

- l'ensemble des programmes Pascal qui terminent est un langage.

Nous montrerons que les deux premiers langages (c'est évident pour le premier) sont effectivement reconnaissables et par contre le troisième ne l'est pas.

Préliminaires : Cardinalité des ensembles et diagonale de Cantor (I)

La *cardinalité* est le terme technique utilisé en théorie des ensembles pour désigner la **taille** d'un ensemble.

Pour les ensembles finis, la taille d'un ensemble est simplement son nombre d'éléments.

Donc, deux ensembles finis ont la même cardinalité si et seulement si ils ont le même nombre d'éléments.

Préliminaires : Cardinalité des ensembles et diagonale de Cantor (I)

Pour des ensembles infinis, c'est un peu plus compliqué.

Si deux ensembles infinis peuvent être mis en **bijection**, c'est-à-dire qu'il existe une fonction bijective d'un ensemble vers l'autre (bijective=injective et surjective), alors nous dirons qu'ils ont la **même cardinalité**.

Grâce à cette notion de cardinalité, on peut regrouper les ensembles de même cardinalité en classes d'équivalence.

Préliminaires : Cardinalité des ensembles et diagonale de Cantor (I)

Etudions maintenant quelques ensembles qui ont la même cardinalité que l'ensemble des nombres naturels (l'ensemble infini le plus simple).

Définition: un ensemble est *dénombrable* (*énumérable*) s'il existe une bijection entre cet ensemble et l'ensemble des nombres naturels.

La cardinalité des ensembles dénombrables est notée $\aleph_0$ ($aleph - 0$).

Préliminaires : Cardinalité des ensembles et diagonale de Cantor (I)

Voici quelques exemples d'ensembles dénombrables :

- l'ensemble des nombres pairs est dénombrable. Voici une bijection qui établit cette propriété:

  $$\{(0,0),(2,1),(4,2),(6,3),\ldots\}$$

  Notons qu'un sous-ensemble stricte infini de l'ensemble des naturels a donc la même cardinalité que l'ensemble des naturels dans son entièreté.

Préliminaires : Cardinalité des ensembles et diagonale de Cantor (I)

- l'ensemble des mots sur l'alphabet $\{a, b\}$ est dénombrable. Pour obtenir une énumération, on classe les mots par tailles croissantes et par ordre alphabétique au sein d'une même taille. On obtient donc :

$$\{(\epsilon, 0), (a, 1), (b, 2), (aa, 3), (ab, 4),$$
$$(ba, 5), (bb, 6), \ldots\}$$

  De manière générale, l'ensemble des mots sur un alphabet fini est énumérable.

Préliminaires : Cardinalité des ensembles et diagonale de Cantor (I)

- l'ensemble des nombres rationnels est dénombrable. En effet, un nombre rationnel est caractérisé par une paire de nombres entiers $\frac{a}{b}$ (et les paires de nombres entiers sont dénombrables). Voici une façon d'énumérer les nombres rationnels :

  "énumérons les rationnels par ordre de sommes croissantes de leur numérateur et dénumérateur, et par ordre de numérateur et puis de dénumérateur pour les sommes identiques". Ceci donne :

$$\{(\tfrac{0}{1},0),(\tfrac{0}{2},1),(\tfrac{1}{1},2),(\tfrac{0}{3},3),(\tfrac{1}{2},4),\ldots\}$$

Préliminaires : Cardinalité des ensembles et diagonale de Cantor (I)

Exercices : montrez que

- l'ensemble des mots (finis) sur un alphabet énumérable est énumérable;

- l'ensemble des suites finies de nombres naturels est énumérable.

Préliminaires : Cardinalité des ensembles et diagonale de Cantor (I)

Montrons maintenant que tous les ensembles infinis ne sont pas énumérables.

Théorème : l'ensemble des sous-ensembles d'un ensemble dénombrable infini n'est pas dénombrable.

Preuve: Nous appliquons la méthode de la diagonale. Soit $A = \{a_0, a_1, a_2, \ldots\}$ un ensemble dénombrable, $S = \{s_0, s_1, s_2, \ldots\}$ l'ensemble de ses sous-ensembles que nous considérons ici dénombrable (démonstration par l'absurde). Montrons que l'ont peut déduire une contradiction.

Pour cela, nous considérons une matrice B doublement infinie qui contient une ligne pour

chaque élément de A et une colonne pour chaque élément de S.

L'élément $B[i,j]$ a la valeur 1 si $a_i \in s_j$ et la valeur 0 sinon. Considérons maintenant l'ensemble $D = \{a_i \mid a_i \notin s_i\}$. Clairement D est un sous-ensemble de A. Mais D n'est égal à aucun des s_i énumérés comme colonnes de B. En effet, supposons que $D = s_k$, ceci est impossible car par définition de D, on a $a_k \in D$ ssi $a_k \notin s_k$.

Rem : on appelle cette méthode la méthode de la diagonale car D est obtenu en changeant de manière systématique la valeur de la diagonale de la matrice B).

Préliminaires : Cardinalité des ensembles et diagonale de Cantor (I)

De manière similaire, on peut montrer que :

- l'ensemble des fonctions des naturels dans les naturels n'est pas dénombrable;
- l'ensemble des séquences infinies sur un alphabet fini n'est pas dénombrable;
- il y a plus de langages que de langages réguliers.

Exercices : établissez ces trois affirmations.

Préliminaires : Cardinalité des ensembles et diagonale de Cantor (I)

Les ensembles qui ont la même cardinalité que l'ensemble des sous-ensembles d'une ensemble dénombrable ont une cardinalité notée $\aleph_1$.

Voici quelques exemples d'ensembles qui ont la cardinalité $\aleph_1$:

- l'ensemble des séquences infinies sur un alphabet fini;

- l'ensemble des fonctions des naturels dans les naturels;

Préliminaires : Cardinalité des ensembles et diagonale de Cantor (I)

- l'ensemble des nombres réels compris entre 0 et 1 (pour montrer cela, il suffit de savoir que chaque nombre réel entre 0 et 1 est représenté par sa suite infinie de décimales).

On dira que les ensembles de cardinalité $\aleph_1$ ont la puissance du continu.

Préliminaires : Cardinalité des ensembles et diagonale de Cantor (I)

Le théorème de Cantor est une des causes des limites fondamentales de l'informatique (et des systèmes formels en général).

Il est évident que l'ensemble des procédures effectives est dénombrable alors que l'ensemble des problèmes est non dénombrable (ensemble des sous-ensembles de l'ensemble des mots finis).

Plan du cours

- Introduction - Motivations
- Notions de problèmes et d'algorithmes
- Cardinalité des ensembles - Diagonale de Cantor
- Modèles de calcul
  - les machines de Turing
  - les fonctions récursives
  - (le lambda calcul)
- Equivalences des modèles de calcul et thèse de Church

- Indécidabilité - Décidabilité

- Complexité
  - complexité en temps et en espace
  - réduction
  - intractabilité
  - P versus NP

- Complément de complexité (en option)
  - calculs parallèles
  - espace logarithmique
  - espace polynomiale

Modèles de calcul

Structure du chapitre :

- les machines de Turing
- les fonctions récursives
- (le lambda calcul)

Modèles de calcul
Introduction

Nous définirons tout d'abord le formalisme des **machines de Turing**. C'est ce modèle que nous choisirons pour donner une définition formelle à la notion intuitive de procédure effective (algorithme).

Il convient de justifier ce choix. Notons qu'une démontration formelle de l'adéquation de ce choix est impossible vu que nous n'avons pas de définition formelle de ce qu'est une procédure effective (c'est exactement ce que nous cherchons). Pour justifier le choix des machines de Turing, nous nous baserons sur des arguments de différents types.

Modèles de calcul
Introduction

Types d'arguments :

- nous allons étudier plusieurs extensions du modèle des machines de Turing et montrer que les langages acceptés par ces machines sont identiques aux langages acceptés par les machines de Turing;

- de plus, nous allons montrer que tout problème résolvable par un programme écrit dans un langage usuel et exécuté sur un ordinateur peut l'être par une machine de Turing;

- nous montrerons également qu'une autre formalisation de la notion de procédure effective, basée sur une idée tout à fait différente : les **fonctions récursives**, est équivalente.

Modèles de calcul
Les machines de Turing
Introduction

Nous allons maintenant définir un formalisme pour reconnaître des langages. Précédement, vous avez déjà étudié (dans le cadre du cours de théorie des langages) différentes classes d'automate qui permettent de reconnaître différentes classes de langages.

Par exemple, on sait que les *automates finis* reconnaissent la classe des *langages réguliers*, la classe des *automates à pile* reconnaissent la classe des *langages hors-contexte*. Est-ce que cette dernière classe est un bon candidat pour notre notion de procédure effective ?

Modèles de calcul
Les machines de Turing
Introduction

La réponse est malheureusement non, car par exemple le langage

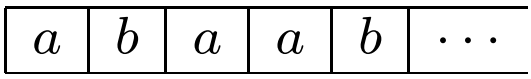
$$\{a^n b^n c^n \mid n \geq 0\}$$

ne peut être reconnu par aucun automate à pile mais il est clair qu'il doit exister une procédure effective qui lorsqu'on lui donne un mot fini détermine si oui ou non ce mot appartient au langage ci-dessus.

Modèles de calcul
Les machines de Turing
Composants

Une machine de Turing déterministe est composée :

- d'une mémoire infinie sous forme d'un "ruban" divisé en cases. Chaque case du ruban peut contenir un symbole d'un *alphabet* appelé *alphabet de ruban*.



- d'une *tête de lecture* qui se déplace le long du ruban et qui pointe vers une case du ruban.

- d'un *ensemble fini d'états* parmi lesquels on distingue un *état initial*.

Modèles de calcul
Les machines de Turing
Composants

- une *fonction (partielle) de transition* qui pour chaque état de la machine et symbole se trouvant sous la tête de lecture précise (si elle est définie):
  - l'état suivant,
  - le caractère qui sera écrit sur le ruban à la place du caractère qui se trouve sous la tête de lecture,
  - un sens de déplacement de la tête (une case à la fois).

Modèles de calcul
Les machines de Turing
Exécution

Une machine de Turing s'exécute selon la procédure suivante:

- Initialement, le mot d'entrée se trouve au début du ruban. Les autres cases du ruban contiennent toutes un symbole spécial appelé le *symbole blanc*, on le notera ♯. La tête de lecture est sur la première case du ruban (extrémité gauche) et la machine se trouve dans son état initial.

Modèles de calcul
Les machines de Turing
Exécution

- A chaque étape de l'exécution, la machine, si la fonction de transition est définie :
  - lit le symbole se trouvant sous la tête de lecture,
  - remplace ce symbole par le symbole donné par la fonction de transition,
  - déplace sa tête de lecture d'une case vers la gauche ou vers la droite suivant le sens précisé par la fonction de transition,
  - change d'état comme précisé par la fonction de transition.

- Un mot est *accepté* par la machine lorsque l'exécution de celle-ci atteint un état accepteur, est *rejeté* si l'exécution s'arrête avant d'atteindre un état accepteur ou est infinie et n'atteint jamais d'état accepteur.

Modèles de calcul
Les machines de Turing
Définition

Passons maintenant à une définition plus formelle. Une machine de Turing M est formellement décrite par les 7 éléments suivants :

$$M = (Q, \Gamma, \Sigma, \delta, s, B, F)$$

- Q est un ensemble fini d'états,
- Γ est l'alphabet du ruban,
- Σ est l'alphabet d'entrée (l'alphabet utilisé par le mot d'entrée), on a $\Sigma \subset \Gamma$,
- $s \in Q$ est l'état initial,

- $F \subseteq Q$ est l'ensemble des états accepteurs,

- $B \in \Gamma \setminus \Sigma$ est le "symbole blanc", que l'on notera $\sharp$,

- $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ est la fonction de transition (L et R sont utilisés pour représenter respectivement le déplacement à gauche et à droite de la tête de lecture). Cette fonction est partielle.

Modèles de calcul
Les machines de Turing
Configuration

Pour définir formellement la notion d'exécution d'une machine de Turing ainsi que la notion de mot accepté par une machine de Turing, nous avons besoin de la notion de configuration.

Une *configuration* contient toute l'information nécessaire à la poursuite de l'exécution de la machine, c'est-à-dire:

- l'état dans lequel la machine se trouve,
- le contenu du ruban,
- la position de la tête de lecture sur le ruban.

Modèles de calcul
Les machines de Turing
Configuration

La seule information “difficile” à représenter est le ruban : c’est une suite **infinie** de symboles.

Mais notons tout de même, cela nous sera utile dans la suite, qu’à tout moment seul un nombre fini de symboles sur le ruban sont différents du symbole ♯.

Il importe donc seulement de garder le prefixe **fini** de symboles qui sont différents de ♯.

Modèles de calcul
Les machines de Turing
Configuration

Plus formellement, nous dénoterons une configuration par une triplet

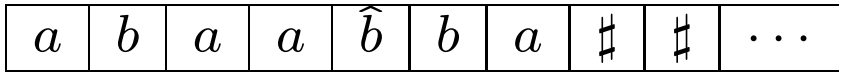
$$(q, w_1, w_2)$$

où :

- $q \in Q$ est l'état de la machine;
- $w_1 \in \Gamma^*$ est le mot apparaissant sur le ruban avant (strictement) la position de la tête de lecture;
- $w_2 \in \epsilon \cup \Gamma^*(\Gamma \setminus \{\sharp\})$, le mot se trouvant sur le ruban entre la position de la tête de lecture et le dernier caractère **non blanc**.

Modèles de calcul
Les machines de Turing
Configuration

Illustrons la définition de configuration. Considérons une machine dans l'état q avec le ruban suivant :



et la tête de lecture en position 5 (le symbole est coiffé d'un chapeau), alors la configuration de la machine est :

$$(q, abaa, bba)$$

Si la tête de lecture se trouve en position 8 (premier caractère blanc) alors la config est

$$(q, abaabba, \epsilon)$$

Modèles de calcul
Les machines de Turing
Exécution

Nous allons maintenant définir formellement l'évolution de la configuration d'une machine en définissant une fonction de transition entre les configurations.

Soit une configuration (q, α_1, α_2). Ecrivons cette configuration sous la forme $(q, \alpha_1, b\alpha'_2)$ en prenant $b = \sharp$ si $\alpha_2 = \epsilon$. Les configurations atteignables à partir de (q, α_1, α_2) sont dans la machine M alors définies comme suit:

- si $\delta(q, b) = (q', b', R)$ nous avons alors

$$(q, \alpha_1, b\alpha'_2) \vdash_M (q', \alpha_1 b', \alpha'_2)$$

- si $\delta(q, b) = (q', b', L)$ et si $\alpha_1 \neq \epsilon$ et est donc de la forme $\alpha'_1 a$, nous avons alors

$$(q, \alpha'_1 a, b\alpha'_2) \vdash_M (q', \alpha'_1, ab'\alpha'_2)$$

Modèles de calcul
Les machines de Turing
Exécution

Nous pouvons maintenant définir la notion de dérivation en plusieurs étapes.

Une configuration C' est dérivable en plusieurs étapes de la configuration C avec la machine M, ce qui sera noté $C \vdash_M^* C'$, si il existe $k \geq 0$ et des configurations intermédiaires $C_0, C_1, \ldots, C_k$ telles que :

- $C_0 = C$;
- $C' = C_k$;
- $C_i \vdash_M C_{i+1}$ pour $0 \leq i < k$.

Modèles de calcul
Les machines de Turing
Langage accepté

Nous sommes maintenant en position pour définir le *langage accepté par une machine de Turing*.

Le langage $L(M)$ accepté par une machine de Turing est l'ensemble des mots w tels que

$$(s, \epsilon, w) \vdash_M^* (q, \alpha_1, \alpha_2) \text{ avec } q \in F.$$

Ce sont donc les mots sur lesquels l'exécution de la machine atteint un état accepteur.

Modèles de calcul
Les machines de Turing
Exemples

Nous allons maitenant considérer deux exemples :

- une machine de Turing qui accepte le langage

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

- une machine de Turing qui accepte l'ensemble des mots finis qui sont des palindromes.

Modèles de calcul
Les machines de Turing
Langages décidés

Nous allons maintenant nous intéresser à une notion importante : celle de **langage décidé**.

Une machine de Turing qui accepte un langage L ne représente pas nécessairement une procédure effective pour reconnaître ce langage. En effet la définition de langage accepté que nous avons donnée n'exclut pas le fait que la machine peut ne jamais terminer son exécution sur un mot qui n'appartient pas au langage que la machine reconnaît. Vu que notre but est de donner une caractérisation formelle de la notion procédure effective pour reconnaître les instances positives d'un problème de décision, nous devons tenir compte des exécutions divergentes.

Modèles de calcul
Les machines de Turing
Langages décidés

Donc à partir d'une configuration (s, ϵ, w), plusieurs cas peuvent se présenter :

- l'exécution de la machine atteint un état accepteur et la machine accepte le mot;

- l'exécution de la machine s'arrête après un temps fini soit parce qu'aucune transition n'est possible ou l'exécution tente de faire bouger la tête de lecture vers la gauche alors que la tête pointe sur la première case du ruban. Dans ces deux cas le mot est rejeté par la machine.

Modèles de calcul
Les machines de Turing
Langages décidés

- l'exécution de la machine est infinie et donc ne termine jamais. Le problème est qu'à aucun moment on ne peut être sûr que l'on peut arrêter l'exécution et que le mot sera rejeté.

Modèles de calcul
Les machines de Turing
Langages décidés

On va maintenant introduire le nouveau concept de *langage décidé* qui exclut la troisième possibilité. Pour définir ce nouveau concept, nous avons besoin du concept d'*exécution* d'une machine de Turing.

L'*exécution* d'une machine de Turing sur un mot w est la suite de configurations

$$(s, \epsilon, w) \vdash_M C_1 \vdash_M C_2 \vdash_M \ldots \vdash_M C_k \vdash_M \ldots$$

maximale.

Modèles de calcul
Les machines de Turing
Langages décidés

Maximale veut dire que soit

- elle est infinie,
- ou elle termine dans un état accepteur,
- ou elle se termine dans une configuration où aucune configuration n'est dérivable.

Modèles de calcul
Les machines de Turing
Langages décidés

Un langage L est *décidé* par une machine de Turing M si :

- M accepte L,
- M n'a pas d'exécution infinie.

Par conséquent, un langage décidé par une machine de Turing peut être reconnu par une procédure effective.

Modèles de calcul
Les machines de Turing
Langages récursifs et récursivement énumérables

Les langages respectivement décidés et acceptés par les machines de Turing sont appelés *récursifs* et *récursivement énumérables*.

Un langage est *récursif* s'il est décidé par une machine de Turing.

Un langage est *récursivement énumérable* s'il est accepté par une machine de Turing.

Modèles de calcul
Thèse de Turing-Church

La thèse de Church au sujet des machines de Turing est la suivante :

> *Les langages reconnus par une procédure effective sont ceux décidés par une machine de Turing*.

Comme nous l'avons déjà souligné, cet énoncé est une **thèse** et non pas un théorème!

Cette thèse est le maillon qui nous manquait pour montrer qu'il existe des problèmes qui ne peuvent pas être résolu par une procédure effective.

Modèles de calcul
Thèse de Turing-Church

Il est clair que ce qui est décidé par une machine de Turing l'est par une procédure effective (on peut par exemple interpréter une machine de Turing sur un ordinateur).

Par contre l'autre direction est plus délicate : tout langage reconnu par une procédure effective est-il décidé par une machine de Turing ? (ou encore tout problème soluble algorithmiquement est-il soluble avec une machine de Turing ?)

Comment peut-on "justifier" la thèse de Church ?

Modèles de calcul
Thèse de Turing-Church
Justifications

Deux types de justifications pour la thèse de Church :

- une variété d'extensions que l'on peut apporter aux machines de Turing ne changent pas la classe des langages qu'elles décident;

- d'autres formalisations de la notion de procédure effective (nous étudierons les fonctions récursives et plus tard des modèles plus proche de nos ordinateurs d'aujourd'hui, on pourrait également parler du λ-calcul), basées sur des concepts tout à fait différents (en tout cas pour les fonctions récursives et pour le λ-calcul), sont équivalentes.

Modèles de calcul
Thèse de Turing-Church
Simplicité des machines de Turing

Pourquoi avoir choisi le modèle “simpliste” des machines de Turing ?

Justement pour leur simplicité !

Modèles de calcul
Thèse de Turing-Church
Extensions

Première extension : **ruban doublement infini**

Que se passe-t-il si on considère une machine avec un ruban infini vers la droite mais également vers la gauche ?

La définition d'une telle machine est identique à celle d'une machine de Turing habituelle à l'exception du fait suivant : la tête de lecture peut toujours se déplacer vers la gauche.

On peut montrer que toute machine à ruban doublement infini peut être "simulée" par une machine à un ruban simplement infini. Nous ne donnons pas la démonstration détaillée, seulement l'idée. Une preuve complète est donnée dans [PW91, page 143-144].

L'idée de la simulation est la suivante : elle consiste à replier le ruban en deux et considérer des pairs de symboles obtenues par ce repliage.

Il suffit alors de considérer une machine sur un alphabet composé des pairs de symboles de l'alphabet de départ et redéfinir la relation de transition de manière adéquate.

Modèles de calcul
Thèse de Turing-Church
Extensions

Autre extension : **machine de Turing à rubans multiples**

Considérons une machine avec plusieurs rubans et plusieurs têtes de lecture. L'état d'une telle machine est constitué de la location courante, du contenu de chaque ruban et de la position de chaque tête de lecture.

On peut à nouveau simuler le comportement d'une telle machine avec un alphabet de symboles composés décrivant le contenu des différents rubans et la position des têtes de lecture.

Modèles de calcul
Thèse de Turing-Church
Extensions

Par exemple, pour une machine à deux rubans, on utilise des symboles de la forme (a_1, a_2, b_1, b_2) où a_1 représente la valeur dans le premier ruban et $a_2 = 1$ si la tête de lecture du premier ruban est dans cette position du ruban, $a_2 = 0$ sinon, b_1 et b_2 encode la même information mais pour le deuxième ruban.

Comment simuler un étape de l'exécution de la machine à deux rubans ?

- trouver la position des têtes de lecture et déterminer les symboles lus,

- modifier ces symboles, déplacer les têtes de lecture et changer d'état.

Modèles de calcul
Thèse de Turing-Church
Extensions

Un autre modèle : **les machines à accès direct**.

Une machine RAM (random access memory) est une machine “semblable” aux ordinateurs réels. Supposons que la machine à simuler comporte une mémoire à accès direct et un certain nombre de registres dont un compteur de programme.

Montrons qu’une machine de Turing (multi-rubans et donc mono-ruban) peut simuler une machine RAM.

Modèles de calcul
Thèse de Turing-Church
Extensions

Les idées derrières la simulation :

- un ruban est utilisé pour chaque registre et un ruban est utilisé pour la memoire

- le contenu de la mémoire est représentée par des paires de mots (adresse, contenu).

$\sharp$	0	$*$	v_0	1	$*$	v_1	$\sharp$	...	$\sharp$	a	a	$*$	v_2	...

Modèles de calcul
Thèse de Turing-Church
Extensions

La machine de Turing simule alors la machine à mémoire RAM de la façon suivante :

- Parcourir le ruban représentant la mémoire jusqu'à trouver l'adresse correspondant au contenu du compteur de programme (PC);

- Lire et décoder l'instruction se trouvant à cette adresse;

Modèles de calcul
Thèse de Turing-Church
Extensions

- Le cas échéant trouver les opérandes de cette instruction;

- Exécuter l'instruction, ce qui implique éventuellement la modification de la mémoire et/ou des registres;

- Incrémenter le compte de programme (sauf si l'instruction est une instruction de branchement) et passer au cycle suivant.

Modèles de calcul
Thèse de Turing-Church
Extensions

Une modification plus fondamentale : **machines de Turing non-déterministes**

Jusqu'à présent nous nous sommes limités à des machines qui pour chacune de leurs configurations ont au plus une configuration atteignable. Que se passe-t-il si on relache cette propriété et que l'on considère des machines où une configuration peut-être suivie par non plus une seule mais un ensemble de configurations, c'est-à-dire si on considère une **relation de transition** à la place d'une fonction de transition ?

Dans ce cas δ devient Δ de type :

$$\Delta : (Q \times \Gamma) \times (Q \times \Gamma \times \{L, R\})$$

Donc à une pair (q, b) correspond un ensemble de triples (q', b', X) où $X \in \{L, R\}$, la machine peut choisir n'importe lequel de ces triplets.

Exercice : redéfinir formellement la notion de configuration successeur.

Modèles de calcul
Thèse de Turing-Church
Extensions

Les autres notions sont identiques à celles données pour les machines déterministes.

Néanmoins, une machine non-déterministe n'a pas une exécution unique sur un mot w et on impose seulement qu'une exécution termine dans un état accepteur pour accepter w.

Est-ce que le non-déterminisme apporte un pouvoir d'expression accru ?

Modèles de calcul
Thèse de Turing-Church
Extensions

Theorème : tout langage accepté par une machine de Turing non déterministe est aussi accepté par une machine de Turing déterministe.

note : la notion de décidé n'as pas de sens pour une machine non-déterministe.

Comment peut-on montrer que le non-déterminisme n'apporte pas de pouvoir d'expression supplémentaire ?

On va montrer comment une machine déterministe peut "simuler" une machine non-déterministe.

Modèles de calcul
Thèse de Turing-Church
Extensions

On va montrer qu'un machine déterministe peut simuler toutes les exécutions d'une machine non-déterministe.

Difficultés :

- il est difficile de simuler toutes les exécutions simultanément : par exemple, comment partager le ruban entre plusieurs executions ?

- on ne peut pas non plus simuler les exécutions une par une. En effet, si on commence une exécution qui est infinie, on ne la terminera jamais... et si il existe une autre exécution qui termine dans un état accepteur, elle ne sera jamais exécutée et on ne se rendra pas compte que le mot est accepté !

Solution : on adopte une méthode intermédiaire.

Modèles de calcul
Thèse de Turing-Church

La solution est de considérer toutes les exécutions en parallèle mais en considérant des prefixes de longueur croissante. D'abord, tous les prefixes de longeur 1, puis 2, puis 3, ...

Comment procéder ?

La première propriété importante dont nous avons besoin est le fait que les machines de Turing sont à branchement fini. Le nombre maximum de successeurs d'une configuration est donné par :

$$\max_{q \in Q, q \in \Gamma} | \{((q,a),(q',x,X)) \in \Delta\} |$$

Modèles de calcul
Thèse de Turing-Church

Supposons que pour chaque pair (q, a), on numérote les choix permis par la relation de transition de 1 à (maximum) r. Avec cette convention, chaque préfixe de longueur m d'une exécution peut-être représenté par une suite de m nombres inférieurs à r.

On peut alors construire une machine de Turing déterministe à trois rubans qui simule la machine non-déterministe de la façon suivante:

- le premier ruban contient le mot d'entrée et n'est pas modifié (ainsi le mot est toujours là quand on recommence les exécutions);

Modèles de calcul
Thèse de Turing-Church
Extensions

- le deuxième ruban servira à contenir les séquences de nombres inférieurs à r;

- le troisième ruban sert à la machine déterministe pour simuler le comportement de la machine non déterministe.

Modèles de calcul
Thèse de Turing-Church
Extensions

La machine déterministe procède alors comme suit:

1. sur le deuxième ruban, elle génère toutes les séquences de nombres inférieurs à r (alphabet fini), par ordre de longueurs croissantes;

2. chaque fois qu'une séquence est générée, elle simule la machine non-déterministe en résolvant les choix comme donnés par la séquence;

3. elle s'arrête et accepte dès qu'une des séquences générées représente une exécution de la machine non-déterministe qui se termine dans un état accepteur.

Modèles de calcul
Machine de Turing universelle

On peut montrer en fait qu'il existe une machine de Turing qui permet de simuler toutes les machines de Turing.

On donne à cette machine deux entrées : le mot d'entrée et un encodage de la machine de Turing qu'on veut simuler. Une telle machine est appelée une machine de Turing universelle.

Nous reviendrons plus en détail sur cette machine lorsque nous nous intéresserons aux problèmes indécidables.

Modèles de calcul
Fonctions calculables par les machines de Turing

Jusqu'à présent, nous avons restreint notre attention aux problèmes de décision. Un mot qui encode une instance de problème est accepté par la machine de Turing si il représente une instance positive du problème, sinon il est rejeté par la machine.

Comment peut-on faire "calculer un résultat" à une machine de Turing et obtenir une notion de fonction calculable par une machine de Turing ?

L'idée est de considérer le contenu du ruban en fin d'exécution.

Modèles de calcul
Fonctions calculable par
les machines de Turing

Une machine de Turing calcule une fonction

$$f : \Sigma^* \to \Sigma^*$$

si, pour tout mot d'entrée w, elle s'arrête toujours dans une configuration où $f(w)$ se trouve sur le ruban.

Rem : décider un langage revient à calculer une fonction dans le codomaine $\{0, 1\}$.

Modèles de calcul
Fonctions calculable par les machines de Turing

On peut réénoncé la thèse de Church-Turing en termes de calcul de fonctions :

> *Les fonctions calculables par une procédure effective sont les fonctions calculables par une machine de Turing*

Exercice :

- expliquez comment on peut faire calculer une fonction de $\boldsymbol{N}$ dans $\boldsymbol{N}$ par une machine de Turing;

- prouvez qu'il existe des fonctions de $\boldsymbol{N}$ dans $\boldsymbol{N}$ qui ne sont pas calculables par une machine de Turing.

Modèles de calcul
Les fonctions récursives

Nous venons de terminer l'introduction aux machines de Turing en montrant qu'elle pouvait calculer des fonctions. D'autres approches sont envisageables pour formaliser la notion de fonction calculable par une procédure effective. Une formalisation alternative est donnée par la notion de fonction récursive.

Considèrons les opérations de bases sur les nombres entiers, comme $+, -, \times, \uparrow, \ldots$. Il est clair qu'elles sont calculables.

Modèles de calcul
Les fonctions récursives

Pourquoi ?

- on peut par exemple connaître un algorithme pour $+, -$ qui étant donné un encodage binaire de a et b calcule l'encodage binaire de $a + b$, $a - b$.

- on peut calculer les fonctions qui sont définissables à partir des fonctions de base. Par exemple, si on a un algorithme pour $+$ et $-$, il est clair qu'on pourra calculer la fonction $a - (b + c)$ par exemple.

Mais il y a des fonctions qui ne peuvent pas être obtenues par composition **simple** de fonctions de base. Pensez à la fonction factorielle: $n!$.

Modèles de calcul
Les fonctions récursives

La décomposition de la fonction factorielle en terme de produits est donnée par :

$$n! = n \times (n-1) \times (n-2) \times \ldots \times 1$$

Malheureusement cette décomposition dépend de la valeur de n. Pourtant il est clair que cette fonction est calculable.

Modèles de calcul
Les fonctions récursives

Pour pouvoir définir des compositions variables, nous utiliserons la notion de récursion.

Voici une définition récursive de la fonction factorielle :

$$0! = 1$$
$$(n+1)! = (n+1) \times n!$$

Cette définition est effective car $f(n+1)$ est défini en terme de $f(n)$ et d'opérations que l'on sait calculables par ailleurs, et de plus $f(0)$ est trivialement calculable.

Modèles de calcul
Les fonctions récursives

Nous structurons cette partie de la façon suivante :

- étude des fonctions définissables à partir: de fonctions de base, de la composition et de la récursion primitive;

- étude de la limite de cette classe de fonctions;

- généralisation de la notion de récursion primitive;

- équivalence de la nouvelle classe de fonctions avec les machines de Turing.

Modèles de calcul
Les fonctions primitives récursives
Définition

Les fonctions primitives récursives sont le sous-ensemble des fonctions

$$\{N^k \rightarrow N \mid k \geq 0\}$$

qui peuvent être définies à l'aide des fonctions primitives récursives de base, d'une règle de composition et d'une règle de récursion.

Modèles de calcul
Les fonctions primitives récursives
Définition

Les fonctions primitives récursives de base sont:

1. la fonction $\mathbf{0}()$, elle n'a pas d'argument et renvoit la valeur 0;

2. les fonctions de projection $\pi_i^k(n_1, \ldots, n_k)$. La fonction π_i^k renvoit comme valeur le i^{eme} argument parmi k;

3. la fonction successeur $\sigma(n)$ est définie par $\sigma(n) = n + 1$.

Modèles de calcul
Les fonctions primitives récursives
Définition

Nous définissons maintenant de manière formelle la notion de composition:

Soit g une fonction à l arguments et $h_1, \ldots, h_l$ des fonctions à k arguments. Si nous dénotons $n_1, \ldots, n_k$ par $\bar{n}$, alors la *composition* de g et de $h_1, \ldots, h_l$ est la fonction $\boldsymbol{N}^k \rightarrow \boldsymbol{N}$ définie par:

$$f(\bar{n}) = g(h_1(\bar{n}), \ldots, h_l(\bar{n}))$$

Modèles de calcul
Les fonctions primitives récursives
Définition

Nous définissons maintenant de manière formelle la notion de récursion primitive:

Soit g une fonction à k arguments et h une fonction à $k+2$ arguments. Alors la fonction f à $k+1$ arguments telle que :

$$\begin{array}{lcl} f(\bar{n},0) & = & g(\bar{n}) \\ f(\bar{n},m+1) & = & h(\bar{n},m,f(\bar{n},m)) \end{array}$$

est la fonction définie à partir de g et h par *récursion primitive*.

Modèles de calcul
Les fonctions primitives récursives
Définition

Si les fonctions g et h qui sont utilisées pour définir une fonction f par récursion primitive sont calculables par une procédure effective, alors f est aussi calculable.

Les fonctions primitives récursives sont finalement définies comme suit :

- les fonctions primitives récursives de base;

- toutes les fonctions obtenues à partir des fonctions primitives de base par un nombre quelconque d'applications de la composition et de la récursion primitive.

Modèles de calcul
Les fonctions primitives récursives
Exemples

Etudions maintenant quelques exemples de fonctions primitives récursives.

Bien entendu, toutes les constantes naturelles sont définissables à l'aide de fonctions primitives récursives: $\mathbf{j}() = j$ est définit grâce à la composition de la façon suivante :

$$\mathbf{j}() = \underbrace{\sigma(\sigma(\cdots\sigma}_{j}(\mathbf{0}())\cdots))$$

La fonction d'addition $plus(n_1, n_2)$ peut être définie par récursion primitive :

$$\begin{array}{lcl} plus(n_1, 0) & = & \pi_1^1(n_1) \\ plus(n_1, n_2 + 1) & = & \sigma(\pi_3^3(n_1, n_2, plus(n_1, n_2))) \end{array}$$

Modèles de calcul
Les fonctions primitives récursives
Exemples

Pour alléger les notations, nous laisserons implicites des notations comme $\pi_1^1(n)$ (la fonction identité)... La définition de *plus* devient:

$$\begin{array}{lcl} plus(n_1, 0) & = & n_1 \\ plus(n_1, n_2 + 1) & = & \sigma(plus(n_1, n_2)) \end{array}$$

Modèles de calcul
Les fonctions primitives récursives
Exemples

Voici un exemple d'évaluation d'une fonction primitive récursive:

$$\begin{aligned} plus(7,4) &= plus(7,3+1) \\ &= \sigma(plus(7,3)) \\ &= \sigma(\sigma(plus(7,2))) \\ &= \sigma(\sigma(\sigma(plus(7,1)))) \\ &= \sigma(\sigma(\sigma(\sigma(plus(7,0))))) \\ &= 11 \end{aligned}$$

Modèles de calcul
Les fonctions primitives récursives
Exemples

La fonction "produit de deux naturels" est primitive récursive :

$$\begin{array}{lcl} prod(n_1, 0) & = & 0 \\ prod(n_1, n_2 + 1) & = & plus(n_1, prod(n_1, n_2)) \end{array}$$

ainsi que la fonction "puissance (n^m)" :

$$\begin{array}{lcl} n_1^0 & = & 1 \\ n_1^{n_2+1} & = & n_1 \times n_1^{n_2} \end{array}$$

Modèles de calcul
Les fonctions primitives récursives
Exemples

La fonction factorielle est également primitive récursive :

$$\begin{aligned} 0! &= 1 \\ (n+1)! &= (n+1) \times n! \end{aligned}$$

Modèles de calcul
Les fonctions primitives récursives
Exemples

Considérons maintenant la fonction "prédécesseur". Vu que nous travaillons dans les naturels, nous définissons -1 de la façon suivante:

$$pred(m) = \begin{cases} 0 & \text{si } m = 0 \\ m - 1 & \text{si } m > 0 \end{cases}$$

Cette fonction est primitive récursive:

$$\begin{aligned} pred(0) &= 0 \\ pred(n+1) &= n \end{aligned}$$

Modèles de calcul
Les fonctions primitives récursives
Exemples

On peut maintenant généraliser la définition précédente pour obtenir une définition de la "différence" :

$$n \dot{-} m = \begin{cases} 0 & \text{si } n < m \\ n - m & \text{si } n \geq m \end{cases}$$

Cette fonction est primitive récursive :

$$\begin{array}{lcl} n \dot{-} 0 & = & n \\ n \dot{-} (m+1) & = & pred(n \dot{-} m) \end{array}$$

Modèles de calcul
Les fonctions primitives récursives
Exemples

On définit la fonction “signe” de la façon suivante :

$$sg(m) = \begin{cases} 0 & \text{si } m = 0 \\ 1 & \text{si } m > 0 \end{cases}$$

Cette fonction est primitive récursive :

$$\begin{aligned} sg(0) &= 0 \\ sg(m+1) &= 1 \end{aligned}$$

Modèles de calcul
Les fonctions primitives récursives
Exemples

On définit le "produit borné par m" d'une fonction $g(\bar{n}, i)$ comme le produit de g pour les m premières valeurs de i :

$$\prod_{i=0}^{m} g(\bar{n}, i)$$

Si g est primitive récursive, la fonction

$$f(\bar{n}, m) = \prod_{i=0}^{m} g(\bar{n}, i)$$

est aussi primitive récursive :

$$\begin{array}{lcl} f(\bar{n}, 0) & = & g(\bar{n}, 0) \\ f(\bar{n}, m+1) & = & f(\bar{n}, m) \times g(\bar{n}, m+1) \end{array}$$

Modèles de calcul
Les fonctions primitives récursives
Prédicats

Un prédicat d'arité k est une fonction de $\boldsymbol{N}^k$ dans $\{vrai, faux\}$ ou $\{0, 1\}$.

Mais pour ne pas introduire l'ensemble de valeurs $\{vrai, faux\}$, on peut également considérer un prédicat d'arité k comme un sous-ensemble de $\boldsymbol{N}^k$ (le sous-ensemble des éléments de $\boldsymbol{N}^k$ où le prédicat est vrai).

Par exemple, nous pouvons définir le prédicat *pair*: $pair(n)$ est vrai si et seulement si n est un nombre pair. Est-ce que ce prédicat est primitif récursif ?

Modèles de calcul
Les fonctions primitives récursives
Prédicats

Pour donner une définition de *prédicat primitif récursif*, nous allons profiter du fait qu'un prédicat peut-être vu comme une fonction (sa fonction caractéristique) dans $\{0, 1\}$: nous déduirons de notre notion de fonction primitive récursive celle de prédicat primitif récursif.

Définissons formellement la notion de fonction caractéristique. La *fonction caractéristique* d'un prédicat $P \subseteq \boldsymbol{N}^k$ est la fonction $f : \boldsymbol{N}^k \to \{0, 1\}$ telle que :

$$f(\bar{n}) = \begin{cases} 0 & \text{si } \bar{n} \notin P. \\ 1 & \text{si } \bar{n} \in P. \end{cases}$$

Un prédicat est *primitif récursif* si et seulement si sa fonction caractéristique est primitive récursive.

Modèles de calcul
Les fonctions primitives récursives
Prédicats - Exemples

Considérons le prédicat *zerop*, qui n'est vrai que pour la valeur 0. Sa définition est la suivante :

$$\begin{aligned} zerop(0) &= 1 \\ zerop(n+1) &= 0 \end{aligned}$$

On définit le prédicat *plus petit que* ($n < m$) de la façon suivante :

$$petit(n, m) = sg(m \dot{-} n)$$

Ce prédicat est donc bien primitif récursif.

Modèles de calcul
Les fonctions primitives récursives
Prédicats - Exemples

Les prédicats obtenus par combinaisons booléennes à partir de prédicats primitifs récursifs (g_1, g_2) sont également primitifs récursifs. En effet :

$$\begin{aligned} et(g_1(\bar{n}), g_2(\bar{n})) &= g_1(\bar{n}) \times g_2(\bar{n}) \\ ou(g_1(\bar{n}), g_2(\bar{n})) &= sg(g_1(\bar{n}) + g_2(\bar{n})) \\ non(g_1(\bar{n})) &= 1 \dot{-} g_1(\bar{n}) \end{aligned}$$

Le prédicat *égalité* est également primitif récursif. En effet, $m = n$ équivaut à $\neg(m < n \vee m > n)$, ce qui s'écrit au niveau des fonctions caractéristiques :

$$egal(n, m) = 1 \dot{-} (sg(m \dot{-} n) + sg(n \dot{-} m))$$

Il est facile maintenant de monter que les autres prédicats usuels $\leq, \geq, \neq, \ldots$ sur les naturels sont également primitifs récursifs.

Modèles de calcul
Les fonctions primitives récursives
Prédicats - Exemples

Observons maintenant d'autres opérations logiques. Intéressons-nous à la quantification bornée.

La *quantification universelle bornée* permet d'exprimer qu'un prédicat est vrai pour toutes les valeurs inférieures à une certaine borne.

$$\forall i \leq m \cdot p(\overline{n}, i)$$

est vrai si $p(\overline{n}, i)$ est vrai pour tout $i \leq m$.

La *quantification existentielle bornée* permet d'exprimer qu'un prédicat est vrai pour au moins une valeur inférieure à une certaine borne.

$$\exists i \leq m \cdot p(\overline{n}, i)$$

est vrai si $p(\overline{n}, i)$ est vrai pour au moins un $i \leq m$.

Modèles de calcul
Les fonctions primitives récursives
Prédicats - Exemples

Montrons que ces deux formes de quantifications sont primitives récursives.

La fonction caractéristique de $\forall i \leq m \cdot p(\bar{n}, i)$ est donnée par :

$$\prod_{i=0}^{m} p(\bar{n}, i)$$

Vu que $\exists i \leq m \cdot p(\bar{n}, i) \equiv \neg \forall i \leq m \cdot \neg p(\bar{n}, i)$, la fonction caractéristique de $\exists i \leq m \cdot p(\bar{n}, i)$ est donnée par :

$$1 \dot{-} \prod_{i=0}^{m} (1 \dot{-} p(\bar{n}, i))$$

Modèles de calcul
Les fonctions primitives récursives
Prédicats - Exemples

Considérons maintenant les fonctions définies par "étude de cas". Soit f la fonction définie par :

$$f(\bar{n}) = \begin{cases} g_1(\bar{n}) & \text{si } p_1(\bar{n}). \\ . & \\ . & \\ . & \\ g_l(\bar{n}) & \text{si } p_l(\bar{n}). \end{cases}$$

si les fonctions $g_1, \ldots, g_l$ et les prédicats $p_1, \ldots, p_l$ sont primitifs récursifs, alors f est primitive récursive.

En effet, si les p_i sont mutuellement exclusifs (ils doivent l'être pour que la définition ci-dessus ait un sens), on a :

$$f(\bar{n}) = g_1(\bar{n}) \times p_1(\bar{n}) + \ldots + g_l(\bar{n}) \times p_l(\bar{n})$$

Modèles de calcul
Les fonctions primitives récursives
Prédicats - Exemples

On définit également la *minimisation bornée*. Ce mécanisme permet de définir une fonction à partir d'un prédicat.

Soit $q(\overline{n}, i)$, un prédicat. Pour un m donné, la valeur que définit la fonction obtenue par *minisation bornée* de ce prédicat est la plus petit valeur $i \leq m$ telle que $q(\overline{n}, i)$ est vrai. Si cette valeur n'existe pas, la fonction renvoit, par convention, la valeur 0.

La minisation bornée d'un prédicat $q(\overline{n}, i)$ est notée :

$$\mu i \leq m \cdot q(\overline{n}, i)$$

Modèles de calcul
Les fonctions primitives récursives
Prédicats - Exemples

La fonction $\mu i \leq m \cdot q(\bar{n}, i)$ est définie par récursion primitive de la façon suivante :

$$\mu i \leq 0 \cdot q(\bar{n}, i) = 0$$

$$\mu i \leq m+1 \cdot q(\bar{n}, i) = \begin{cases} 0 \\ \quad \text{si } \neg\exists i \leq m+1 \cdot q(\bar{n}, i) \\ \mu i \leq m \cdot q(\bar{n}, i) \\ \quad \text{si } \exists i \leq m \cdot q(\bar{n}, i) \\ m+1 \\ \quad \text{si } q(\bar{n}, m+1) \\ \quad\quad \text{et } \neg\exists i \leq m \cdot q(\bar{n}, i) \end{cases}$$

Modèles de calcul
Les fonctions primitives récursives
Limites

On vient de voir que les fonctions primitives récursives permettent de définir les fonctions et les prédicats les plus usuels. On pourrait maintenant se demander si les fonctions primitives récursives couvrent bien la notion intuitive que nous avons de fonctions calculables?

On va malheureusement montrer que ce n'est pas le cas ! On va montrer qu'il existe des fonctions qui sont intuitivement calculables et qui ne sont pas primitives récursives.

Modèles de calcul
Les fonctions primitives récursives
Limites

Il est évident qu'il existe des fonctions qui ne sont pas primitives récursives. En effet, les fonctions primitives récursives sont énumérables (on peut énumérer les chaînes de caractères qui les représentent) il y a en a donc un nombre dénombrable alors que, souvenez-vous, il y a un nombre non dénombrable de fonctions.

Mais nous allons voir qu'il existe également des fonctions qui sont calculables mais pas primitives récursives.

L'argument que nous allons développer est à nouveau basé sur la méthode de la diagonale.

Modèles de calcul
Les fonctions primitives récursives
Limites

Théorème : il existe des fonctions calculables qui ne sont pas primitives récursives.

Preuve : notons tout d'abord que l'on peut effectivement énumérer les fonctions primitives récursives. Soit $f_0, f_1, \ldots, f_n, \ldots$, une telle énumération. Construisons à partir de cette énumération une fonction calculable F qui n'est pas primitive récursive. Nous définissons F de la façon suivante :

$$F(n) = f_n(n) + 1$$

Par construction la fonction F est différente de toutes les fonctions primitives récursives. Montrons maintenant qu'elle est calculable. Pour cela, on doit établir une méthode de calcul qui

soit valable pour calculer $F(n)$ pour n'importe quel n. La méthode pour calculer $F(n)$ est la suivante :

1. énumérer les fonctions primitives récursives jusque la fonction f_n;

2. évaluer $f_n(n)$;

3. ajouter 1 à la valeur obtenue au pt 2.

Il important de noter que la démonstration est valide car les fonctions primitives récursives sont effectivement énumérables et de plus l'évaluation d'une fonction primitive récursive **termine toujours**.

Modèles de calcul
Les fonctions μ-récursives

Les fonctions μ-récursives étendent les fonctions primitives récursives en permettant la *minimisation non-bornée*.

La minimisation non bornée d'un prédicat $q(\overline{n}, i)$ est une fonction que l'on note

$$\mu i \cdot q(\overline{n}, i)$$

et la définition de cette fonction est :

$$\mu i \cdot q(\overline{n}, i) =$$
$$\begin{cases} \text{le plus petit } i \text{ tel que } q(\overline{n}, i) = 1 \\ 0 \text{ si un tel } i \text{ n'existe pas} \end{cases}$$

Modèles de calcul
Les fonctions μ-récursives

La minisation non bornée ne produit pas des fonctions primitives récursives. Elle peut même produire des fonctions qui ne sont pas calculables.

En effet, la méthode naturelle pour déterminer la valeur renvoyée par $\mu i \cdot q(\overline{n}, i)$ est d'évaluer $q(\overline{n}, i)$ pour des valeurs croissantes de i. Cette évaluation peut ne jamais terminer si il n'existe pas de i tel que $q(\overline{n}, i) = 1$!

Modèles de calcul
Les fonctions μ-récursives

Nous voulions étendre les fonctions récursives mais nous voulons garder des fonctions qui soient calculables. Pour parvenir à cela, nous avons desoin d'une nouvelle notion.

Un prédicat $q(\overline{n}, i)$ est dit *sûr* si

$$\forall \overline{n} \cdot \exists i \cdot q(\overline{n}, i) = 1.$$

Nous pouvons maintenant définir les fonctions μ-récursives.

Modèles de calcul
Les fonctions μ-récursives

Les fonctions et les prédicats μ-récursifs sont ceux obtenus à partir des fonctions primitives récursives de base par :

- composition;
- récursion primitive;
- minimisation non bornée de prédicats sûrs.

Modèles de calcul
Les fonctions μ-récursives

Il est clair que les fonctions μ-récursives sont calculables dans le sens intuitif du terme (et ce seulement car on applique la minimisation non bornée à des prédicats sûrs).

On peut maintenant se poser à nouveau la question :

Est-ce qu'il existe des fonctions calculables qui ne soient pas exprimables sous forme de fonctions μ-récursives ?

Modèles de calcul
Les fonctions μ-récursives
ef fonctions calculables

Dans une partie précédente, nous avons énoncé la thèse de Church: "les fonctions calculables par une procédure effective sont celles calculables par une machine de Turing".

Nous allons montrer ici que les fonctions μ-récursives couvrent exactement les fonctions calculables par une machine de Turing.

C'est un argument fort pour la thèse de Church: deux formalismes fondamentalement différents, proposés pour formaliser la notion de procédure effective, identifient exactement la même classe de fonctions.

Modèles de calcul
Les fonctions μ-récursives
ef fonctions calculables

Comment établir cette équivalence ?

Il y a un premier obstacle : les machines de Turing manipulent des chaînes de caractères alors que les fonctions récursives manipulent des nombres entiers.

On va montrer qu'on peut coder chaque nombre entier comme un mot sur un alphabet fini et que l'on peut faire correspondre à chaque chaîne de caractère un entier qui la représente.

Modèles de calcul
Les fonctions μ-récursives
ef fonctions calculables

Lemme. Il existe une représentation effective des nombres naturels par des chaînes de caractères.

Les représentations binaires et décimales sont des exemples de telles représentations. Ces représentations sont effectives car il existe des fonctions μ-récursives (en fait primitives récursives) qui étant donné un nombre naturel m, permettent de calculer la longueur de sa représentation binaire ($\log_2 m$) ainsi que pour $n \leq \log_2 m$, le n^e caractère de sa représentation binaire.

Le contraire est également vrai.

Lemme. Il existe une représentation effective des chaînes de caractères par les naturels.

Ce passage d'une chaîne de caractère à un nombre naturel porte le nom de "Godelisation".

Montrons comment cela est possible.

Soit Σ l'alphabet sur lequel sont définis les mots. Supposons $|\Sigma| = k$. On choisit d'abord une représentation de chaque symbole $a \in \Sigma$ par un entier entre 1 et k, qu'on note $gd(a)$. L'encodage d'une chaîne $w = w_0 w_1 \dots w_l$ est alors:

$$gd(w) = \sum_{i=0}^{l} k^{l-i} gd(w_i)$$

Il est facile de se convaincre que cette représentation est effective.

Modèles de calcul
Des fonctions μ-récursives
aux machines de Turing

Théorème. Toute fonction μ-récursive est calculable par une machine de Turing.

Preuve. Nous n'irons pas dans le détail de la démonstration. Il suffit de se convaincre qu'il est possible d'écrire dans un langage usuel comme PASCAL un algorithme qui interprète les fonctions μ-récursives quand leur arguments sont donnés en encodage binaire par exemple.

Bien entendu, il est important que la minimisation bornée soit appliquée seulement à des prédicats sûrs.

Modèles de calcul
Des machines de Turing
aux fonctions μ-récursives

Théorème. Toute fonction calculable par machine de Turing est μ-récursive.

Preuve. Soit M une machine de Turing d'alphabet Σ calculant la fonction $f_M : \Sigma^* \to \Sigma^*$ et soit gd un encodage des élément de Σ^* par des naturels. Nous montrons qu'il existe une fonction μ-récursive f telle que

$$f_M(w) = gd^{-1}(f(gd(w)))$$

Nous définissons les fonctions suivantes :

- $init(x)$ qui donne la configuration initiale de M pour le mot d'entrée w de nombre de Godel x c'est-à-dire que $x = gd(w)$;

- $ConfigSuiv(x)$ qui renvoit la configuration suivante de M si la configuration actuelle est x;

- $Config(x, n)$ qui donne la configuration atteinte après n pas à partir de x. Cette fonction est définie par récursion :

$$\begin{cases} Config(x, 0) & = & x \\ Config(x, n+1) & = & ConfigSuiv(Config(x, n)) \end{cases}$$

- $Stop(x) = \begin{cases} 1 & \text{si } x \text{ final} \\ 0 & \text{sinon} \end{cases}$

- $Sortie(x)$ renvoit pour une configuration finale la valeur calculée par la machine de Turing (l'encodage du contenu du ruban)

Il suffit de se convaincre que ces fonctions sont μ-récursives (elles sont en fait primitives récursives).

La fonction μ-récursive f correspondant à la fonction f_M est alors :

$$f(x) = Sortie(Config(init(x), NbDePas(x)))$$

où

$$NbDePas(x) = \mu i \cdot Stop(Config(Init(x), i))$$

Modèles de calcul
Les fonctions partielles

Les fonctions partielles sont des fonctions qui sont définies pour certaines valeurs de leurs arguments et non pour d'autres.

Par exemple, la *division entière* est une fonction partielle.

On dira qu'une fonction partielle $f = \Sigma^* \rightarrow \Sigma^*$ est calculée par une machine de Turing M si,

- pour tout mot d'entrée w pour lequel f est définie, M s'arrête dans une configuration où $f(w)$ se trouve sur le ruban;

- pour tout mot d'entrée w pour lequel f n'est pas définie, M ne s'arrête pas ou

s'arrête en signalant, par une valeur conventionnelle écrite sur le ruban, que la fonction n'est pas définie.

Une fonction partielle calculable est alors une fonction partielle calculée par une Machine de Turing.

Modèles de calcul
Les fonctions partielles

Une fonction partielle $f = \Sigma^* \rightarrow \Sigma^*$ est μ-récursive si elle peut-être définie à partir des fonctions primitives récursives de base par

- composition,
- récursion primitive,
- minimisation non bornée.

La minimisation non bornée pouvant être appliquée à un prédicat non sûr. La fonction $\mu i \cdot q(\bar{n}, i)$ n'est définie que si il existe un i tel que $q(\bar{n}, i) = 1$.

Modèles de calcul
Les fonctions partielles

On peut montrer à nouveau l'équivalence de ces deux nouvelles notions :

Théorèmes. Une fonction partielle est μ-récursive si et seulement si elle est calculable par une machine de Turing.

Décidabilité - Indécidabilité
Introduction

Rappelons que nous avons établi que :

- problème = langage;
- algorithme (procédure effective) = machine de Turing;
- il y a plus de langages que de machines de Turing.

Donc il y a des langages qui ne peuvent pas être décidés par une machine de Turing. En d'autres termes, il existe des problèmes qui ne sont pas solubles par une procédure effective.

Décidabilité - Indécidabilité
Introduction

Mais peut-on “concrétiser” ce résultat ?

Peut-on énoncer un problème particulier qui ne soit pas décidable ?

C’est l’objet de ce chapitre !

On va en effet donner des exemples de problèmes qui sont indécidables.

Décidabilité - Indécidabilité
Introduction

La démarche que nous utiliserons dans ce chapitre est la suivante :

- on démontrera tout d'abord l'indécidabilité de deux problèmes grâce à la méthode de la diagonale;

- ensuite, nous utiliserons le principe de la réduction pour démontrer l'indécidabilité d'autres problèmes.

Décidabilité - Indécidabilité
Introduction

Ce chapitre sera organisé de la façon suivante:

- deux problèmes indécidables (deux niveaux d'indécidabilité);
- technique de la réduction;
- réduction : applications;
- propriétés des langages récursivement énumérables;
- d'autres problèmes indécidables et théorème de Rice;
- fonctions non-calculables.

Décidabilité - Indécidabilité
Problèmes et classes de décidabilité

Problème = langage de l'encodage de ses instances positives.

Rem : dans la suite nous nous permettrons de ne pas fixer explicitement l'encodage.

Nous noterons $\mathbf{R}$ l'ensemble des langages qui sont décidés par une machine de Turing. On notera donc $L \in \mathbf{R}$ le fait que L (le langage, le problème) soit décidable.

Décidabilité - Indécidabilité
Problèmes et classes de décidabilité

Nous étudirons également une autre classe de langages : la classe $\mathbf{RE}$ des langages récursivement énumérables. C'est la classe des langages qui sont **acceptés** par une machine de Turing.

A priori la classe $\mathbf{RE}$ ne contient pas que des langages décidables.

Néanmoins, intuitivement $\mathbf{RE}$ est proche de la décidabilité d'où son intérêt.

Bien entendu nous avons que $\mathbf{R} \subseteq \mathbf{RE}$.

Décidabilité - Indécidabilité
Problèmes et classes de décidabilité

Synonymes :

- les langages de la classe $\mathbf{R}$ sont appelés : récursifs, décidables, calculables, ou encore solubles algorithmiquement;

- les langages de la classe $\mathbf{RE}$ sont appelés : partiellement récursifs, partiellement décidables, partiellement calculables,partiellement solubles algorithmiquement, ou encore récursivement énumérables;

Décidabilité - Indécidabilité
Un premier langage indécidable

Avec la méthode de la diagonale, nous allons montrer l'existence d'un langage L_0 tel que $L_0 \notin \mathbf{RE}$.

Notons qu'on aura tout de suite que $L_0 \notin \mathbf{R}$ (mais on aura montrer quelque chose de plus fort).

Les machines de Turing et les mots finis sont dénombrables. On peut donc construire un tableau infini A qui contient en ligne une énumération $M_0, M_1, \ldots, M_n, \ldots$ des machines de Turing et en colonne $w_0, w_1, \ldots, w_n, \ldots$ une énumération des mots.

Décidabilité - Indécidabilité
Un premier langage indécidable

Les cases du tableau ont les valeurs suivantes:

- $A[i,j] = O$ si la machine M_i accepte le mot w_j;

- $A[i,j] = N$ si la machine M_i n'accepte pas le mot w_j (s'arrête et rejette, ou boucle).

Nous définissons le langage

$$L_0 = \{w \mid w = w_i \wedge A[i,i] = N\}.$$

Décidabilité - Indécidabilité
Un premier langage indécidable

Théorème : Le langage L_0 n'appartient pas à la classe $\mathbf{RE}$.

Preuve : faisons l'hypothèse que le langage $L_0 \in \mathbf{RE}$. Dans ce cas, il existe une machine M_{L_0} qui accepte L_0. Montrons que ce n'est pas possible. En effet, cette machine M_{L_0} doit appartenir à l'énumération. Faisons l'hypothèse qu'elle a le numéro j ($M_{L_0} = M_j$). Par définition on a $w_j \in L_0$ ssi $A[j, j] = N$. Si $w_j \in L_0$, on obtient une contradiction car on a que w_j n'est pas accepté par M_j et donc n'est pas accepté par M_{L_0} ce qui n'est pas possible. De même si $w_j \notin L_0$, on obtient une contradiction car on a que w_j est accepté par M_j et donc accepté par M_{L_0}. Dans les deux cas, on obtient une contradiction.

Décidabilité - Indécidabilité
Un deuxième langage indécidable

L_0 n'est pas décidable et il n'est pas non plus "partiellement décidable".

Nous cherchons maitenant un langage L tel que $L \in \mathbf{RE}$ mais $L \notin \mathbf{R}$. Nous cherchons donc un problème qui se trouve entre la décidabilité et l'indécidabilité : qui est indécidable mais partiellement décidable.

Notons que c'est important d'avoir un problème le plus "faible" possible pour pouvoir appliquer la méthode de la réduction.

Décidabilité - Indécidabilité
Un deuxième langage indécidable

Nous avons besoin de quelques résultats intermédiaires.

Lemme. Le complément d'un langage de la classe $\mathbf{R}$ est un langage de la classe $\mathbf{R}$.

Preuve. Si $L \in \mathbf{R}$ alors il est décidé par une machine de Turing M. A partir de M, il suffit de construire une machine M' telle que M' répond oui ssi M répond non. (Bien entendu cette construction ne marche que parce qu'on sait que M n'a pas d'exécution infinie).

Décidabilité - Indécidabilité
Un deuxième langage indécidable

Lemme. Si $L \in \mathbf{RE}$ et $\bar{L} \in \mathbf{RE}$ alors $L \in \mathbf{R}$ et $\bar{L} \in \mathbf{R}$.

Preuve. Notons que, vu le lemme précédent, nous devons seulement montrer que $L \in \mathbf{RE}$ et $\bar{L} \in \mathbf{RE}$ implique $L \in \mathbf{R}$. Si $L, \bar{L} \in \mathbf{RE}$, il existe M_L et $M_{\bar{L}}$ qui acceptent L et $\bar{L}$. Pour décider L, nous construisons une machine M' qui simule en parallèle M_L et $M_{\bar{L}}$. M' s'arrête lorsque soit M_L, soit $M_{\bar{L}}$ s'arrête. M' accepte exactement dans les deux cas suivant :

- M_L provoque l'arrêt et accepte;
- $M_{\bar{L}}$ provoque l'arrêt et rejette.

Notons que nous sommes assurés que M' s'arrêtera sur tout mot w car soit $w \in L$, et alors M_L doit s'arrêter après un nombre fini de pas, soit $w \notin L$ et donc $w \in \bar{L}$ et $M_{\bar{L}}$ s'arrête après un nombre fini de pas.

Décidabilité - Indécidabilité
Un deuxième langage indécidable

Une conséquence du lemme précédent est que, par rapport aux classes $\mathbf{R}$ et $\mathbf{RE}$, la position d'une langage L doit correspondre à un des trois cas suivants :

- L et $\bar{L} \in \mathbf{R}$;
- $L \notin \mathbf{RE}$ et $\bar{L} \notin \mathbf{RE}$;
- $L \notin \mathbf{RE}$ et $\bar{L} \in \mathbf{RE} \cap \bar{\mathbf{R}}$.

Le troisième cas nous donne un moyen de trouver un langage récursivement énumérable et non décidable : il suffit de trouver un langage qui est dans $\mathbf{RE}$ et dont le complément n'est pas dans $\mathbf{RE}$.

Décidabilité - Indécidabilité
Un deuxième langage indécidable

Montrons que le complément du langage L_0 est un tel langage.

Lemme. Le langage

$$\overline{L_0} = \{w \mid w = w_i \wedge M_i \text{ accepte } w_i\}$$

est dans la classe $\mathbf{RE}$.

Preuve. La preuve est constructive : on montre qu'on peut construire une machine de Turing qui accepte $\overline{L_0}$. Cette machine procède de la façon suivante pour un mot d'éntrée w :

- elle détermine i tel que $w_i = w$. Pour cela il suffit d'énumérer les mots w_i et vérifier l'égalité;

- elle détermine M_i (énumération systématique jusque la i^e machine). Pour énumérer les machines, il suffit d'énumérer les sequences des symboles et pour chaque reconnaitre (critère syntaxique) si la séquence encode une machine. On peut ainsi déterminer la i^e machine.

- simuler l'exécution de M_i sur w_i et accepte si M_i accepte w_i.

Décidabilité - Indécidabilité
Un deuxième langage indécidable

On a donc établit le résultat suivant :

Théorème. Le langage $\overline{L_0}$ défini par $\overline{L_0} = \{w \mid w = w_i \wedge M_i \text{ accepte } w_i\}$ est indécidable (n'appartient pas à $\mathbf{R}$) mais appartient à $\mathbf{RE}$.

Décidabilité - Indécidabilité
Technique de la réduction

Nous venons d'établir l'existence de deux problèmes indécidables. Comment pouvons-nous exploiter ces deux résultats pour montrer que d'autres problèmes sont indécidables ?

Une technique permet de monter que d'autres problèmes sont indécidables, c'est la technique de la **réduction**.

Cette technique nous permettra de démontrer l'indécidabilité de problèmes plus "naturels" que L_0 et $\overline{L_0}$.

Décidabilité - Indécidabilité
Technique de la réduction

La technique de la réduction permet, connaissant l'indécidabilité d'un problème L_1, de montrer l'indécidabilité d'un problème L_2 en raisonnant de la façon suivante :

- on démontre que si il existe un algorithme qui décide le problème L_2 alors il existe un algorithme qui décide L_1. Cela se fait en donnant un *algorithme* pour L_1 qui utilise un algorithme pour L_2 comme sous-programme. Ce type d'algorithme est appelé une *réduction* car il réduit la décidabilité de L_1 à celle de L_2.

- on déduit que L_2 est non décidable car sa décidabilité nous obligerait à conclure à la décidablité de L_1 qu'on sait indécidable.

Décidabilité - Indécidabilité
Réduction : applications

Théorème : le langage universel

$$LU = \{\langle M, w\rangle \mid M \text{ accepte } w\}$$

est indécidable.

Preuve. Nous utilisons la méthode de la réduction. Faisons l'hypothèse que LU est décidable et montrons que sous cette hypothèse, nous pouvons construire une machine de Turing qui décide $\overline{L_0}$.

Pour rappel

$$\overline{L_0} = \{w \mid w = w_i \wedge M_i \text{ accepte } w_i\}$$

Décidabilité - Indécidabilité
Réduction : applications

L'algorithme (hypothétique) pour décider si $w \in \overline{L_0}$ est le suivant :

- déterminer l'indice i tel que $w = w_i$;
- déterminer la machine M_i;
- utiliser la procédure pour décider si $\langle M_i, w_i \rangle \in LU$, si le résultat est positif, w est accepté, sinon il est rejetté.

Nous devons donc rejeter la décidabilité de LU.

Remarquons que $\overline{LU}$ est également indécidable mais aussi non récursivement énumérable (vu que LU est récursivement énumérable).

Décidabilité - Indécidabilité
Problèmes indécidables

Montrons maintenant quelques autres problèmes indécidables. Un des plus connus est le problème de l'arrêt.

Théorème. Le langage

$$H = \{\langle M, w\rangle \mid M \text{ s'arrête sur } w\}$$

n'est pas récursif.

Preuve. Montrons que l'existence d'une procédure de décision pour H impliquerait l'existence d'une procédure de décision de LU (ce qui est impossible, nous venons de le montrer).

L'algorithme de décision pour LU serait :

- décider (avec l'algorithme hypothétique) si $\langle M, w \rangle \in H$;

- si la réponse obtenue au point précédent est négative, répondre non; par contre si la réponse obtenue est positive, on simule l'execution de M sur w et on répond ce que M répond.

Dans les deux cas on est assuré de la terminaison (Pq ?).

Décidabilité - Indécidabilité
Problèmes indécidables

Le problème de l'arrêt est également indécidable pour les langages usuels (comme PASCAL).

Pour établir ce résultat on procède par réduction au problème de l'arrêt des machines de Turing:

- construire un programme PASCAL P qui, étant donné une machine de Turing M et un mot w, simule le comportement de M sur w;

- utiliser la procédure de décision pour savoir si P s'arrête pour les données $\langle M, w \rangle$.

Décidabilité - Indécidabilité
Problèmes indécidables

Un grand nombre de variantes du problème de l'arrêt sont indécidables.

Exercices. Montrez que :

- déterminer si une machine de Turing s'arrête lorsque qu'elle est lancé sur le mot vide est indécidable;

- déterminer si une machine de Turing M s'arrête au moins sur un mot w (problème de l'arrêt existentiel) est indécidable;

- déterminer si une machine de Turing M s'arrête sur tous les mots w (problème de l'arrêt universel) est indécidable.

Décidabilité - Indécidabilité
Langages énumérés
par une procédure effective

Nous avons vu que les langages acceptés par une machine de Turing sont dits *"récursivement énumérables"*. Donc intuitivement, si un langage L est un langage accepté par une machine de Turing M alors il doit exister une **procédure effective** qui énumère ce langage.

Comment énumérer un tel langage ?

En première approximation, il suffirait de :

- énumérer par ordre lexicographique (par exemple) les mots de Σ^*;

- simuler l'exécution de la machine M sur chacun de ces mots et conserver le mot si M accepte celui-ci.

Décidabilité - Indécidabilité
Langages énumérés
par une procédure effective

Ce schéma simpliste ne marche malheureusement pas. En effet, dès qu'un mot est rejetté par M pour cause "d'exécution infinie", la procédure "bloque".

Il faut une solution plus *astucieuse* : considérons les pairs (w, n) où $w \in \Sigma^*$ et $n \in \mathbb{N}$. Ces pairs sont énumérables. On considère les pairs (w, n) dans l'ordre de leur énumération et on effectue le traitement suivant : simuler l'exécution des n premiers pas d'exécution de M sur w. Si M accepte w en n pas alors on produit w.

Décidabilité - Indécidabilité
Langages énumérés
par une procédure effective

Notons que d'une manière générale :

Thm : une énumération dans l'ordre lexicographique est impossible.

Preuve : faisons l'hypothèse du contraire. Soit L un langage récursivement énumérable (et pas récursif) énuméré dans l'ordre lexicographique (par exemple) par M. Pour décider si $w \in L$ (ce qui n'est pas possible en général vu que L est non récursif), il suffit d'exécuter M jusqu'au moment où soit w sort, et donc on sait que $w \in L$ ou on dépasse w dans l'ordre lexicographique et on sait que $w \notin L$.

Décidabilité - Indécidabilité
Quelques autres problèmes indécidables

Les problèmes de validité et satisfiabilité dans la logique du premier ordre sont indécidables. C'est-à-dire que déterminer si une formule est satisfaite par toutes les structures du premier-ordre ou déterminer si il existe au moins une structure du premier ordre qui satisfait une formule sont des problèmes indécidables.

Notons que l'ensemble des formules du premier ordre qui sont valides sont récursivement énumérables (théorème de complétude de Goedel) et donc par conséquent, l'ensemble des formules non valides du premier ordre ne sont pas récursivement énumérables. (pourquoi ?)

Décidabilité - Indécidabilité
Quelques autres problèmes indécidables

Le *dixième problème de Hilbert* est indécidable. Ce problème consiste à déterminer si léquation :

$$p(x_1, \ldots, x_n) = 0$$

où $p(x_1, \ldots, x_n)$ est un polynôme à coefficients entiers a une solution dans le domaine des entiers.

Décidabilité - Indécidabilité
Théorème de Rice

Ce théorème dit que "toute propriété non triviale des langages récursivement énumérables est indécidable".

Théorème : Suppons que C soit un sous-ensemble propre et non vide de **RE**. Alors la question de savoir si, étant donnée une machine de Turing M, $L(M) \in C$ est indécidable.

Preuve : Faisons l'hypothèse que $\emptyset \notin C$ (sinon considérer le complément de C). De plus, vu que C est non vide (C est une propriété non triviale), nous pouvons faire l'hypothèse qu'il existe un langage $L \in C$, accepté par une machine M_L (C est un ensemble de langages récursivement énumérables et donc L est accepté par une machine de Turing).

Considérons une machine M et un mot w. Faisons l'hypothèse, sans perte de généralité, que M diverge sur w si $w \notin L(M)$.

Supposons que nous voulons **décider** si M s'arrête sur w (ce qui est impossible).

Pour cela, nous construisons une machine M_w dont le langage est soit L, soit $\emptyset$.

Dans un premier temps, sur l'entrée y, M_w simule M sur w. Si $M(w) =$"oui", alors M_w, à la place d'arrêter, continue et simule M_L sur l'input y : soit $y \in L$ et la simulation de M_L s'arrête et donc M_w accepte y, soit $y \notin L$ et la simulation de M_L sur y diverge et donc M_w diverge sur y.

On a donc :

$M_w(y)$: **si** $M(w) =$"oui" **alors** $M_L(y)$ **sinon** "diverge".

On a donc bien que

$$M_w \text{ accepte le langage } L$$
$$\text{si et seulement si } M \text{ accepte } w$$

En d'autres termes, la construction de M_w à partir de w est une réduction du problème d'arrêt de M sur w au test $L(M_w) = L$ et donc $L(M_w) \in C$.

Donc si on pouvait décider $L(M_w) \in^? C$ alors on pourrait décider le problème de l'arrêt de M sur w !

Toute question non triviale sur un langage récursivement énumérable est donc indécidable.

Complexité
Introduction

Nous venons de voir qu'il existe des problèmes qui n'ont pas de solutions algorithmiques. Nous avons classifié les problèmes en décidables, semi-décidables et indécidables.

Quand un problème est décidable, on peut tout de même se demander si pour le résoudre, il ne faut pas mettre en oeuvre des ressources trop importantes (de temps ou de mémoire, par exemple).

Y a-t-il des problèmes qui sont décidables mais pour lesquels il n'existe pas d'algorithme efficace ?

Complexité
Introduction

La difficultée liée à la résolution d'un problème semble difficile à quantifier.

Notons tout de même qu'il existe des mesures qui permettent d'affirmer qu'un problème est difficile à résoudre. Par exemple, admettons que nous arrivions à monter que pour résoudre certaines instances d'un problème, il faut une mémoire en nombre de bits égale au nombre d'atomes dans l'univers (de l'ordre de 10^{50}) alors on peut certainement annoncer "sans crainte" que le problème ne pourra être résolu par un ordinateur dans toute sa généralité.

Complexité
Introduction

La complexité d'un problème sera mesurée à partir d'une fonction qui relie la taille d'un problème (de son encodage) et les ressources (temps ou mémoire) utilisées pour résoudre le problème. Ces fonctions définissent des classes de complexité.

Toute fonction de $\boldsymbol{N}$ dans $\boldsymbol{N}$ est a priori un candidat pour définir une classe de complexité. Néanmoins, nous nous limitterons à des fonctions qui définissent des classes contenant des problèmes intéressants.

Pour séparer les notions de complexité acceptable et non-acceptable, nous utiliserons la frontière définie par la limite entre les fonctions polynomiales et les fonctions exponentielles.

Complexité
Introduction

Nous dirons donc qu'un problème requiert des *ressources raisonables* si la quantité de ces ressources est bornée par une fonction polynomiale dépendant de la taille du problème.

Inversément, nous dirons qu'un problème requiert des *ressources non-raisonables* si la quantité de ces ressources est bornée par une fonction exponentielle dépendant de la taille du problème.

Par exemple la fonction n^2 est considérée comme acceptable, par contre la fonction 2^n ne l'est pas. (est-il raisonable de considérer n^{100} comme acceptable ?)

Complexité
Introduction

Arguments en faveur de cette frontière :

- nous utiliserons cette frontière pour montrer que certains problèmes **ne** sont **pas** solvables par un algorithme de complexité raisonable;

- souvent lorsqu'un problème a une complexité polynomiale, il possède un algorithme dont la complexité est bornée par un polynôme de faible degré (rarement supérieur à 3 ou 4).

Complexité
Introduction

De plus, on peut montrer que cette frontière est :

- insensible au matériel utilisé (le gap d'efficacité entre, par exemple, une machine de Turing et un algorithme exécuté par une machine moderne est polynomiale);

- insensible à l'encodage utilisé pour représenter des instances du problème (en tout cas encodage raisonable);

- significative en pratique.

Complexité
Introduction

De quelles techniques disposons-nous pour montrer qu'un problème n'est pas soluble efficacement ?

Techniques disponibles :

- la technique de diagonalisation;
- la technique de réduction.

Néanmoins, la diagonalisation ne permet de prouver le résultat escompté. En effet, même si elle permet de monter que certains problèmes n'ont pas de solution polynomiale, elle ne permet de le faire que pour une classe restreinte de problèmes.

Complexité
Introduction

Il existe en effet un grand nombre de problèmes pour lesquels on suspecte qu'il n'existe pas d'algorithmes polynomiaux mais pour lesquels la méthode de la diagonale ne nous permet pas d'en faire la preuve. Nous devrons nous contenter d'un résultat plus faible.

Grâce à la méthode de la réduction nous montrerons pour une classe de problèmes (intéressants en pratique), pour lesquels on n'a jamais trouvé d'algorithme polynomial et pour lesquels on pense qu'il n'en existe pas, que si il existait un algorithme polynomial pour résoudre un seul des problèmes de cette classe alors tous les problèmes de cette classe posséderaient un algorithme polynomial. Cette classe est appelée la classe des problèmes **NP-complets**.

Complexité
Mesurer la complexité
Notions générales

Dans un premier temps, nous nous intéresserons principalement au temps nécessaire à l'exécution des algorithmes. Ce temps d'exécution dépend principalement de :

- la machine utilisée;
- les données sur lesquelles l'algorithme est appliqué.

Complexité
Mesurer la complexité
Notions générales

Mais doit-on considérer des mesures de complexité du type suivant ?

> Pour le nombre 64165461351654, sur la machine AlphaX22, l'algorithme A a un temps d'exécution de $1.2 \times 10^{-3} s$ et l'algorithme B a un temps d'exécution de $9.8 \times 10^{-4} s$.

Cette information manque de généralité.

Complexité
Mesurer la complexité
Notions générales

Il convient donc d'abstraire cette mesure. Nous le ferrons dans trois directions :

- taille versus valeur, la taille est certainement plus importante que les valeurs particulières sur lesquelles l'algorithme travaille;

- néanmoins, pour des tailles égales d'instances, les valeurs peuvent avoir une certaine importance. Nous tiendrons compte du plus mauvais cas (worst-case analysis);

Complexité
Mesurer la complexité
Notions générales

- pour ne pas tenir compte du temps particulier nécessaire pour la résolution d'un problème sur une machine particulière (PC vs Super Computer Q), nous mesurerons la complexité en nombre de pas (opérations élémentaires). Notons que la vitesse pour effectuer un pas peut simplement être considérée comme une constante multiplicative.

Complexité
Mesurer la complexité
Notions générales

Par exemple, la méthode de tri “bubble sort” a une fonction de complexité de la forme cn^2. C’est-à-dire qu’elle permet de trier n entiers de taille bornée (par exemple encodés sur 32bits) en un temps proportionnel à n^2. Dans cette mesure la constante c n’est pas très significative. Dans la suite, nous utiliserons d’ailleurs la notation $\mathbf{O}$.

Complexité
La notation $\mathbf{O}$

Une fonction $g(n)$ est dite $\mathbf{O}(f(n))$ s'il existe des constantes c et n_0 telle que pour tout $n > n_0$

$$g(n) \leq cf(n)$$

Avantages à l'utilisation de $\mathbf{O}$:

- cette notation simplifie les notations;
- elle introduit implicitement le facteur constant nécessaire;

Complexité
La notation $\mathbf{O}$

- elle n'exprime qu'une borne supérieure;

- elle ignore les comportements pour les petites valeurs (évite les problèmes liés aux instructions d'initialisation par exemple).

Exemple : $\mathbf{O}(cn^2) = \mathbf{O}(n^2)$, de même

$$\mathbf{O}(c_1 n^2 + c_2 n) = \mathbf{O}(n^2)$$

car pour $n \geq 1$

$$c_1 n^2 + c_2 n \leq (c_1 + c_2) n^2.$$

Complexité
Mesurer la complexité
Notions générales

En résumé, de quoi dépend une classe de complexité ?

De trois paramètres :

- le modèle de calcul (ici on considère les machines de Turing avec 1 ou plusieurs rubans);

- le mode d'exécution : déterministe - non déterministe;

- la ressource que l'on veut borner : le nombre de pas d'exécution (temps calcul) ou la mémoire. On bornera cette ressource grâce à une fonction qui dépend de la taille de l'input (problème à résoudre). Cette fonction sera exprimé en $\mathbf{O}$.

Complexité
Algorithmes efficaces

Quand un algorithme est-il efficace ?

$$\mathbf{O}(n), \mathbf{O}(n^2), \mathbf{O}(n^3), \ldots \text{ ?}$$

Difficile à dire...

Par contre une complexité exponentielle $\mathbf{O}(c^n)$ est presque toujours excessive.

Par exemple si $c = 2$ et $n = 100$ (taille modeste), on a $2^{100} = 10^{30}$. Si chaque opération prend un nano-seconde, l'exécution prendra de l'ordre de 3×10^{11} siècles ! Et si $n = 1000$ ... Et donc quelque soit les facteurs constants (vitesse de l'ordinateur) qui affectent l'exponentielle, le temps de calcul est clairement excessif même pour des tailles modeste de problèmes.

Complexité
Algorithmes efficaces

Ceci étant dit, peut-on dire qu'une complexité polynomiale est acceptable ? En général, non ! Mais en pratique la différence entre complexité polynomiale et exponentielle est tellement forte qu'il est tentant de fixer là une limite. De plus en pratique, il est rare d'obtenir des complexités polynomiales avec un exposant élevé.

De plus, notre but principal est de montrer que des problèmes n'ont pas de solution algorithmique efficace.

Complexité
Problème et langage

Ici, à nouveau, nous nous intéresserons à des problèmes binaires.

Néanmoins, quelques précautions doivent être prises. Notre but est de tirer des conclusions sur la complexité de problèmes à partir de la complexité (de décision) du langage de l'encodage des instances positives du problème.

Heureusement, vu que nous considérons la frontière polynomiale-exponentielle comme notre mesure discriminante, la situation est assez simple.

En effet, il est facile de se convaincre que les encodages naturels des instances d'un problème sont équivalents à un polynôme près.

Complexité
Problème et langage

Considérons l'encodage d'un problème relatif à un graphe fini (V, E). On peut par exemple considérer les deux encodages traditionnels :

1. chaque sommet est représenté par un nombre binaire de longueur $\lceil \log_2(|V|) \rceil$ bits; chaque arc par une pair (v_1, v_2); la representation de ce graphe étant alors la liste de ces m sommets et n arcs, l'encodage est alors de complexité $\mathbf{O}((m+n) \times \log_2(n))$

2. un autre encodage consiste à représenter les sommets comme ci-dessus mais les arcs à l'aide d'une matrice d'incidence. La complexité de cette encodage est alors $\mathbf{O}(n^2)$.

Complexité
Problème et langage

Propriété d'un encodage raisonable (qui permet de tirer une conclusion sur un problème à partir de son encodage) :

- l'encodage ne peut contenir de *bourrage*. Par exemple, il y a bourrage si il existe un encodage de longueur n et qu'on ajoute à la suite de cet encodage $2^n - n$ caractères sans signification. Si il existe un algorithme polynomiale sur cet encodage, on ne peut bien entendu pas en conclure que le problème soit en effet polynomial;

- il doit être possible de décoder la représentation d'un problème en temps polynomial (on ne veut pas d'encodage où un chiffre serait codé via le problème de l'arrêt d'une

machine de Turing par exemple, dans ces circonstances on pourrait passer à côté d'une solution polynomiale pour le problème).

- les nombres ne peuvent pas être représentés en numération unaire.

Complexité et machines de Turing

Comme pour la décidabilité, nous adopterons les machines de Turing pour formaliser la notion d'algorithme polynomial.

Définition : soit M une machine de Turing déterministe qui s'arrête toujours. La *complexité en temps* de M est la fonction $T_M(n)$ définie par

$$T_M(n) = \max\{m | \exists x \in \Sigma^*, |x| = n \text{ et l'exécution de } M \text{ sur } x \text{ comporte } m \text{ étapes}\}$$

C'est bien une complexité au cas le plus mauvais.

Définition : Une machine de Turing M est *polynomiale* si il existe un polynôme $p(n)$ tel que la fonction de complexité $T_M(n)$ satisfait :

$$T_M(n) \leq p(n)$$

pour tout $n \geq 0$.

Complexité
et machines de Turing

Thèse : si il existe un algorithme polynomiale pour résoudre un problème alors il existe une machine de Turing polynomiale pour résoudre ce problème.

Théorèmes qui appuient cette thèse :

Thm : Etant donnée un machine de Turing *multi*-rubans M qui opère en temps $\mathbf{O}(f(n))$ alors il existe un machine de turing M' qui décide exactement le même langage en temps $\mathbf{O}(f(n)^2)$.

Thm : Si une machine RAM Π calcule une fonction ϕ en temps $\mathbf{O}(f(n))$, alors il existe une machine de Turing M à 7 rubans qui calcule ϕ en temps $\mathbf{O}(f(n)^3)$.

Nous ne démontrerons pas ces thms. Le lecteur intéressé par plus de détails est renvoyé à [CP93].

Complexité
La classe P

On définit maintenant la classe des problèmes polynomiaux.

Définition : La classe P est la classe des langages décidés par une machine de Turing polynomiale.

Nous affirmons que :

- la classe P caractérise bien la notion d'algorithme efficace. Les algorithmes polynomiaux sont souvent efficaces et les algorithmes exponentiels le sont rarement;

- la classe P a une définition robuste : elle est insensible au choix particulier de machine et d'encodage.

Complexité
Transformations polynomiales

Deux exemples de problèmes qui n'ont pas de solution polynomiale connue :

- *problème du voyageur de commerce* : soit V un ensemble de n villes; $d : V \times V \rightarrow \mathbb{N}$ une fonction qui renvoit la distance entre deux pairs de villes, et b une borne. Le problème consiste à déterminer si il existe un circuit passant une et une seule fois par chaque ville et ayant une longueur inférieure à b.

  Algo évident : essayer toutes les permutations possibles des n villes (on teste chaque circuit possible). Mais sa complexité est $\mathbf{O}(n!)$.

Complexité
Transformations polynomiales

- *problème du circuit hamiltonien* : soit un graphe $G = (V, E)$, le problème consiste à décider si G contient un circuit fermé qui passe une et une seule fois par chaque ville.

  Algo évident : l'algorithme qui vérifie l'ensemble des permutations de villes possibles est également une solution, non polynomiale malheureusement.

Complexité
Transformations polynomiales

A propos des deux problèmes précédents, on peut démontrer le fait suivant :

Thm : le problème du voyageur (TS) de commerce est dans $\mathbf{P}$ si et seulement si le problème du circuit hamiltonien (HC) est dans $\mathbf{P}$.

Une telle propriété est intéressante car elle remplace deux questions HC $\in^? \mathbf{P}$ et TS $\in^? \mathbf{P}$ par une seule question $\{\text{HC}, \text{TS} \subset^? \mathbf{P}\}$

Pour établir le thm ci-dessus, on utilise la notion de transformation polynomiale.

Complexité
Transformations polynomiales

Définition : soient un langage $L_1 \in \Sigma_1^*$ et un langage $L_2 \in \Sigma_2^*$. Une *transformation polynomiale* de L_1 vers L_2 (notation $L_1 \propto L_2$) est une fonction $f : \Sigma_1^* \to \Sigma_2^*$ qui satisfait les conditions suivantes :

- elle est calculable en temps polynomial,
- $f(x) \in L_2$ si et seulement si $x \in L_1$.

On utilisera également le terme *réductions polynomiales* pour désigner une transformation polynomiale.

Complexité
Transformations polynomiales

On peut rephraser la définition précédente en terme de problème :

> Une transformation polynomiale est une fonction f calculable en temps polynomiale qui à partir d'une instance x d'un problème P_1, calcule une instante $f(x)$ d'une problème P_2 telle que x est positive ssi $f(x)$ l'est.

Complexité
Transformations polynomiales

Thm : il existe une transformation polynomiale du problème du circuit hamiltonien vers le problème du voyageur de commerce.

Preuve : Soit une instance de HC définie par un graphe $G = (V, E)$. On obtient l'instance de TS suivante :

- l'ensemble des villes est égal à l'ensemble des sommets du graphe : $C = V$;

- la fonction d est définie comme suit :

$$d(v_i, v_j) = \begin{cases} 1 & \text{si } (v_i, v_j) \in E. \\ 2 & \text{si } (v_i, v_j) \notin E. \end{cases}$$

- $b = |V|$

Il faut maintenant établir deux choses :

- la transformation est polynomiale. En effet, la liste des villes créée est la liste des sommets, la définition de d est obtenue en inspectant les arcs du graphe. Ces deux opérations sont clairement polynomiales.

- la transformation préserve le caractère positif et négatif des instances.

  En effet, pour le caractère positif : supposons que l'instance de TS construite soit positive. S'il existe un parcours des villes de longueur $b = |V|$, ce parcours ne peut contenir que des arcs de longueur 1. Et donc HC est positif.

  Pour le caractère négatif : (contraposée) si HC est positif, le circuit définit un parcours des villes n'utilisant que des arcs de

longueur 1 et donc bien un circuit de longueur totale $|V|$.

On a donc bien HC $\propto$ TS. Bien que ce soit moins évident on peut également montrer TS $\propto$ HC.

Complexité
Transformations polynomiales
Propriétés

Lemme : si $L_1 \propto L_2$ alors

- si $L_2 \in \mathbf{P}$ alors $L_1 \in \mathbf{P}$;
- si $L_1 \notin \mathbf{P}$ alors $L_2 \notin \mathbf{P}$.

Preuve : la deuxième affirmation est la contraposée de la première, nous ne justifions donc que la première.

Pour résoudre $w \in L_1$, on transforme w en $f(w)$ avec la transformation polynomiale f. Nous notons $p(n)$ le polynôme qui définit la complexité de la transformation f. Ensuite on teste $f(w) \in L_2$. Vu que L_2 est polynomial on note $p'(m)$ le polynôme définissant la complexité de ce test. La complexité totale pour tester $w \in L_1$ est donc, $p(n) + p'(p(n))$ qui est bien un polynôme.

Complexité
Transformations polynomiales
Propriétés

Notons que le lemme précédent nous permet détablir que HC $\in \mathbf{P}$ ssi TS $\in \mathbf{P}$.

Lemme : Les transformations polynomiales sont transitives : si $L_1 \propto L_2$ et $L_2 \propto L_3$ alors $L_1 \propto L_3$.

Preuve : Pour obtenir une transformation de L_1 à L_3, il suffit de composer les transformations de L_1 à L_2 et de L_2 à L_3.

Complexité
Transformations polynomiales

Définition : Deux langages L_1 et L_2 sont *polynomialement équivalents* si et seulement si $L_1 \propto L_2$ et $L_2 \propto L_1$.

La relation "polynomialement équivanlent" est une relation d'équivalence (transitive, réflexive et symétrique). Cette relation définit des classes d'équivalence.

Dans une même classe d'équivalence soit aucun problème n'a de solution polynomiales soit tous ont une solution polynomiale.

Si on veut montrer qu'un langage L appartient à une classe d'équivalence polynomiale C, il suffit de trouver $L_1 \in C$ et $L_2 \in C$ tels que $L \propto L_1$ et $L_2 \propto L$.

Complexité
Transformations polynomiales

Ceci est très utile : on peut replacer les questions individuelles sur les langages d'une classe concernant leur polynomialité par "la classe admet-elle oui ou non une solution polynomiale".

En pratique, on remarque la classe d'équivalence qui contient HC et TS, contient beaucoup de problèmes dont aucun n'a de solution polynomiale connue.

Complexité
La classe NP

La résolution du problème du circuit hamiltonien est difficile,

- non pas parce qu'il est difficile de décider si une proposition de solution est effectivement une solution. Déterminer si une sequence de sommets est ou non un circuit hamiltonien est facile;

- par contre il est difficile car il y a un nombre exponentiel de solutions potentielles.

Cette caractéristique est vraie pour un grand nombre de problèmes.

Complexité
La classe NP

Si l'énumération ne coutait rien alors ces problèmes auraient une solution algorithmique efficace.

Si on adopte le modèle (non réaliste) des machines de Turing **non-déterministe** pour énumérer les différentes solutions potentielles alors ces problèmes sont "faciles" à résoudre.

Complexité
La classe NP

Pour résoudre le problème du circuit hamiltonien avec une machine de Turing non déterministe, on procède comme suit :

- chaque exécution possible de la machine génère une permuttation des sommets du graphe (utilisation du non-déterminisme);

- la machine n'accepte que les exécutions qui correspondent à des circuits hamiltoniens.

Complexité
La classe NP
Complexité des machines non-déterministes

Définition : Le temps de calcul d'une machine de Turing sur un mot w est donné par

- la longueur de la **plus courte** exécution acceptant le mot si celui-ci est accepté;
- la valeur 1 si le mot n'est pas accepté.

Donc, on ne tient pas compte du temps calcul des mots non accepté

Définition : Soit M une machine de Turing non déterministe. La complexité en temps de M est la fonction $T_M(n)$ définie par

$$T_M(n) = \max\{m | \exists x \in \Sigma^*, |x| = n \text{ et le temps de calcul de } M \text{ sur } x \text{ est } m\}$$

Complexité
Définition de la classe NP

Définition : la classe **NP** (**N**on déterministe **P**olynomial) est la classe des langages acceptés par une machine de Turing non déterministe polynomiale.

Théorème : Soit L un langage appartenant à **NP**. Alors il existe une machine de Turing **déterministe** et un polynôme $p(n)$ tel que M décide L et est de complexité en temps bornée par $2^{p(n)}$.

Preuve : Soit M_{nd} la machine non-déterministe de complexité $q(n)$ qui accepte L. On sait donc que si M_{nd} accepte un mot w de longueur m, alors elle l'accepte avec une exécution d'une longueur bornée par $q(m)$.

Nous construisons une machine de Turing déterministe M qui va :

- simuler toutes les exécutions de M_{nd} sur w, exécutions de longueur $l \leq q(m)$;

- accepter w si une de ces exécutions termine dans un état accepteur.

Quelle est la complexité de M ? Soit r le nombre maximum de choix possibles pour une configuration successeur dans M_{nd} (degré de non-déterminisme). Il y a donc $r^{q(n)}$ exécutions possibles à simuler. Chaque exécution à une longueur bornée par $q(n)$ et donc doit pouvoir être simulée en un temps borné par un polynôme en $q(n)$, donc par un polynôme $q'(n)$. Le temps calcul global est donc $r^{q(n)} \times q'(n)$, qui est borné par $2^{\log_2(r)(q(n)+q'(n))}$ qui est bien de la forme $2^{p(n)}$ pour un polynôme p.

Ce théorème établit que l'on peut résoudre les problèmes de la classe **NP** avec un algorithme exponentiel. Il y a beaucoup de raisons de penser que l'on ne peut pas résoudre ces problèmes avec un algorithme polynomial. Aucune preuve n'a toutefois pu être construire à l'heure actuelle. C'est le fameux problème :

$$\mathbf{P} =^{?} \mathbf{NP}$$

Complexité
Structure de la classe NP

La notion de polynomialement équivalent permet de structure **NP**.

Nous définition l'ordre partiel suivant :

Définition : Une classe d'équivalence polynomiale C_1 est inférieure à une classe déquivalence C_2 (notation $C_1 \preceq C_2$) s'il existe une transformation polynomiale de tout langage C_1 vers tout langage C_2.

Intuitivement $C_1 \preceq C_2$ si "les problèmes de C_1 sont plus faciles à résoudre que les problèmes de C_2".

Complexité
Structure de la classe NP

Théorème : La classe **NP** contient la classe **P** ($\mathbf{P} \subseteq \mathbf{NP}$).

La preuve est immédiate.

Lemme : La classe **P** est une classe d'équivalence polynomiale.

Preuve : Montrons que pour tout $L_1, L_2 \in \mathbf{P}$ on a $L_1 \propto L_2$. Soit w un mot :

- on détermine si $w \in L_1$: temps polynomial requit;

- si $w \in L_1$, générer w' tel que $w' \in L_2$ (ce w' peut-être le même pour tous les w). si $w \notin L_1$, générer (un autre) w' tel que $w' \notin L_2$.

Complexité
Structure de la classe NP

De manière similaire, on peut montrer :

Lemme : Pour tout $L_1 \in \mathbf{P}$ et pout tout $L_2 \in \mathbf{NP}$, on a $L_1 \propto L_2$.

Complexité
Les problèmes NP-complet

NP-Complet = classe des problèmes les "plus difficiles" dans **NP**.

Définition : Un langage L est **NP**-complet si

1. $L \in \mathbf{NP}$,

2. pour tout langage $L' \in \mathbf{NP}$, $L' \propto L$.

Théroème : Si il existe un langage **NP**-complet L est décidé par un algorithme polynomial, alors tous les langages **NP** sont décidables en temps polynomial, c'est-à-dire $\mathbf{P} = \mathbf{NP}$.

Complexité
Etablir la NP-complétude

Pour démontrer $L \in \mathbf{NPC}$, on montre que : (i) $L \in \mathbf{NP}$ (ii) pour tout langage $L' \in \mathbf{NP}$, $L' \propto L$.

A la place du point deux, on peut montrer qu'il existe $L' \in \mathbf{NPC}$ tel que $L' \propto L$ (ceci grâce à la propriété de transitivité des transformations polynomiales).

Remarque : un problème est **NP**-dur si il est au moins **NP**. Il existe des problèmes qui sont plus difficiles que les problèmes **NP**-complet.

Complexité
Un premier problème NP-complet

Tables de vérité du calcul booléen :

p	$\neg p$
0	1
1	0

p	q	$p \vee q$
0	0	0
0	1	1
1	0	1
1	1	1

p	q	$p \wedge q$
0	0	0
0	1	0
1	0	0
1	1	1

p	q	$p \longrightarrow q$
0	0	1
0	1	1
1	0	0
1	1	1

Complexité
Un premier problème NP-complet

- Expression booléenne :

$$(1 \wedge (0 \vee (\neg 1))) \rightarrow 0$$

  La valeur de vérité d'une expression booléenne est évaluable en temps polynomial (tables de vérité).

- Calcul propositionnel : introduction de variables

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

- Fonction d'interprétation : $f : P \rightarrow \{0, 1\}$, f attribue une valeur de vérité aux variables booléennes.

- Formule satisfaisable : une formule propositionnelle ϕ est *satisfaisable* si et seulement si il existe une fonction d'interprétation f qui rend vraie ϕ (on note $f \models \phi$ le fait que f rend vrai ϕ).

- Formule valide : une formule propositionnelle ϕ est *valide* si et seulement si toute fonction d'interprétation f rend vraie la formule ϕ.

- Problème : le problème de satisfaisabilité (validité) propositionnel est de déterminer si une formule ϕ est satisfaisable (validité).

- Une formule ϕ est en *forme normale conjonctive* si c'est une conjonction de disjonctions de littéraux.

ex : $(\neg p \vee q) \wedge (r \vee \neg s \vee p)$

- Une formule ϕ est en *forme normale disjonctive* si c'est une disjonction de conjonctions de littéraux.

ex : $(\neg p \wedge q) \vee (r \wedge \neg s \wedge p)$

Complexité
Un premier problème NP-complet
Théorème de Cook

Problème **SAT** : problème de décider la satisfaisabilité d'une formule propositionnelle en forme normale conjonctive.

Théorème : le problème **SAT** est **NP**-complet.

Preuve :

(i) **SAT** est bien dans **NP**. En effet, étant donné une fonction d'interprétation f, il existe bien un algorithme polynomial pour décider si $f \models^{?} \phi$.

(ii) **SAT** est **NP**-dur. On va montrer qu'il existe une transformation polynomiale de tout langage de **NP** vers $L_{\mathbf{SAT}}$.

- transformation à 2 arguments : un mot w et un langage L.

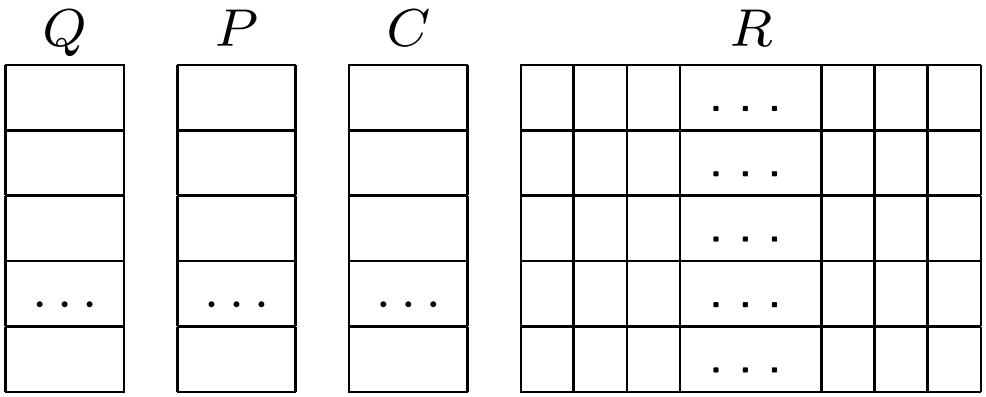
- on s'appuie sur le fait que chaque langage L est caractérisé par une machine de Turing non déterministe polynomiale (bornée par un polynôme $p(n)$).

Soit un mot w ($|w| = n$) et une machine de Turing non déterministe $M = (Q, \Gamma, \Sigma, \Delta, s, B, F)$ (borné par $p(n)$).

Le schéma de preuve pour montrer le fait que **SAT** est **NP**-dur est le suivant : on va montrer que pour chaque machine M et mot w, on peut construire une formule $\phi_{M,w}$ telle que cette formule est satisfaisable si et seulement si $w \in L(M)$. De plus, on va montrer que la formule construite est en forme normale conjonctive et sa longueur est bornée polynomialement par rapport à $p(n)$.

Idée : une interprétation f satisfait $\phi_{w,M}$ si f "décrit" une exécution acceptée de M sur w.

Le tableau suivant, quand il est complété, contient la description d'une exécution acceptée de M sur w. Ce tableau contient un nombre de lignes et de colonnes borné par $p(n)$.



Représentation d'une exécution par des variables propositionnelles :

- une proposition $r_{ij\alpha}$ pour $0 \leq i, j \leq p(n)$ et $\alpha \in \Gamma$.

$r_{ij\alpha}$ est vraie ssi lors du pas i de l'exécution, la case j du ruban contient le symbole α.

- une proposition $q_{i\kappa}$ pour $0 \leq i, j \leq p(n)$ et $\kappa \in Q$.

  $q_{i\kappa}$ est vraie ssi lors du pas i de l'exécution la machine est dans l'état κ.

- une proposition p_{ij} pour $0 \leq i, j \leq p(n)$.

  p_{ij} est vraie ssi lors du pas i de l'exécution, la tête de lecture se trouve en position j.

- une proposition c_{ik} pour $0 \leq i, j \leq p(n)$ et $1 \leq k \leq r$.

c_{ik} est vraie ssi le choix effectué lors du pas i de l'exécution est le choix numéro k.

On construit alors la formule $\phi_{w,M}$ comme la conjonction des formules propositionnnelles suivantes :

- chaque case du ruban contient exactement un symbole :

$$\bigwedge_{0 \leq i,j \leq p(n)} \left[\left(\bigvee_{\alpha \in \Gamma} r_{ij\alpha} \right) \wedge \bigwedge_{\alpha \neq \alpha' \in \Gamma} (\neg r_{ij\alpha} \vee \neg r_{ij\alpha'}) \right]$$

  Notons que cette formule a une longueur de $\mathbf{O}(p(n)^2)$.

- à chaque moment de l'exécution, la tête de lecture se trouve exactement sur une position du ruban :

$$\bigwedge_{0\leq i\leq p(n)}\left[\bigvee_{0\leq j\leq p(n)}(p_{ij}\wedge(\bigwedge_{0\leq j\neq j'\leq p(n)}(\neg p_{ij}\vee\neg p_{ij'})))\right]$$

Notons que cette formule a une longueur bornée par $\mathbf{O}(p(n)^3)$.

- description de l'état initial de la machine :

$$\left[\bigwedge_{0\leq j\leq n-1} r_{0jw_{j+1}}\wedge\bigwedge_{n\leq j\leq p(n)} r_{0jB}\right]\wedge q_{0s}\wedge p_{00}$$

Notons que cette formule a une longueur bornée par $\mathbf{O}(p(n))$.

- Les transitions de la machine sont exprimées à l'aide de formules de la forme suivante :

$$\bigwedge_{\substack{0 \leq i < p(n) \\ 0 \leq j \leq p(n) \\ \alpha \in \Gamma}} \left[(r_{ij\alpha} \wedge \neg p_{ij}) \rightarrow r_{(i+1)j\alpha} \right]$$

Cette formule peut facilement être mise en forme normale conjonctive (transformer l'implication en une disjonction). Sa longueur est de $\mathbf{O}(p(n)^2)$.

- Autre formule pour formaliser la relation de transition :

$$\bigwedge_{\substack{0 \leq i < p(n) \\ 0 \leq j \leq p(n) \\ \alpha \in \Gamma \\ 1 \leq k \leq r}} \left[\begin{array}{l} (q_{i\kappa} \wedge p_{ij} \wedge r_{ij\alpha} \wedge c_{ik}) \rightarrow q_{(i+1)\kappa'}) \wedge \\ (q_{i\kappa} \wedge p_{ij} \wedge r_{ij\alpha} \wedge c_{ik}) \rightarrow r_{(i+1)j\alpha'} \wedge \\ (q_{i\kappa} \wedge p_{ij} \wedge r_{ij\alpha} \wedge c_{ik}) \rightarrow p_{(i+1)(j+d)} \end{array} \right]$$

Où $d \in \{-1, 1\}$ (-1 si déplacement de la tête de lecture vers la gauche, $+1$ vers la

droite). Cette formule peut facilement être mise en forme normale conjonctive (transformer l'implication en une disjonction). Sa longueur est de $\mathbf{O}(p(n)^2)$.

- La formule suivante exprime que l'état final est atteint :

$$\bigvee_{\substack{0 \leq i \leq p(n) \\ \kappa \in F}} [q_{i\kappa}]$$

Cette formule est de longeur $\mathbf{O}(p(n))$.

Donc la longueur de la formule entière est de $\mathbf{O}(p(n)^3)$. Elle peut être construite en temps polynomiale par rapport à $p(n)$. On a bien que $\phi_{w,M}$ est satisfaisable ssi $w \in L(M)$. On a donc bien une transformation polynomiale de tout langage **NP** vers **SAT**.

Complexité
D'autres problèmes NP-complets
3SAT

Nous allons maintenant montrer qu'un cas particulier de **SAT**, appelé **3SAT**, est également **NP**-complet.

3SAT : satisfaisabilité des formules propositionnelles en forme normale conjonctives qui ont exactement 3 littéraux par clause.

Théorème : $\mathbf{SAT} \propto \mathbf{3SAT}$.

Une fonction d'interprétation $f' : P' \to \{0,1\}$ est une extension de $f : P \to \{0,1\}$ ssi $P \subset P'$ et $\forall p \in P : f'(p) = f(p)$.

Complexité
D'autres problèmes NP-complets
3SAT

Il faut montrer que pour toute formule ϕ en forme normale conjonctive, on peut construire une formule ϕ' en forme normale conjonctive avec exactement 3 littéraux par clause et telle que ϕ' est satisfaisable ssi ϕ est satisfaisable.

De plus la formule ϕ' doit être constructible en temps polynomial par rapport à la taille de ϕ.

Complexité
D'autres problèmes NP-complets
3SAT

Pour construire ϕ', on procède de la façon suivante :

- une clause $\psi \equiv (x_1 \vee x_2)$ contenant deux littéraux est remplacée par

  $$\psi' \equiv (x_1 \vee x_2 \vee y) \wedge (x_1 \vee x_2 \vee \neg y)$$

  où $y \in P' \setminus P$ (y est une nouvelle variable). Notons qu'on a bien que que pour tout f et f' une extension de f:

  $$f \models \psi \text{ ssi } f' \models \psi'$$

  Et donc, notre transformation préserve bien la satisfaisabilité.

Complexité
D'autres problèmes NP-complets
3SAT

- une clause $\psi \equiv (x_1)$ comportant un seul littéral est remplacée par :

$$\psi' \equiv \begin{array}{l} (x_1 \vee y_1 \vee y_2) \wedge \\ (x_1 \vee y_1 \vee \neg y_2) \wedge \\ (x_1 \vee \neg y_1 \vee y_2) \wedge \\ (x_1 \vee \neg y_1 \vee \neg y_2) \end{array}$$

Complexité
D'autres problèmes NP-complets
3SAT

- une clause $\psi \equiv (x_1 \vee x_2 \vee \ldots \vee x_i \vee \ldots \vee x_l)$ comportant $l \geq 4$ littéraux est remplacée par :

$$
\begin{aligned}
(x_1 \vee x_2 \vee y_1) \quad & \wedge(\neg y_1 \vee x_3 \vee y_2) \\
& \wedge(\neg y_2 \vee x_4 \vee y_3) \wedge \ldots \\
& \wedge(\neg y_{i-2} \vee x_i \vee y_{i-1}) \wedge \ldots \\
& \wedge(\neg y_{l-4} \vee x_{l-2} \vee y_{l-3}) \\
& \wedge(\neg y_{l-3} \vee x_{l-1} \vee x_l)
\end{aligned}
$$

  A nouveau on peut se convaincre que la formule obtenue est de taille polynomiale par rapport à la clause de départ et cette clause est satisfaisable par et seulement par les extensions des fonctions d'interprétation qui satisfont la clause de départ.

Complexité
D'autres problèmes NP-complets
La couverture de sommets (VC)

VC : Etant donnés un graphe $G = (V, E)$ et un entier $j \leq |E|$, il faut déterminer s'il existe un sous-ensemble $V' \subseteq V$ tel que $|V'| \leq j$ et tel que pour tout arc $(u, v) \in E$, soit u, soit $v \in V'$.

Théorème : $\mathbf{3SAT} \propto \mathbf{VC}$.

Réduction : Pour chaque instance positive (négative) de $\mathbf{3SAT}$, on construit une instance positive (négative) de $\mathbf{VC}$.

Complexité
D'autres problèmes NP-complets
La couverture de sommets (VC)

Instance de $\mathbf{3SAT}$:

$$\Psi \equiv E_1 \wedge \ldots \wedge E_i \wedge \ldots \wedge E_k$$

où chaque E_i est de la forme :

$$x_{i1} \vee x_{i2} \vee x_{i3}$$

où x_{ij} est un littéral. Ces formules du calcul propositionnel sont construites sur l'ensemble de propositions :

$$\mathcal{P} = \{p_1, \ldots, p_l\}$$

Complexité
D'autres problèmes NP-complets
La couverture de sommets (VC)

Etant donnée la formule Ψ, on construit l'instance suivante de **VC**.

Le graphe $G = (V, E)$ contient l'ensemble de sommets V suivant :

(i) pour chaque proposition $p_i \in \mathcal{P}$, V contient deux sommets: p_i et $\neg p_i$;

(ii) pour chaque clause $x_{i1} \vee x_{i2} \vee x_{i3}$, V contient trois sommets x_{i1}, x_{i2} et x_{i3}.

$| V |$ est donc égale à $2 \times l + 3 \times k$.

Complexité
D'autres problèmes NP-complets
La couverture de sommets (VC)

L'ensemble E contient les paires suivantes :

(i) l'arc $(p_i, \neg p_i)$ pour chaque paire de sommets $p_i, \neg p_i$, $1 \leq i \leq l$;

(ii) les arcs (x_{i1}, x_{i2}), (x_{i2}, x_{i3}) et (x_{i3}, x_{i1}) pour chaque clause $x_{i1} \vee x_{i2} \vee x_{i3}$;

(iii) un arc entre chaque sommet x_{ij} et le sommet p ou $\neg p$ représentant le littéral correspondant.

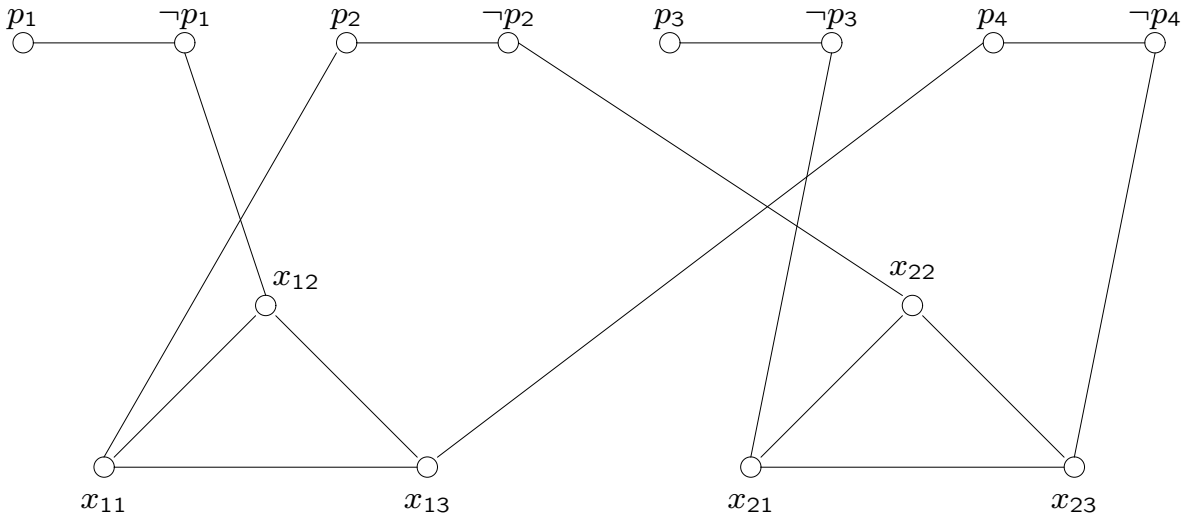
Le nombre d'arcs du graphe G est donc de $l + 6 \times k$.

Finalement, on fixe la constante j à $l + 2 \times k$.

Complexité
D'autres problèmes NP-complets
La couverture de sommets (VC)

Voici une illustration de cette réduction :

$$(p_2 \vee \neg p_1 \vee p_4) \wedge (\neg p_3 \vee \neg p_2 \vee \neg p_4)$$



Complexité
D'autres problèmes NP-complets
La couverture de sommets (VC)

Preuve de correction : on a donc construit à partir de la formule Ψ, une instance de $\mathbf{VC}$ dont le graphe est $G = (V, E)$ et $j = l + 2 \times k$. Il faut maintenant montrer que

Ψ est satisfaisable

iff

G a une couverture C de taille $\leq j$

Complexité
D'autres problèmes NP-complets
La couverture de sommets (VC)

($\Rightarrow$)
On sait que Ψ est satisfaisable et donc il existe une valuation f telle que $f \models \Psi$. On va construire $C \subset V$ à partir de f de la façon suivante:

- pour tout littéral u de la forme p_i ou $\neg p_i$, on a $u \in C$ iff $f \models u$.

- pour chaque triangle, on inclut deux sommets dans C de telle sorte que le sommet non choisi du triangle est connecté à un littéral u avec la propriété que $f \models u$. Notons qu'un tel sommet existe toujours vu que chaque clause évalue à vrai pour l'interprétation f, et donc il existe au moins un littéral x_{ij} par clause i qui évalue à vrai pour f.

Nous avons donc construit C et $|C| = l + 2 \times k$.

Complexité
D'autres problèmes NP-complets
La couverture de sommets (VC)

Montrons que C est bien une couverture de G:

- les arcs $(p_i, \neg p_i)$ sont couverts car soit $p_i \in C$ ou $\neg p_i \in C$ (toute valuation f rend soit p_i vrai, soit $\neg p_i$ vrai);

- les arcs formant les triangles sont couverts. En effet, en sélectionant deux sommets parmis les trois d'un triangle, on couvre toujours les arcs qui forment le triangle;

- les arcs qui relient un sommet du triangle sélectionné dans le point précédent au littéral correspondant sont couverts; l'arc qui relie le sommet non sélectionné du triangle à l'ensemble des littéraux est couvert par le littéral car ce littéral évalue à vrai et fait donc partie de l'ensemble de couverture.

Complexité
D'autres problèmes NP-complets
La couverture de sommets (VC)

($\Leftarrow$) Soit C une couverture de G de taille j. Montrons qu'à partir de C, on peut construire une fonction d'interprétation f telle que $f \models \Psi$.

On définit f de la façon suivante :

$$\text{pour tout } p \in \mathcal{P}, f(p) = 1 \textbf{ iff } p \in C$$

Montrons maintenant que $f \models \Psi$.

Soit E_i, une clause et x_{i1}, x_{i2}, x_{i3}, les trois sommets qui correspondent à cette clause. On sait que $|C| = l + 2k$. Montrons que parmi les trois sommets de la clause i, il y a en a exactement deux qui appartiennent à C.

En effet, on sait qu'il y en a au moins deux qui appartiennent à C sinon un des arcs (x_{i1}, x_{i2}), (x_{i2}, x_{i3}), (x_{i3}, x_{i1}) est non couvert. Montrons maintenant qu'au plus deux sommets du triangle peuvent être couverts. Faisons l'hypothèse que trois sommets d'un triangle sont couverts. Dans ce cas, C ne peut contenir que

$$2 \times k + l - (3 + 2 \times (k - 1)) = l - 1 \text{ sommets}$$

parmi l'ensemble des sommets $\{p_i, \neg p_i \mid p_i \in \mathcal{P}\}$. Comme la cardinalité de $\mathcal{P}$ est l, il existe un arc $(p_i, \neg p_i)$ qui n'est pas couvert, ce qui est impossible.

Donc, exactement deux sommets de chaque triangle sont couverts. Le troisième sommet, soit x_{ij} est l'origine d'un arc de la forme (x_{ij}, p) (si $x_{ij} = p$), ou de la forme $(x_{ij}, \neg p)$ (si $x_{ij} = \neg p$). Supposons arbitrairement que ce soit l'arc (x_{ij}, p) qui existe. Vu que $x_{ij} \notin C$, on a que $p \in C$. Par définition on a donc $f \models p$ et donc la clause E_i est satisfaite par f. On peut faire le même raisonnement pour chaque clause de Ψ. On a donc établit que $f \models \Psi$.

Complexité
Le matching à trois dimensions (3DM)

Définition de 3DM.

Le "machting à trois dimensions" est une généralisation du problème du mariage : étant donné n hommes non mariés et n femmes non mariées, une liste de toutes les paires homme-femme acceptables, décider si il est possible d'arranger n mariages de telle sorte que la polygamie est évitée et que chacun recoive un époux ou une épouse qu'il/elle trouve acceptable. De manière analogue, dans le matching à trois dimensions, trois ensembles W, X et Y correspondent à trois sexes et chaque triplet de M correspond à un mariage à trois acceptable.

Note : **2DM** peut être résolu en temps polynomial alors que **3DM** est **NP-Complet**.

Complexité
Le matching à trois dimensions (3DM)

Theorème : **3DM** est **NP-Complet**.

Preuve :

(i) **3DM** est *NP-Facile* : Cette partie du résultat est aisée à établir. En effet, un algorithme non-déterministe doit seulement deviner $q = |W| = |X| = |Y|$ triplets de M et vérifier, *en temps polynomial*, qu'aucun des triplets devinés n'ont de coordonnées communes.

(ii) **3DM** est *NP-Difficile* : on va établir cette partie du résultat en montrant $\mathbf{3SAT} \propto \mathbf{3DM}$.

Complexité
Le matching à trois dimensions (3DM)

Considérons :

- $U = \{u_1, u_2, \ldots, u_n\}$, n variables propositionnelles;

- $C = \{c_1, c_2, \ldots, c_m\}$, m clauses sur U constituant une instance arbitraire de **3SAT**.

On va construire les ensembles disjoints W, X et Y tels que $|W| = |X| = |Y|$ et l'ensemble $M \subseteq W \times X \times Y$ de telle sorte que M contient un matching **si et seulement si** C est satisfaisable.

Complexité
Le matching à trois dimensions (3DM)

M sera partionné en trois sous-ensembles :

1. les lignes qui assignent une valeur de vérité aux variables propositionnelles;

2. les lignes qui testent la satisfaction des clauses;

3. les lignes “garbage collection”.

Complexité
Le matching à trois dimensions (3DM)

Lignes "valeur de vérité"

Pour chaque variable propositionnelle $u \in U$, on construit un composant dont la structure dépend du nombre m de clauses dans C. Considérons la variable propositionnelle u_i ($1 \leq i \leq n$), le composant associé à cette variable implique les éléments :

- *internes* $a_i[j] \in X$ et $b_i[j] \in Y$ qui n'apparaîtront dans aucun autre triplet externe à ce composant;

- *externes* $u_i[j], \bar{u}_i[j] \in W$, $1 \leq j \leq m$ qui apparaîtront dans d'autres tuples.

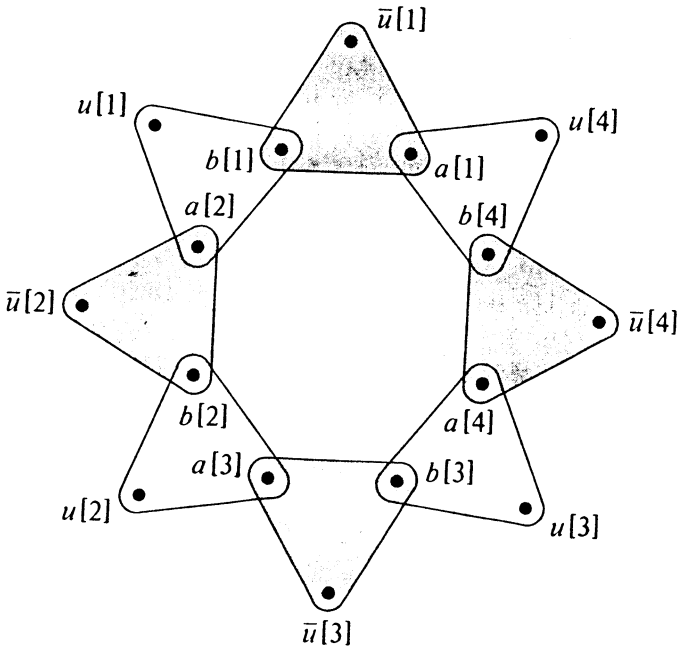
Note : W contient $2 \times m \times n$ éléments. Un matching devra contenir exactement ce nombre d'éléments.

Complexité
Le matching à trois dimensions (3DM)

Les triplets de ce composant peuvent être partionnés en deux sous-ensembles:

1. $T_i^t = \{(\bar{u}_i[j], a_i[j], b_i[j]) \mid 1 \leq j \leq m\}$

2. $T_i^f = \{(u_i[j], a_i[j+1], b_i[j]) \mid 1 \leq j \leq m\} \cup \{(u_i[m], a_i[1], b_i[m])\}$



Vu qu'aucun des éléments internes

$$\{a_i[j], b_i[j] \mid 1 \leq j \leq m\}$$

ne vont apparaître en dehors des triplets de $T_i = T_i^t \cup T_i^f$, il est facile de voir que tout matching M' doit comporter exactement m triplets de T_i, soit **tous** les triplets de T_i^t ou **tous** les triplets de T_i^f. Donc, on peut voir les lignes de T_i comme nous forçant à choisir la valeur de vérité vrai ou fausse pour u_i. Donc $M' \subseteq M$ définit un assignement de vérité pour chaque variable $u_i \in U$: u_i est vraie **ssi** $M' \cap T_i = T_i^t$.

Rem : pour simuler l'attribution d'une valeur de vérité à chaque variable propositionnelle, on choisit donc $m \times n$ lignes de M.

Complexité
Le matching à trois dimensions (3DM)

Lignes "test de satisfaction"

Pour chaque clause $c_j \in C$, on construit un composant de test de satisfaction. Celui-ci contient :

- deux éléments *internes* :

$$s_1[j] \in X \text{ et } s_2[j] \in Y, \text{ et}$$

- les éléments *externes* de l'ensemble

$$\{u_i[j], \bar{u}_i[j] \mid 1 \leq i \leq n\}$$

  suivant quels littéraux apparaissent dans c_j.

Complexité
Le matching à trois dimensions (3DM)

Les triples de ce composant sont :

$$C_j = \{(u_i[j], s_1[j], s_2[j]) \mid u_i \in c_j\}$$
$$\cup\{(\bar{u}_i[j], s_1[j], s_2[j]) \mid \bar{u}_i \in c_j\}$$

Donc tout matching M' contiendra exactement un triplet de C_j. Mais cela n'est possible que si un élément $u_i[j]$ (ou $\bar{u}_i[j]$) d'un littéral $u_i \in c_j$ ($\bar{u}_i \in c_j$) n'apparaît pas dans $T_i \cap M'$, ce qui sera le cas si et seulement si la fonction d'interprétation déterminé par M' satisfait la clause c_j.

Rem : on choisit donc ici m lignes (une pour chaque clause).

Complexité
Le matching à trois dimensions (3DM)

Lignes "garbage collection"

Pour rappel, $W = \{U_i[j], \bar{u}_i[j] \mid 1 \leq i \leq n, 1 \leq j \leq m\}$ et contient donc $2 \times m \times n$ éléments. Jusqu'à présent, $n \times m + m$ éléments ont été choisis et cela a été possible **ssi** l'instance de 3SAT est satisfaisable. Il faut maintenant permettre de choisir des lignes contenant les $m \times (n-1)$ éléments non encore choisis. On construit finalement un composant G, qui implique:

- les éléments *internes* $g_1[k] \in X$, $g_2[k] \in Y$ avec $1 \leq k \leq m(n-1)$;
- et les éléments *externes* $u_i[j]$ et $\bar{u}_i[j]$ de W.

G contient l'ensemble des triplets

$$G = \{(u_i[j], g_1[k], g_2[k]), (\bar{u}_i[j], g_1[k], g_2[k]) \mid 1 \leq k \leq m(n-1), 1 \leq i \leq n, 1 \leq j \leq m\}$$

Complexité
Le matching à trois dimensions (3DM)

Donc chaque paire $g_1[k], g_2[k]$ doit être matchée avec un unique $u_i[j]$ ou $\bar{u}_i[j]$ qui n'apparait dans aucun triplet de $M' \setminus G$. Il y a exactlement $m \times (n-1)$ éléments ayant cette propriété et la structure de G garantit qu'ils peuvent toujours être couverts en choisissant dans $M' \cap G$ de manière appropriée.

En d'autres termes, G garantit, que si un sous-ensemble de $M \setminus G$ satisfait toutes les contraintes imposées par les lignes du composant "valeur de vérité" et du composant "test de satisfaction", alors ce sous-ensemble peut-être étendu en un matching pour M.

Complexité
Le matching à trois dimensions (3DM)

En résumé, on a définit :

- $W = \{u_i[j], \bar{u}_i[j] \mid 1 \leq i \leq n, 1 \leq j \leq m\}$
- $X = A \cup S_1 \cup G_1$ avec :
  - $A = \{a_i[j] \mid 1 \leq i \leq n, 1 \leq j \leq m\}$
  - $S_1 = \{s_1[j] \mid 1 \leq j \leq m\}$
  - $G_1 = \{g_1[j] \mid 1 \leq j \leq m(n-1)\}$
- $Y = B \cup S_2 \cup G_2$ avec :
  - $B = \{b_i[j] \mid 1 \leq i \leq n, 1 \leq j \leq m\}$
  - $S_2 = \{s_2[j] \mid 1 \leq j \leq m\}$
  - $G_2 = \{g_2[j] \mid 1 \leq j \leq m(n-1)\}$

et on a donc :

$$M = (\bigcup_{i=1}^{n} T_i) \cup (\bigcup_{j=1}^{m} C_j) \cup G$$

On a bien $M \subseteq W \times X \times Y$ et vu que $|M| = 2mn + 3m + 2m^2n(n-1)$, on peut construire M en temps polynomial.

Des commentaires ci-dessus, il est clair que M ne peut pas contenir de matching si C n'est pas satisfaisable.

Montrons maintenant que si C est satisfaisable alors M contient un matching.

Complexité
Le matching à trois dimensions (3DM)

Soit $f : U \rightarrow \{0, 1\}$ une valuation des variables propositionelles telle que pour toute clause c_j, on a $f \models c_j$. Et soit $z_j \in \{u_i, \bar{u}_i\} \cap c_j$, un littéral de la clause c_j tel que $f \models z_j$. Un tel littéral doit exister vu que f est une valuation propositionelle qui satisfait chaque clause.

On construit alors $M' \subseteq M$ de la façon suivante:

$$M' = \begin{array}{l} \bigcup_{f(u_i)=1} T_i^t \bigcup_{f(u_i)=0} T_i^f \\ \bigcup(\cup_{j=1}^{m} \{(z_j[j], s_1[j], s_2[j])\}) \bigcup G' \end{array}$$

où G' est choisi de façon appropriée dans les lignes de G de telle sorte que chaque élément $g_1[k], g_2[k]$ apparaissent une et une seule fois et chaque $u_i[j], \bar{u}_i[j]$ n'ayant pas encore été choisi apparaisse une et une seule fois.