ACT

Sophie Tison
Université Lille
Master1 Informatique

COMPLEXITÉ DES PROBLÈMES

- Rappels
- Paradigmes d'algorithme
- Complexité des problèmes
  - Généralités
  - La classe P
  - La classe NP
- Algorithmique Avancée

QUE FAIRE FACE À UN PROBLÈME *NP*-DUR?

- s'acharner à chercher avoir un algorithme polynomial en espérant montrer $P = NP$ ...
- abandonner l'idée d'avoir un algorithme polynomial
  - pour certains problèmes NP-durs, il exise des algorithmes pseudo-polynomiaux.
  - utiliser la spécificité des données; on peut parfois concevoir des algorithmes "efficaces à paramètres fixés" (algorithmes FPT, fixed-parameter tractable)...
  - utiliser des méthodes "améliorées" d'énumération des solutions: exploiter la symétrie des solutions, élaguer l'arbre des solutions (pruning), prioriser les branches les plus prometteuses - cf branch and bound,
  - utiliser les solveurs existants, par exemple des solveurs SAT ou de programmation linéaire en entiers.
- abandonner l'idée d'avoir un algorithme qui donne toujours la solution optimale

ABANDONNER L'IDÉE D'AVOIR UN ALGORITHME QUI DONNE TOUJOURS LA SOLUTION OPTIMALE

Le cadre est celui des problèmes d'optimisation: on cherche à construire une solution à un problème qui optimise une fonction objectif.

On peut utiliser un algorithme d'approximation: il fournit toujours une solution mais pas forcément une solution optimale. Bien sûr, on souhaite qu'il soit efficace.

DÉTERMINISTE OU NON

Un algorithme d'approximation peut être:

- déterministe - pour une entrée donnée, il donnera toujours la même solution (heuristiques gloutonnes, optimum local, tabou....)
- non déterministe: recuit simulé, algorithme génétique,...

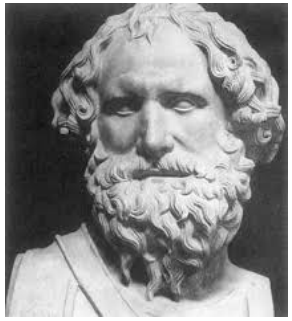
"AD HOC" OU NON

Un algorithme d'approximation peut être:

- basé sur un critère glouton propre au problème- on parle d'heuristique(gloutonne).
- l'instanciation d'une méta-heuristique -algorithmes génétiques, recuit simulé, tabou..., souvent basée sur des analogies avec des phénomènes physiques (recuit simulé) ou naturels (algorithmes génétiques, colonies de fourmis, essaims...)

HEURISTIQUE

Art d'inventer, de faire des découvertes.
Du grec ancien heurískô (ń trouver ż), dont est aussi issu eurêka.



COMMENT MESURER LA QUALITÉ D'UN ALGORITHME D'APPROXIMATION?

NOTIONS DE GARANTIES

Soit $Opt(I)$ la solution optimale pour l'instance I, $A(I)$ la solution produite pour I par l'algorithme A et f la fonction objectif à optimiser.

On définit le Ratio de garantie.

C'est la borne sup de $\frac{f(A(I))}{f(Opt(I))}$ pour toute instance I si la fonction objectif est à minimiser, ou de $\frac{f(Opt(I))}{f(A(I))}$ pour toute instance I si la fonction objectif doit être maximisée.

Le ratio de garantie

Quelques remarques

Le ratio de garantie n'est pas toujours fini.

Il est toujours supérieur ou égal à 1.

Si il est égal à 1 c'est que l'algorithme est exact!

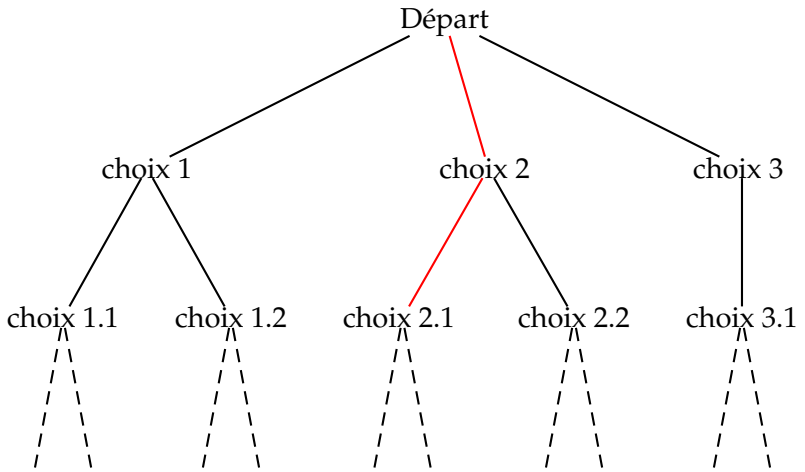
Plus ce ratio est proche de 1, meilleure est la garantie de l'approximation.

Le ratio "exprime le pire des cas".

Le principe d'une heuristique gloutonne

Le principe est proche de celui des algorithmes gloutons: construire incrémentalement une solution en faisant les choix qui paraissent judicieux à défaut de pouvoir garantir qu'ils sont optimaux.

Heuristiques Gloutonnes



L'heuristique gloutonne fait ses choix localement pour ne pas explorer tout l'arbre.

L'exemple du BinPack ou de la mise en sachets:

Donnée:
n , le nombre d'objets
$p_1, p_2, ..., p_n$ les poids
cap la capacité d'un sac

Objectif? mettre en sachets en utilisant le moins de sacs, i.e. trouver:

aff: $[1..n] -> [1..n]$

tel que $\sum_{aff(i)=s} p_i <= cap$ pour tout sac s

Qui minimise le nombre de sacs, i.e. $max(aff(i))$

UN EXEMPLE

Donnée:
$N = 5$
$p_1 = 2, p_2 = 3, p_3 = 3, p_4 = 2, p_5 = 2$. les poids
$cap = 6$
Q? Optimum:

mettre 1,4,5 dans un sac, 2 et 3 dans un autre

Le problème est $NP-$dur: il y a peu de chances d'avoir un algorithme exact polynomial!

UNE HEURISTIQUE ?

L'HEURISTIQUE DE LA CAISSE

APPELÉE ÉGALEMENT *NextFit*

```
ns=1;//le premier sac
c=cap;//place restante dans le sac courant
for i de 1 à n do
    if (p_i ≤ c) then
        l'objet rentre dans sac courant
        aff(i)=ns; c=c − p_i;
    else
        // on le met dans un nouveau sac
        ns++; aff(i)=ns;c=cap-p_i;
end
```

L'algorithme est en $O(n)$ (taille de la donnée $>$ n).

GARANTIE?

Soit k le nombre de sacs utilisés par l'heuristique sur une instance:
le poids des objets des sacs 1 et $2 > cap$
Le poids des objets des sacs 3 et $4 > cap$
$\vdots$
le poids des objets des sac $2 * i - 1$et $2 * i > cap$
le poids des objets des sac $2 * (k/2) - 1$ et $2 * (k/2) > cap$
Donc le poids des objets $> cap * k/2$
Donc la solution optimale utilise au moins$(k/2 + 1)$sacs
$\frac{f(A(I))}{f(Opt(I))} \leq 2$ pour toute instance I.

Le ratio est au plus 2.

LE RATIO EST BIEN 2

Soit
$p_1 = k, p_2 = 1, p_3 = k, p_4 = 1, p_5 = k....p_{4k} = 1$
$cap = 2 * k$

L'algorithme glouton va utiliser $2k$ sacs:
x 1, x 1,...

Alors que l'optimale va utiliser $k + 1$ sacs:
x x, x x,....,1 1

Donc le ratio est supérieur à $\frac{2k}{k+1}$ pour tout k.
Comme le ratio est au plus 2, le ratio est donc 2;

D'AUTRES IDÉES D'HEURISTIQUE?

- Heuristique 1 (NextFit)
- Heuristique 2 (FirstFit) Pour chaque objet, on regarde si il rentre dans un des sacs créés: si oui, on le met dans le premier qui convient; sinon, on créee un nouveau sac et on y met l'objet;
- Heuristique 3 (BestFit) Pour chaque objet, si il rentre dans un des sacs créés, on le met dans le celui qui est le plus rempli parmi ceux qui conviennent. Sinon, on crée un nouveau sac et on y met l'objet.
- Heuristique 4 (WorstFit) Pour chaque objet, si il rentre dans un des sacs créés, on le met dans le celui qui est le moins rempli. Sinon, on crée un nouveau sac et on y met l'objet.
- Pour chaque heuristique, on peut soit l'appliquer "online" -on range les objets au fur et à mesure de leur arrivée, soit offline: dans ce cas, on peut trier les objets par poids décroissants avant de les ranger.

D'AUTRES IDÉES D'HEURISTIQUE?

On peut essayer aussi de placer les "gros" objets de façon optimale en utilisant en algorithme exact, puis de placer les petits objets.
On obtient un schéma paramétré par la taille minimum des objets qu'on considère dans la première phase.

UNE GARANTIE $< 3/2$?

LE RETOUR DES RÉDUCTIONS POLYNOMIALES

> Si il existait une heuristique polynomiale de garantie strictement inférieure à 3/2 pour BinPack, on aurait P=NP.

En effet, si on avait une heuristique polynomiale de garantie <3/2, on aurait un algorithme polynomial pour Partition.

Pourquoi? (voir Tableau)

Peut-on toujours trouver un algo polynomial avec garantie pour un problème NP-dur?

Quel que soit le problème associé à un problème de décision *NP*, peut-on toujours trouver un algorithme polynomial déterministe d'approximation avec une garantie?

Non! Il y a plusieurs situations possibles:

Sans Garantie

Pour certains problèmes, on montre que, quel que soit le ratio de garantie constant, on ne peut pas trouver d'algorithme polynomial qui offre ce ratio de "garantie", sauf si $P = NP$.

C'est le cas du problème du voyageur de commerce, si on ne suppose pas que la distance vérifie l'inégalité triangulaire.

Avec une certaine Garantie

Pour certains on connaît des algorithmes polynomiaux qui offrent une certaine garantie. C'est le cas du BinPack.

Schéma d'approximations

Pour d'autres, on montre que quel que soit le ratio de garantie, on peut trouver un algorithme polynomial qui offre cette garantie.
On a dans ce cas deux types de schémas:

- dans le cas le moins favorable, le degré du polynôme dépend de la qualité de l'approximation souhaitée.
- Dans l'autre cas, on a un schéma complet polynomial d'approximation i.e. on a un algorithme polynomial qui accepte en entrée une instance du problème et une qualité de solution souhaitée et ressort une solution ayant cette qualité. C'est le cas du Knapsack.

En pratique, ce n'est pas forcément très intéressant car le degré du polynôme peut être grand!

UN AUTRE EXEMPLE

VertexCover
Donnée: (G=(S,A))
Q? Trouver C un sous-ensemble de S de cardinal minimum qui couvre A i.e. tel que toute arête a une extrémité dans S.

Le problème est NP-dur.

QUEL ALGORITHME CHOISIR?

C=EnsVide;
while A non vide **do**
  choisir une arête (u,v) de A;
  ajouter ses deux extrémités à C;
  éliminer de A tous les arcs dont une extrémité est u ou v.
end

ou?

C=EnsVide;
while A non vide **do**
  choisir une arête (u,v) de A;
  ajouter une des extrémités, par ex. u à C;
  éliminer de A tous les arcs dont une extrémité est u
end

QUE CHOISIR?

Quelle est celle qui offre la meilleure garantie?

- La première offre un ratio de 2: soit O une solution optimale, C la gloutonne: nécessairement O contient au moins la moitié des éléments de C!
- La deuxième: pas de garantie

HEURISITIQUES GLOUTONNES ET GARANTIES: BILAN

- Les heuristiques gloutonnes sont souvent simples et efficaces (linéaires, ou presque)
- Le ratio de garantie ... garantit une certaine qualité mais n'est qu'une indication correspondant au pire des cas!
- Certains problèmes NP-durs n'ont pas d'algorithmes polynomiaux d'approximation avec garantie, sauf si P=NP.

LES MÉTAHEURISTIQUES

DEUX GRANDES FAMILLES

- les méthodes par voisinage ou par parcours de l'espace des solutions
- les méthodes à base de populations

MÉTAHEURISTIQUES: LES MÉTHODES PAR VOISINAGE

Idée: transformer une solution petit à petit, explorer l'espace des solutions en partant d'un point et en se "déplaçant" de de proche en proche;

- Le hill climbing
- La méthode taboue
- Le recuit simulé

VOISINAGE

Explorer l'espace des solutions en se "déplaçant" de de proche en proche:

> Il faut définir une notion de voisinage.

EXEMPLES DEVOISINAGE

- Sac à dos: échanger x -au plus- objets contre y-au plus- autres.
- BinPack: échanger x -au plus- objets d'un sac contre y-au plus- autres d'un autre.
- TSP: modifier un peu -et seulement un peu- l'ordre de la tournée: voir TP.

Recherche d'un optimum local ou Hill-Climbing

On part d'une solution si possible "bonne" (par exemple donnée par une heuristique ci-dessus) et on balaie l'ensemble des voisins de cette solution;

- si il n'existe pas de voisin meilleur que notre solution, on a trouvé un optimum local et on arrête;
- sinon, on choisit le meilleur des voisins et on recommence.
- Une autre implémentation consiste non pas à passer au meilleur des voisins à chaque étape mais au premier meilleur voisin trouvé.

Le schéma d'algorithme

```
//Hillclimbing
Sol= solution obtenue par ex. par un glouton
//Variante non déterministe:
// Sol tirée au hasard;
tant que Sol a un voisin meilleur
   Sol <- son meilleur voisin
   //variante:
   // SOl <-un voisin meilleur qu'elle
 fin tant que;
```

La solution obtenue est un optimum local.

Bien Choisir son voisinage

Le voisinage doit être un bon compromis:

- Si le voisinage est trop petit, risque de se bloquer très vite dans un optimum "de mauvaise qualité"
- Si le voisinage est trop grand, chaque itération est très longue (surtout dans la première version)

Les inconvénients de la méthode

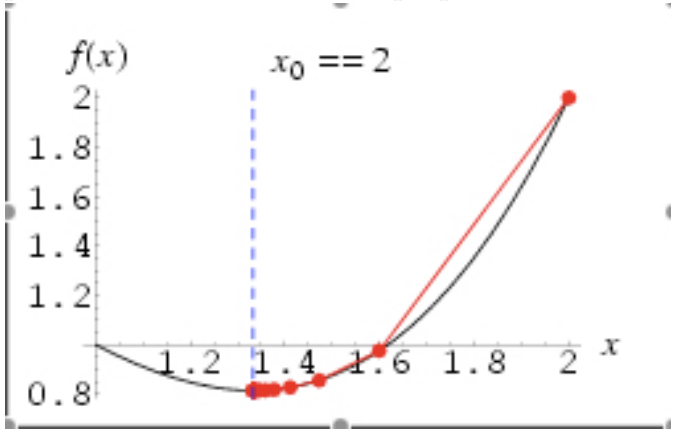
- On n'obtient qu'un optimum local.

  On peut diversifier en lançant plusieurs recherches à partir de solutions différentes.
- La convergence peut être lente.

  On peut fixer un nombre maxi d'itérations.

HILL CLIMBING ET DESCENTE DU GRADIENT

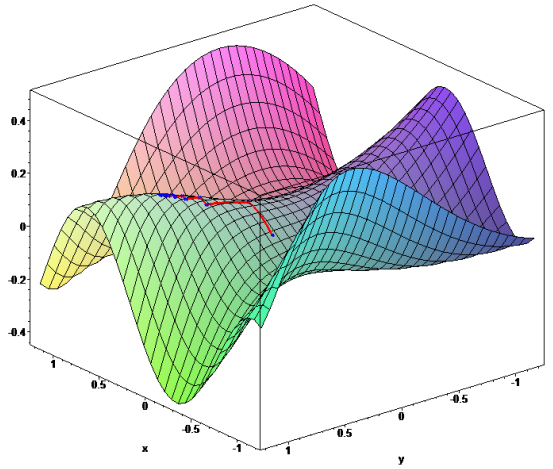
Cette méthode est le pendant dans le cas discret de la méthode du gradient.
pour minimiser $y = f(x)$ à chaque pas, $x- > x - \epsilon * f'(x)$



$x- > x - \epsilon * f'(x)$

HILL CLIMBING ET DESCENTE DU GRADIENT

En 3D, pour minimiser $z = f(x, y)$:



HILL CLIMBING ET DESCENTE DU GRADIENT

La méthode de descente du gradient et ses dérivées est très utilisée, par exemple pour calculer les paramètres d'un modèle (cf FAA, réseaux de neurones).

Comment éviter de rester bloqué dans un optimum local?

LA MÉTHODE TABOUE

Idée: permettre de s'échapper de l'optimum local.

On pourra aller chez un voisin moins bon: on choisit le meilleur de ceux qu'on n'a jamais visités, ou qu'on n'a pas visités depuis longtemps.

On maintient une liste -FIFO- taboue des derniers visités.

LE SCHÉMA DU TABOU

```
Sol= solution obtenue par ex. par un glouton ;
//Ou Sol tirée au hasard;
Meil=Sol;
nb_boucles=0;nb_boucles_sans_amelioration=0;
Init(Tabou)//Tabou:File, capacité=paramètre
loop
si Sol meilleure que Meil
  {Meil<-Sol;nbam=0;}
 Ajouter(Sol, Tabou); //Tabou File
  Sol <- son meilleur voisin non Tabou
 nb_boucles++;
 nb_boucles_sans_amelioration++;
 exit si nb_boucles> MAX 1
    ou si nb_boucles_sans_amelioration>MAX2;
end_loop;
return Meil;
```


LA LISTE TABOUE

- Gérée en File (FIFO)
- Si la liste est trop courte, on n'arrive pas à s'"échapper".
- Si la liste est trop longue, cela peut être coûteux de vérifier qu'une solution n'est pas taboue.
- On stocke souvent non pas les solutions, mais les transformations faites.

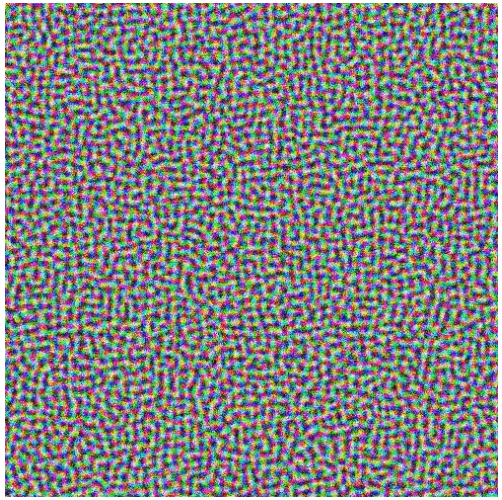
La méthode "taboue" peut donc être vue comme une généralisation de la recherche d'optimum local.

si N=0, on retombe dans le cas du hill-climbing;

Le recuit simulé

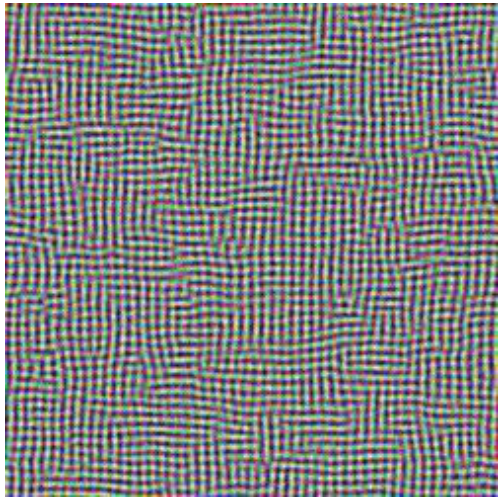
C'est une méthode non déterministe.
La méthode (simulated annealing) est inspirée de la physique: le recuit est utilisé pour obtenir des états de faible énergie.
Le principe est utilisé en métallurgie depuis des millénaires -on chauffe très fort le métal puis on le refroidit lentement pour obtenir un alliage sans défaut- ou dans la fabrication du verre.
Idée: au départ température haute, état instable. Il y a assez d'énergie pour changer de configuration même pour aller dans une moins stable; on "laisse les choses" s'organiser petit à petit et au fur et à mesure, on refroidit.

Images de Wikipedia



Pixels aléatoires de différentes couleurs "rangés" par un recuit avec un refroidissement rapide.

Images de Wikipedia



Pixels aléatoires de différentes couleurs "rangés" par un recuit avec un refroidissement lent.

- On part d'une solution quelconque.
- A chaque étape, on tire au hasard une solution voisine.
  - Si elle est meilleure, on l'adopte;
  - sinon on va adopter cette solution avec une probabilité liée à son augmentation de coût: plus le coût augmente, plus la probabilité de la garder sera faible- mais aussi au temps écoulé: au début, on accepte facilement un changement qui produit une solution plus coûteuse, à la fin on l'accepte très difficilement.

FAUT-IL ACCEPTER LE CHANGEMENT?

- On calcule l'augmentation de coût Δ.
- Puis on calcule $accept = E^{-\Delta/T}$, T étant une quantité que nous appellerons *la température*.
- On tire au hasard un réel p entre 0 et 1:
  - si p est inférieur à *accept*, on adopte la nouvelle solution,
  - sinon on garde l'ancienne.
- Donc plus T est élevée, plus on accepte le risque du changement.

LE SCHÉMA DU RECUIT SIMULÉ

```
init(sol);
T:=T0;
loop
  sol':=voisin_au_hasard(sol)
  if meilleur (sol',sol) then sol:=sol';
  else
    Delta=cout(sol') - cout(sol);
    --surcoût de sol'
    tirerproba(p); --p réel dans [0,1]
    if p<= e ^ (-Delta/kT)
    then sol:=sol';
    --on accepte sol' avec proba accept
    --sinon on ne change pas
    end if;
  T:=refroidissement(T);
end loop;
```


RÉGLAGE DE LATEMPÉRATURE

Une difficulté de cette heuristique réside dans le réglage des paramètres selon l'instance du problème considérée. Il faut:

- *choisir la température initiale:* La température initiale doit être assez élevée pour que *accept* soit assez grand au départ même si l'augmentation de coût est grande. C'est un paramètre fixé par l'utilisateur.
- *régler le refroidissement:* on peut par exemple multiplier T par un réel inférieur à 1 à chaque fois. Plus ce réel est proche de 1, plus le refroidissement est lent. Ce "coefficient de refroidissement" est lui aussi un paramètre fixé lors de l'utilisation.

LES ALGORITHMES GÉNÉTIQUES

Les algorithmes génétiques, créés par J. Holland (75) puis développés par David Goldberg, sont basés sur deux principes de génétique: la survie des individus les mieux adaptés et la recombinaison génétique. Le principe de base est de simuler

l'évolution d'une population de solutions avec ces deux règles pour obtenir une solution ou un ensemble de solutions les plus adaptées.

"A chaque génération, un nouvel ensemble de créatures artificielles est créé en utilisant des parties des meilleures solutions précédentes avec éssssssventuellement des parties innovatrices." (Goldberg)

LE SCHÉMA D'UN ALGORITHME GÉNÉTIQUE

```
créer la population initiale
--par exemple au hasard
boucle{
  sélectionner les individus
  --sélection des plus adaptés
   hybridation à l'aide d'opérateurs de croisement
   -- crossover
  (optionnel) effectuer des mutations aléatoires
   -- modification d'un gène (innovation)}
```

L'arrêt se fera par exemple quand la population ne s'"améliore" plus de manière significative ou après un nombre donné de générations.

Pour la mise en oeuvre il faut donc:

- choisir le codage (souvent binaire) d'une solution
- fixer la taille d'une population
- définir la fonction d'adaptation (fitness)
- définir les opérateurs de sélection, le principe étant: meilleure est notre adaptation, plus on a de chances d'être sélectionné
- définir les opérateurs de croisement
- définir éventuellement les opérateurs de mutation.

L'un des problèmes de la méthode est encore la convergence prématurée vers une population non optimale.
D'autres métaheuristiques inspirées par le monde du vivant ont été introduites, comme les colonies de fourmis.

CE QU'IL POURRAIT AU MOINS RESTER QUAND VOUS AUREZ TOUT OUBLIÉ

- Un peu plus de rigueur dans l'écriture d'algos: pourquoi ce que j'écris marche?
- comment décomposer le problème en sous-problèmes? Suis-je dans le cas de la programmation dynamique?
- Un peu de réflexion sur le coût des algorithmes: est-il efficace? Puis-je évaluer sa complexité?
- Un peu de réflexion sur la nature du problème: peut-on faire mieux? n'est-ce pas un problème connu (P ou non)? Est-il NP-dur? Puis-je le résoudre efficacement de façon exacte?


The End