# Optimisation Discrete

## Notions sur la Théorie de la Complexité
## et
## Introduction à la resolution des problémes d'optimisation combinatoire

Sonia Cafieri

ENAC

sonia.cafieri@enac.fr



# Outline

1. Notions sur la Théorie de la Complexité des Algorithmes
   - Machines de Turing
   - Classes de Complexité
2. Resoudre un Probleme d'Optimisation Discrete
   - Remarques préliminaires
   - Introduction aux méthodes de resolution

# Introduction

Peut-on connaitre la difficulté (ou la *complexité*)
d'une réponse par un algorithme à un problème
formulé de manière mathématique ?

Est-il possible énoncer *formellement* et rigoureusement ce qu'est
la complexité d'un algorithme ?

(avec une approche indépendante des facteurs matériels, comme le temps d'accès à la mémoire, le langage de programmation et le compilateur utilisé)



# Théorie de la Complexité

**Théorie de la Complexité :**

Le but : donner une *évaluation du temps de calcul ou de l'espace de calcul* nécessaire en fonction de la taille du probleme, qui sera notée $n$.

La théorie de la complexité établit des hiérarchies de difficultés entre les problèmes algorithmiques, dont les niveaux sont appelés des *classes de complexité*.

# Modèles de calcul

L'analyse de la complexité est étroitement associée à un modèle de calcul.

L'un des modèles de calcul les plus utilisés :
$\Rightarrow$ Machines de Turing.

Des autres modèles de calcul :

- les fonctions récursives
- le lambda-calcul
- les automates cellulaires.



# Machines de Turing

La machine de Turing : un modèle de calcul qui a été proposé en 1936 par Alan Turing.

La *thèse Church-Turing* postule que : *tout problème de calcul fondé sur une procédure algorithmique peut être résolu par une machine de Turing*.

Cette machine peut être adaptée pour simuler la logique de tout algorithme, et est particulièrement utile pour expliquer les fonctions d'une CPU dans un ordinateur.

Un calcul est constitué d'étapes élémentaires :
*à chacune de ces étapes, pour un état donné de la mémoire de la machine, une action élémentaire est choisie dans un ensemble d'actions possibles.*

Exemple d'une règle d'une machine de Turing :
"Si vous êtes dans l'état 2 et que vous voyez un 'A',
le changer pour un 'B' et aller à gauche."

## Machines de Turing

### Definition (Machines Déterministes)

Les Machines Déterministes sont telles que chaque action possible est *unique*;

l'action à effectuer est dictée de manière unique par l'état courant de celle-ci.

Ces machines correspondent tout à fait aux ordinateurs (avec un programme écrit dans un langage impératif quelconque).

### Definition (Machines non-Déterministes)

Les Machines non-Déterministes sont telles qu'il va exister des *choix possibles* pour effectuer une action;

l'action à effectuer est à considérer parmi un choix possibles d'actions et l'algorithme fera toujours le bon choix.

De telles machines n'existent pas et son purement abstraite; elles servent pour la définition de la théorie de la complexité.



## Algorithmes polynomiaux

### Definition

Une fonction $f(n)$ est $O(g(n))$ s'il existe une constante $c$ tel que

$$|f(n)| \leq c|g(n)| \quad \forall n \geq 0.$$

Un algorithme polynomial en temps est un algorithme dont la complexité est $O(p(n))$ pour une fonction polynomiale $p(n)$ de la taille des données.

Un algorithme pour lequel la fonction de complexité ne peut pas etre bornée de cette manière est dit algorithme exponential.

# Classe P et NP

**Definition (Classe P)**

Un problème est dit de classe P s'il existe un algorithme de *complexité polynomiale* en temps (par rapport à la taille de ses données) *pour le résoudre*.

Exemple :

- le classement des éléments d'un ensemble par ordre croissant ou décroissant



# Classe P et NP

**Definition (Classe NP)**

Un problème est dit de classe NP si il est *vérifiable* (i.e. on peut prouver qu'une réponse est bien la solution du problème) par un algorithme de *complexité polynomiale* en temps (par rapport à la taille de ses données)

Equivalentement : NP est l'ensemble des problèmes de décision qui peuvent être résolus par un algorithme *non-déterministe de complexité polynomiale* en temps.

L'équivalence des deux définitions :

un algorithme sur une machine non déterministe consiste en deux phases

- la première consiste en une conjecture sur la solution qui est généré de manière non-déterministe,
- la seconde consiste en un algorithme déterministe qui vérifie la solution.

## Classe P et NP

Exemple de problème NP :

- le problème général SAT de satisfaction de formule logique
- le problème du voyageur de commerce
- les problèmes de coloration de graphes



## Classe P et NP

**Proposition**

$$P \subseteq NP$$

Une question ouverte :

$$P = NP \quad \text{ou} \quad P \neq NP \;?$$

# Classe NP-complet

## Definition (NP-Complet)

Un problème est NP-Complet si il est dans NP est si il existe un *transformation polynomiale* qui transforme ce problème en un problème de la classe *NP-Complet*.

Pour montrer qu'un problème est NP-Complet, il suffit donc de montrer que par une *réduction polynomiale* il se ramène à un problème connu de la classe des problèmes NP-Complet.

Définition incrémentale : il a fallu donner un premier problème de la classe

$\Downarrow$

Théorème (de Stephen Cook) : le *problème SAT* est NP-Complet.



# Classe NP-complet

La classe NP-Complet est la classe la plus étudiée car elle constitue la classe des problèmes les plus difficiles de NP (c'est-à-dire ceux qui n'ont pas d'algorithmes polynomiaux connus pour les résoudre).

De plus, ils sont tous reliés par des transformations (ou réductions) polynomiales.

Pour montrer que $P = NP$, il suffit de trouver un algorithme polynomial pour résoudre un problème classé NP-Complet.

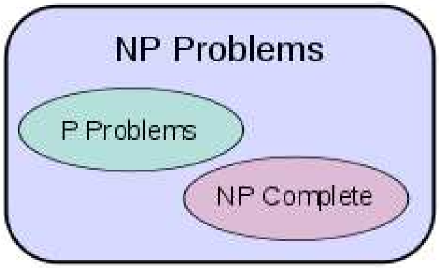
Les théoriciens de cette discipline pensent en général que

$$P \neq NP$$

.

# Classe NP-complet

On qualifie de NP-complets les problèmes de décision,
on qualifie de NP-difficiles (NP-hard) les problèmes d'optimisation.






# Resoudre par Programmation Lineaire continue?

PLNE : la fonction objectif et les contraintes sont des relations linéaires
$\Rightarrow$ ce ressemble exactement à un programme linéaire standard,
sauf que les variables ne peuvent pas prendre des valeurs réelles.

Est-ce que les modèles entieres peuvent être résolus par des méthodes adaptées à des valeurs réelles des variables?

*Rounding* : résoudre le problème par programmation linéaire standard et ensuite arrondir les valeurs non entières à la valeur entier la plus proche.

$\Downarrow$

N'est pas une bonne strategie !
on ne peut pas utiliser des solutions à valeurs réelles afin d'optimiser des modèles entières.

# Resoudre par Programmation Lineaire continue?

Exemple (Hillier et Lieberman) :

$$\begin{array}{ll} \min & Z = x_1 + 5x_2 \\ \text{s.t.} & x_1 + 10x_2 \leq 20 \\ & x_1 \leq 2 \\ & x_1, x_2 \text{ integer} \end{array}$$

La solution de programmation linéaire donne $Z = 11$ en (2, 1.8).

L'arrondissement de $x_2$ a la plus proche valeur entière, soit $x_2 = 2$, donne (2, 2) qui est infaisable par la première contrainte.

L'arrondissement de $x_2$ dans la direction opposée, c'est à dire $x_2 = 1$, donne (2, 1) avec Z = 7. Mais ce n'est pas le point optimal!

Le point entier (réalisable) optimal est à (0, 2) où $Z = 10$.



# Remarque

Si la solution de la relaxation continue $R_P$ de $P$ est entiere, $\bar{x}* \in \mathbb{Z}^n$,

alors cette solution c'est aussi la solution optimale de $P$

(*total unimodularity property*)

$\Downarrow$

on peut resoudre $P$ en resolvant simplement $R_P$

La resolution de la relaxation continue (PL) est souvent
la base de méthodes de resolution du PLNE.

# Les principales idees algorithmiques

- énumération “intelligent” de toutes les solutions
  *(Branch & Bound)*
- utilisation des méthodes pour PL (algorithme du simplexe) en tant que démarche algorithmique principale
- ajouter progressivement des contraintes au problème
  *(cutting planes)*
- alternativement, trouver seulement une solution approchée du problème
  *(heuristiques)*



# Méthodes de resolution

## Méthodes exactes

- garantissent de trouver une solution optimale pour une instance ou, s’il n’y a pas de solution, la preuve de son infaisabilité
- effectuent des recherches exhaustives
- sont généralement basés sur l’exploration d’un arbre binaire de recherche
- ne vérifient pas toutes les possibilités (ils utilisent généralement des stratégies visant à accélérer la recherche)
- leurs performances peuvent être limitées pour les grandes instances

## Méthodes approchées

- ne garantissent pas de trouver une solution optimale
- sont rapides à trouver une solution (si en trouvent)
- sont souvent basées sur des recherches locales et des algorithmes évolutionnaires

# Bibliography I


Michael R. Garey and David S. Johnson.
*Computers and Intractability: A Guide to the Theory of NP-Completeness.*
W. H. Freeman, 1979.

Frederic Messine.
*Notions sur la Theorie de la Complexité des algorithmes.*
2010.

John W. Chinneck.
*Practical Optimization: a gentle introduction*
`www.sce.carleton.ca/faculty/chinneck/po.html`, 2010.

L.R. Foulds.
*Combinatorial Optimization for Undergraduates*
Springer-Verlag, 1984.