Optimisation Combinatoire

Jean-Philippe Préaux

Définition

- Un problème d'optimisation combinatoire, consiste à minimiser une fonction *f* de domaine *S* fini. $f : S \to \mathbb{Z}$ ou $\mathbb{R}$

  c'est-à-dire trouver un minimum $s_0 \in S$

$$f(s_0) = \min_{s \in S} \{ f(s) \}$$

  selon le contexte, *f* est la *fonction de coût*, la *fonction économique* ou la *fonction objectif* (…)

- Avec une telle définition, la classe des problèmes d'O.C. est très large.
- Exemple : un *problème d'existence* c'est déterminer si il existe *s* dans *S* tel que *s* vérifie une propriété *P(s)*.

  Il s'agit d'un problème d'optimisation combinatoire appliqué à *–f* si l'on considère :

$$f : S \to \{0,1\}$$

$$f(s) = 1 \text{ si } P(s) \text{ vrai}$$

$$f(s) = 0 \text{ sinon}$$

Remarques

- On pourrait changer dans la définition ‘minimiser’ par ‘maximiser’, car minimiser *f* c’est maximiser *–f*.
- Il est toujours possible d’apporter une solution en calculant toutes les valeurs de *f* (car *S* est fini). Seulement le temps de calcul sera proportionnel au cardinal de *S* (qui sera en général énorme !). On rejette ce type de solution : on cherche les meilleures solutions.
- Apporter une solution à un problème très particulier n’a pas de grand intérêt. On met en évidence des classes de problèmes généraux. L’expérience montre que tous les problèmes connus s’y ramènent.

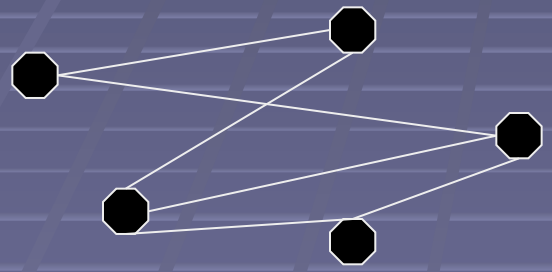
Difficulté de l'O.C.

- Relativement récente (car inspirée par le monde de l'entreprise et de l'ingéniérie de pointe) ; sa généralité recouvre une très large classe de problèmes.
- L'analyse mathématiques ne s'applique pas : la perte de différentiabilité et de continuité entraîne celle d'outils puissants.
- Tenter d'interpoler *f* sur un domaine continu est voué à l'échec. De nombreux exemples sont connus.

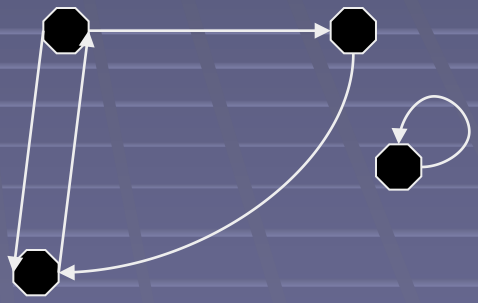
Difficulté de l'O.C.

- Il y a des problèmes faciles que l'on sait résoudre 'rapidement'.
- D'autres, malgré l'effort acharné depuis plus de quarante ans de centaines de chercheurs éminents n'ont toujours pas de solution 'satisfaisante', comme le problème du voyageur de commerce. On ne sait surtout pas si une 'bonne' solution existe.
- Indissociable de la 'théorie de la complexité', il est assez admis que ces problèmes difficiles seront de toute première importance dans le développement des sciences dures (math, info) du siècle naissant.

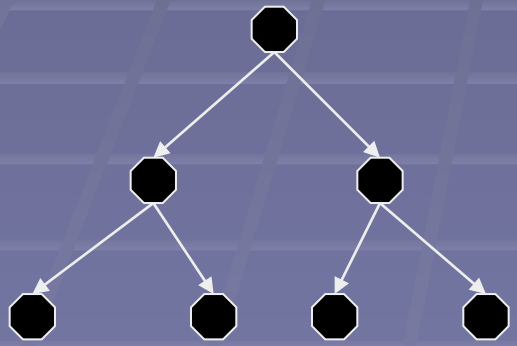
- Une partie importante des problèmes d'O.C. peuvent se formaliser en utilisant la théorie des graphes, dans toutes ses déclinaisons :



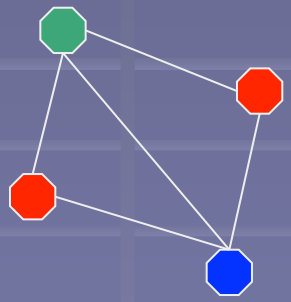
Graphe non orienté, connexe



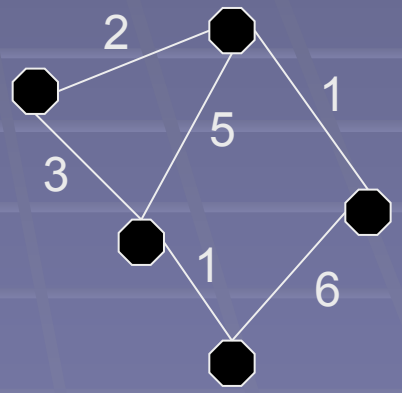
Graphe orienté, 2 composantes connexes



arborescence



Graphe colorié



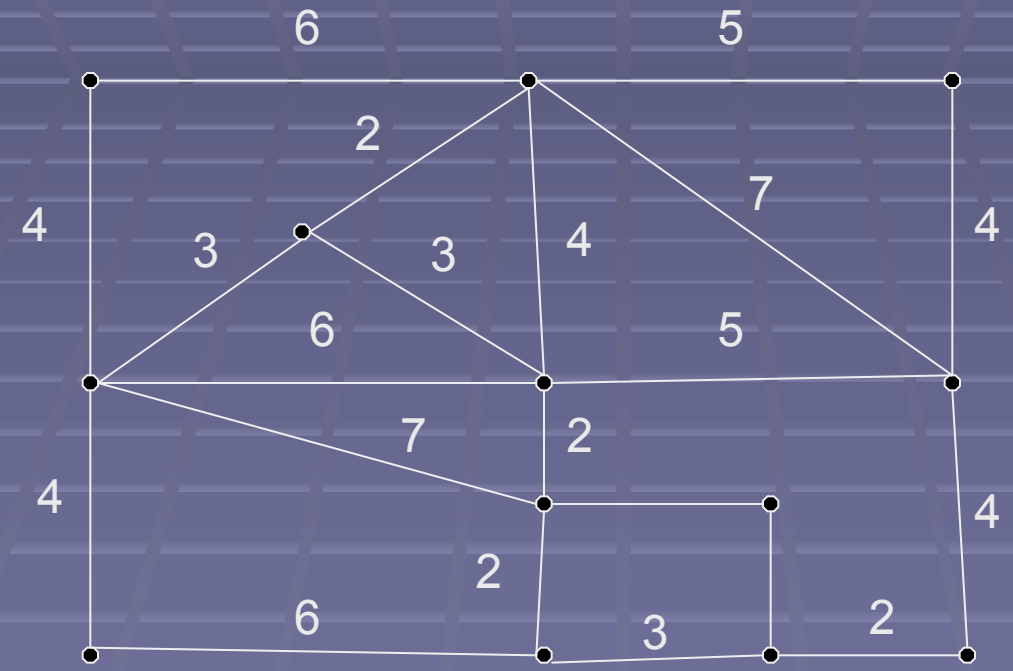
Etc.....

Graphe pondéré

Problèmes célèbres

- Le problème du plus court chemin :
- Peut s'écrire : Données *n* villes, des routes les reliant et leur longueurs, trouver le plus court chemin de la ville *i* à la ville *j*.
- Se formalise : Dans un graphe non orienté pondéré, trouver le plus court chemin du sommet *i* au sommet *j*.
- 'Bonne solution' (en $O(n^2)$) : Algorithme de Ford-Dijkstra (ou 'de marquage').

Le problème du postier



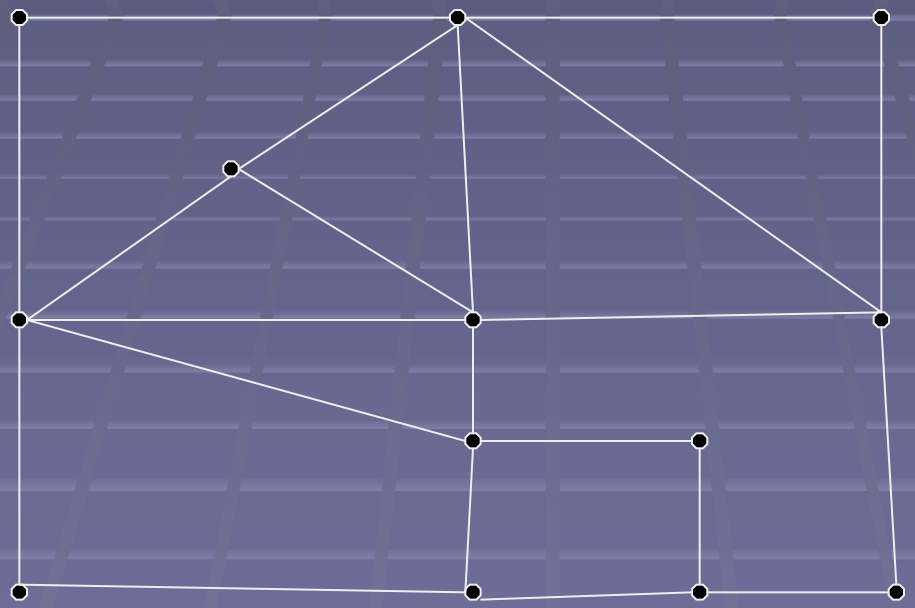
Donnée : Un graphe pondéré (rues à parcourir, tournée à faire, etc…)

Problème : Trouver un chemin passant au moins une fois par chaque arête, et de poids total minimal.

Problème facile !!

Jean-Philippe Préaux http://www.i2m.univ-mrs.fr/~preaux

Variante : Les cycles Eulériens



Donnée : Un graphe

Problème : Trouver un chemin (s'il existe !!) passant exactement une fois par chaque arête. (Ici il n'en existe pas)

(Ex : Tracer une figure sans lever la plume, en ne repassant jamais sur une ligne)

- Problème de coloration minimale

  Etant donné un graphe, peut-on colorer ses sommets avec *k* couleurs, de façon à ce que deux sommets adjacents soient de couleur différente.

Ex : Coloration des cartes géographiques. Quand le graphe est planaire au plus 4 couleurs suffisent (c'est le célèbre théorème des 4 couleurs.)

C'est un problème difficile !!

- Si l'on énumère toutes les possibilités, par exemple avec *n* sommets et 2 couleurs, il y a 2^n possibilités.
- En supposant que l'examen de chaque possibilité prend une fréquence de calcul, sur un processeur 1Ghz, on aurait les temps de calcul :

Nb de Sommets	Nb de Possibilités	Temps de calcul
10	10^3	10^{-6} sec.
20	10^6	10^{-3} sec.
30	10^9	1 sec.
40	10^{12}	17 min.
50	10^{15}	11,5 jours
60	10^{18}	32 ans
70	10^{21}	3200 ans

- Problème du voyageur de commerce.

  Parcourir en avion *n* villes en passant une seule fois par chaque ville, et en minimisant la distance parcourue.

  Modélisation :

  Chercher un cycle hamiltonien dans une graphe complet pondéré, de coût minimal.

  Problème très difficile. Dans un sens le plus difficile des problèmes raisonnables (*problème NP-complet*)

- Des problèmes non modélisés par des graphes :
- Problème du sac à dos :

  n objets de poids $p_1,p_2,\ldots,p_n$, et de profit $c_1,c_2,\ldots,c_n$.

  Un sac à dos de capacité C.

  Trouver un ensemble d'objets de poids total <=C de profit total maximal.

  Si $p_i = c_i$, c'est un problème d'empilement.

- Le problème de satisfaisabilité

  Donné un ensemble de clauses, existe-t-il une assignation de leur variable libre qui les rendent toutes simultanément vraies.

  De grande importance en IA !!

« Il est crucial pour l'informatique théorique et l'intelligence artificielle. Par exemple dans un système expert le problème de savoir si un but donné peut être prouvé en partant d'un certain nombre de faits peut s'y ramener ».

Ce fût le premier problème à être montré NP-complet [Cook, 1971]

FIN DE L'INTRODUCTION

Fondement théorique : Complexité d'un algorithme.

- Fait admis : Le concept d'**algorithme** est bien défini (*Church, Gödel, Turing, '30).*
- Informellement : Un algorithme prend en entrée des données, et par un procédé déterministe retourne en sortie le résultat.
- Il ne boucle jamais indéfiniment, et retourne toujours un résultat… Même si le temps de calcul théorique prend 30 milliards d'années…

- Question : comment définir et mesurer la ‘complexité’ d’un algorithme,
- Elle devrait correspondre à l’idée de temps de calcul, mais cela indépendamment de la machine utilisée, ou du langage de programmation employé (…)??
- Même mesurer une ‘moyenne’ des temps de calcul sur diverses machines ne serait pas acceptable, car trop temporel.
- Ce que l’on veut mesurer c’est si l’algorithme gardera, même dans les pires cas, des temps de calcul raisonnables.

Fonction de compléxité

- Une *opération élémentaire :*
  - +, -, *, /.
  - connecteurs logiques : ou, et, non, xor.
  - décalages de bit (shift), permutation de n bits.
  - Tests booléens, <, >, ==, not==, etc…
  - …
- La *fonction de compléxité* d'un algorithme est une application $f : \mathbb{N} \rightarrow \mathbb{N}$. (ou $f : \mathbb{N}^p \rightarrow \mathbb{N}$)
  - *f(n)* est le max du nombre d'opérations élémentaires nécessaire au calcul pour une entrée de taille n.
  - C'est le nombre de calculs nécessaires pour une entrée de taille n (ou n entrées de taille 1…), dans le pire des cas !!

- La taille de l'entrée peut-être comprise comme la quantité d'espace mémoire nécessaire à son stockage. Il importe peu pour notre propos de la définir proprement (…).
- Par exemple pour un graphe on aura coutume de considérer sa taille comme le couple (N,M).
- On pourra par exemple avoir des algorithmes d'*ordre N, ou N log(M), etc…*

Ordre d'une application à valeur entière.

- Objectif : Etablir une relation d'équivalence entre applications à valeurs entières.
- S'intéresser uniquement à leur comportement asymptotique :
- Il s'agit d'une mesure de leur vitesse de divergence vers +oo

Ordre d'une application

- Soient $f,g:\mathbb{N}\longrightarrow\mathbb{N}$ 2 applications, on dit que *f* est un 'grand O' de *g* si il existe un nombre *C*>0 tel que : $\exists n_0,\forall n\geq n_0$

$$f(n)\leq Cg(n)$$

On note : $f=O(g)$

C'est une relation d'ordre.

- *f* et *g* ont même ordre si : $f=O(g)$ et $g=O(f)$

C'est une relation d'équivalence notée $f=\Theta(g)$.

- $f = \Theta(a.f + b)$ (a,b constants et a > 0) .
- Les fonctions bornées >0 ont même ordre noté O(1).
- $g = O(f) \Rightarrow f + g = \Theta(f)$
- Ces relations sont compatibles avec + et **x**.

- Si f polynôme de degré *d* alors $f = \Theta(n^d)$
- En notant $f \prec g$ si $f = O(g)$ et $g \neq O(f)$:

  $\ln(n) \prec n \prec n^2 \prec n^3 \prec \ldots \prec e^n \prec n! \prec n^n$

Compléxité d'un algorithme

- C'est l'ordre de sa fonction de compléxité.
- Un algorithme en O(1) a un temps de calcul borné.
- Un algorithme de compléxité polynomiale est un algorithme dont l'ordre est un $O(n^{\alpha})$
  - $O(\ln(n))$ très rapide
  - $O(n)$ rapide
  - $O(n\ln(n))$ assez rapide
  - $O(n^2)$ correct
  - …
- Un algorithme de complexité polynomiale est un algorithme correct. Au dela c'est un algorithme non efficient (d'ordre sur-polynomial, exponentiel, ou plus encore …)

Complexité exponentielle

- Un algorithme sera dit de complexité exponentielle si son ordre est plus grand que tout polynôme.
- Il peut avoir un ordre exponentiel $a^n, n > 1$, un ordre plus grand $n!$, n^n ou un ordre plus petit $n^{\ln(n)}$.
- Dans la pratique cela signifie qu'au-delà d'une certaine taille de données, les calculs ne sont plus efficients.

	20	50	100	200
$1000.n$	0.02s	0.05s	0.1s	0.2s
$100.n^2$	0.04s	0.25s	1s	4s
$10.n^3$	0.02s	1s	10s	1min
2^n	1s	36 ans	-	-
3^n	58 min	2.10^{11} ans	-	-
$n!$	77100 ans	-	-	-

Processeur 1 Ghz

- : supérieur à 1000 milliards d'années

Jean-Philippe Préaux

http://www.i2m.univ-mrs.fr/~preaux

Conséquence

- Il est faux de croire qu'il suffit d'accroître la puissance de son ordinateur pour résoudre un problème. Passer du calcul de n à n+1 pourrait nécessiter de multiplier par 1000 (ou +) les capacités de son ordinateur.
- Un bon programmeur doit fuir la complexité exponentielle 'comme la peste'.
- Il faut toujours éviter les méthodes consistant à analyser tous les cas : l'énumération des parties d'un ensemble ($O(2^n)$), des affectations de n objets à n autres ($O(n!)$), de n décisions invoquant k possibilités ($O(k^n)$).

Problèmes **P** et **NP**

- La classe **P** est la classe des problèmes pour lesquels il existe un algorithme de complexité au plus polynomiale pour déterminer une solution.
- La classe **NP** est la classe des problèmes pour lesquels il existe un algorithme de complexité au plus polynomiale pour déterminer si un candidat donné est ou non solution.
- Clairement : $\mathbf{P} \subset \mathbf{NP}$.

  On conjecture : $\mathbf{P} \subsetneq \mathbf{NP}$,

  (c'est l'un des '6 grands problèmes du siècle').
  A une importance fondamentale notamment en cryptographie.

Problèmes **NP**-complets

- Un problème *A* dans **NP** est dit **NP**-complet si tout problème *B* dans **NP** admet un algorithme de complexité au plus polynomiale qui déduit d'une solution à *A* une solution à *B*.
- De tels problèmes existent !! (ex : PVC, satisfaisabilité des clauses,...). Ce sont les problèmes courants les plus difficiles.
- Ex : trouver une solution polynomiale au PVC montrerait **P=NP**... (et permettrait en particulier de 'casser' toutes les clés actuelles (*RSA, DES,...)* de cryptographie à clé publique.)