Optimisation combinatoire : méthodes approchées

Master 1, IUP SI

Philippe Muller

2011-2012

Plan du cours

Objectif : maîtriser l'explosion combinatoire

- Retour les sur problèmes de complexité NP
- Méthodes de recherche incomplètes, méta-heuristiques locales
- Méthodes d'approximation
- Mise en pratique en TP sur un problème caractéristique

Quelques Problèmes des méthodes exactes/complètes

1. Les méthodes complètes/exactes explorent de façon systématique l'espace de recherche. $\rightarrow$ ne marche pas dans de nombreux problèmes à cause explosion combinatoire.
2. ces méthodes sont généralement à base de recherche arborescente $\rightarrow$ contrainte par l'ordre des noeuds dans l'arbre $\rightarrow$ impossibilité de compromis qualité/temps ($\neq$ algo "anytime")

Or il y a souvent des cas ou on peut se contenter d'une solution approchée, pourvu qu'elle soit suffisament "bonne". (ex du TSP, on ne veut pas forcement le minimum mais quelque chose d'assez court, par exemple $<=$ distance donnée).

Un autre principe : les méthodes incomplètes

- si on ne peut prouver que l'on a la meilleure solution,
- et si ce n'est pas grave si on trouve un optimum le plus rapidement possible, de façon assez bonne / à un critère déterminé.

on peux adopter deux types de stratégie :

- garder une méthode complète et stopper après un temps déterminé (mais suivant les problèmes ça n'est pas toujours possible)
- plonger dans l'arbre de recherche avec des heuristiques sans jamais revenir en arrière (mais : comment être sur qu'on va trouver vite une bonne solution ?)

Une famille de méthodes incomplètes : les méthode "locales"

Le principe d'une **recherche locale** est de

1. partir d'une solution sinon approchée du moins potentiellement bonne et d'essayer de l'améliorer itérativement.
   Pour améliorer une solution on ne fait que de légers changements (on parle de changement **local**, ou de solution **voisine**).
2. relancer la méthode plusieurs fois en changeant le point de départ pour avoir plus de couverture
3. tout problème est considéré comme un problème d'optimisation (même les problèmes de satisfaction : le coût à optimiser est alors le nombre de contraintes insatisfaites).

Quelques définitions

une solution est une affectation de toutes les variables du problème.

une solution optimale est une solution de coût minimal

un mouvement est une opération élémentaire permettant de passer d'une solution à une solution voisine (exemple : changer la valeur d'une variable, échanger deux variables).

le voisinage d'une solution est l'ensemble des solutions voisines, c'est à dire l'ensemble des solutions accessibles par un mouvement (et un seul).

un essai est une succession de mouvements.

une recherche locale est une succession d'essais.

Un algorithme de recherche locale typique

```
A* ← creer_une_solution()
for all  t=1 a max_essais  do
    A ← nouvelle_solution()
    for all  m=1 a max_mouvements do
        A' ← choisir_voisin(A)
        d ← (f(A')-f(A))
        if  acceptable(d) then
            A ← A'
        end if
    end for
    if  f(A*)>f(A) then
        A* ← A
    end if
end for
retourner A*,f(A*)
```

f : fonction à optimiser

Les paramètres à régler

- max_essais : le nombre d'essai à réaliser
- max_mouvements : ...
- creer_une_solution : génère une solution (aléatoirement ou non)
- nouvelle_solution : même chose
- choisir_un_voisin : choisit dans le voisinage de la solution courante (c'est généralement ce qui caractérise principalement une méthode de recherche locale)
- acceptable : accepte ou pas la nouvelle solution générée

Illustration dans le cas de fonctions continues : 1D

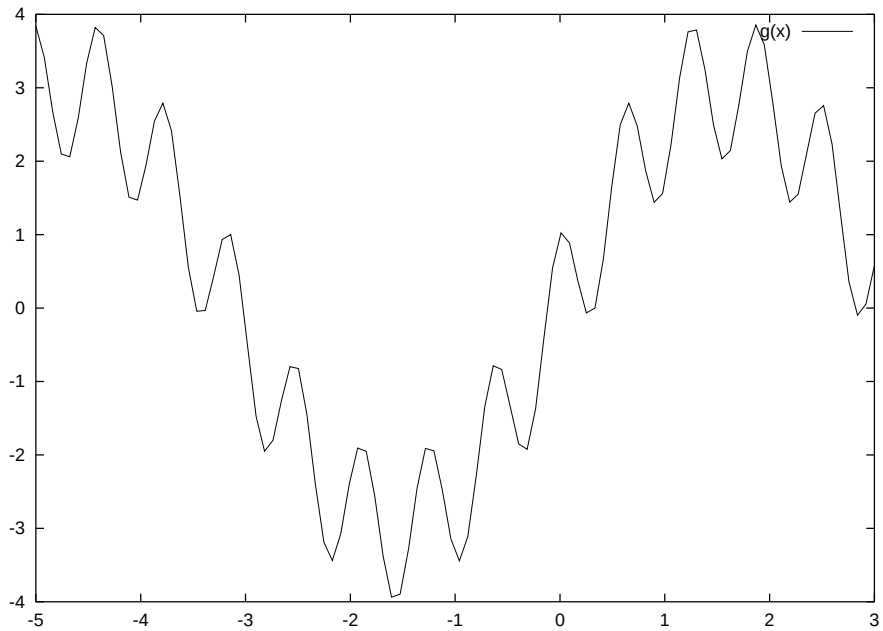
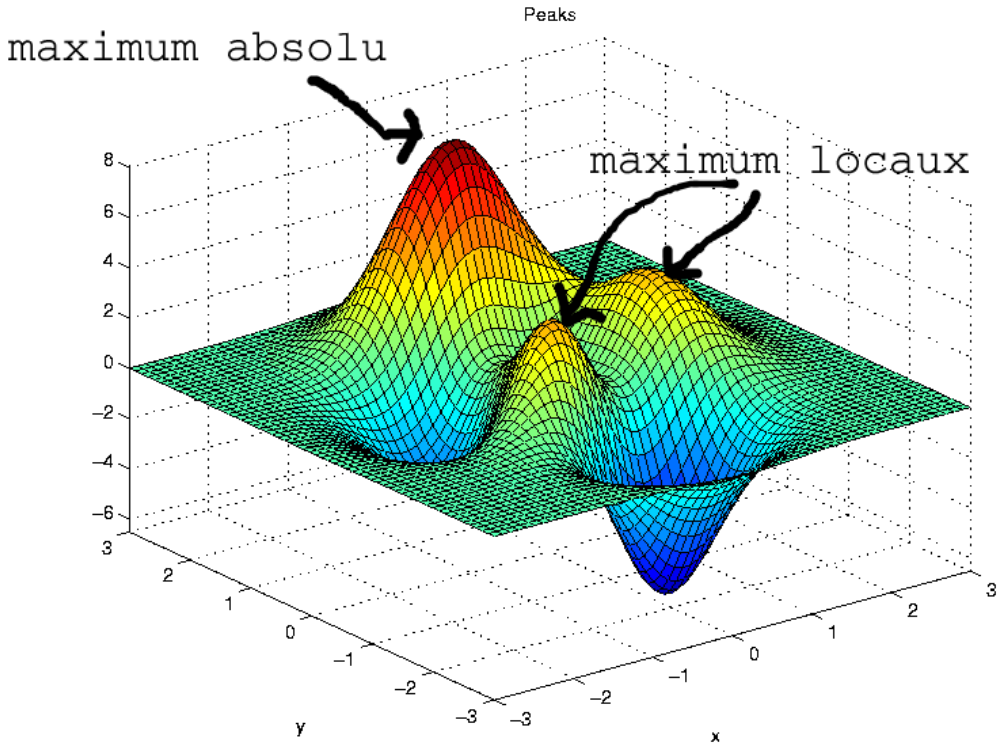


Illustration dans le cas de fonctions continues : 2D



etc

Exemples de recherches locales

le hill-climbing Aussi appelé descente en gradient suivant le sens de l'optimisation (min ou max recherché).
choisir_un_voisin : choix aléatoire dans le voisinage courant.
acceptable : seulement s'il il y a une amélioration ($d \leq 0$).
méthode opportuniste

Exemples de recherches locales

le steepest hill-climbing : prendre "la plus grande pente"

choisir_un_voisin : après avoir déterminé l'ensemble des meilleures solutions voisines de la solution courante (exceptée celle-ci), on en choisit une aléatoirement.

acceptable : si amélioration (donc arrêt sur optimum local)

algorithme glouton (*greedy*)

Exemples de recherches locales

minimisation de conflits choisir_un_voisin :

1. déterminer l'ensemble des variables en conflit (eg. liées dans un ensemble de contraintes).
2. choisir une de ces variables au hasard.
3. déterminer l'ensemble de ses meilleures valeurs (recherche complète ou pas ?).
4. en choisir une au hasard, affecter la variable.
5. retourner la solution.

acceptable : toujours.
il ne peut y avoir de dégradation de solutions ($\rightarrow$ arrêt sur optimum local).

Les classiques en question(s)

Comportement général :

1. une majorité de mouvements améliorent la solution courante.
2. le nombre d'améliorations devient de plus en plus faible.
3. il n'y a plus d'améliorations : on est dans un optimum, qui peut être local.

Les questions à se poser (1)

- quand faut-il s'arrêter ?
- faut-il être opportuniste ou gourmand ?
- comment ajuster les paramètres nombre d'essais/nombre de mouvements ?
- comment comparer les performances de deux méthodes différentes ? (qualité de la solution vs. temps consommé)

Illustration (1)

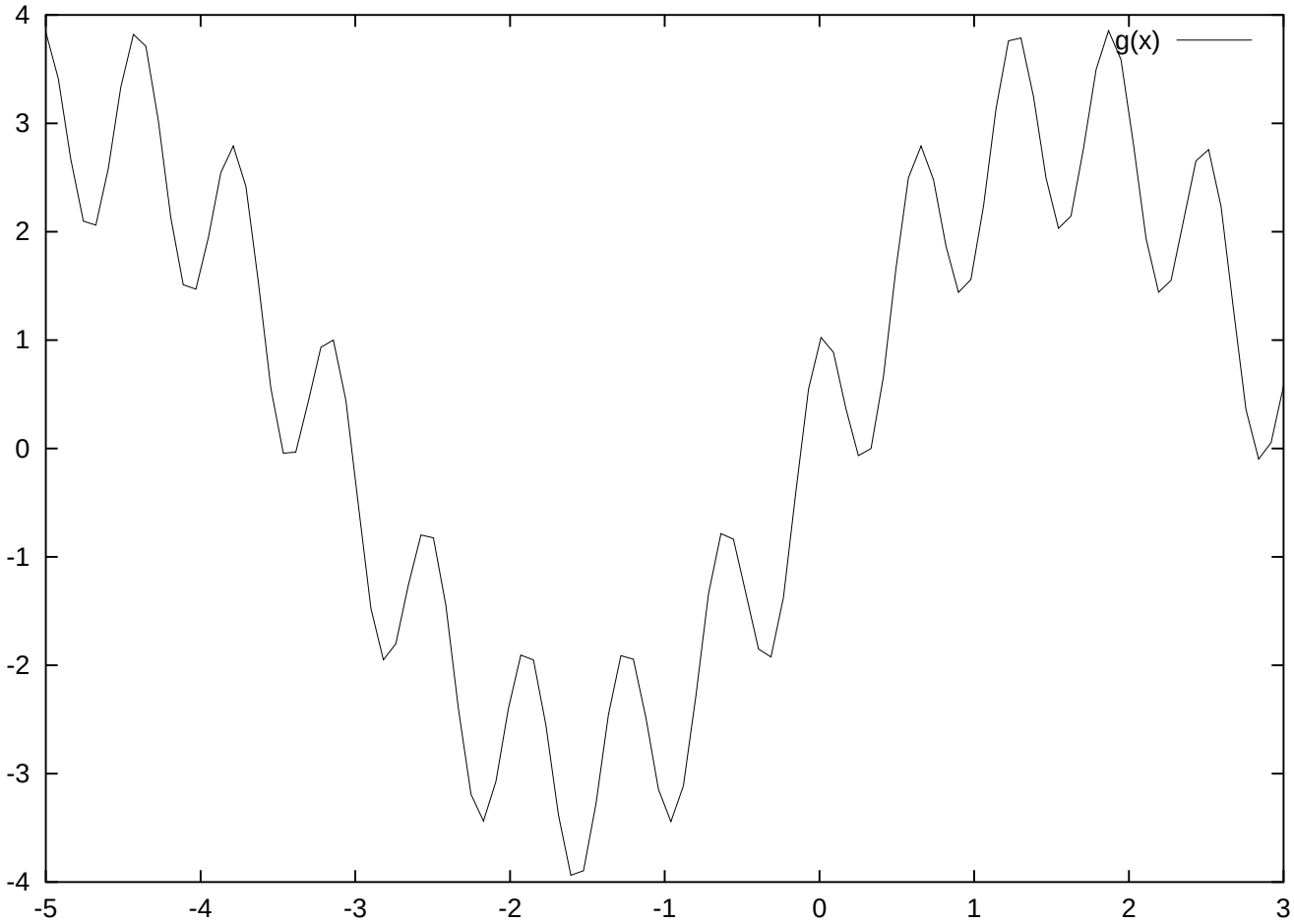


Illustration (2)

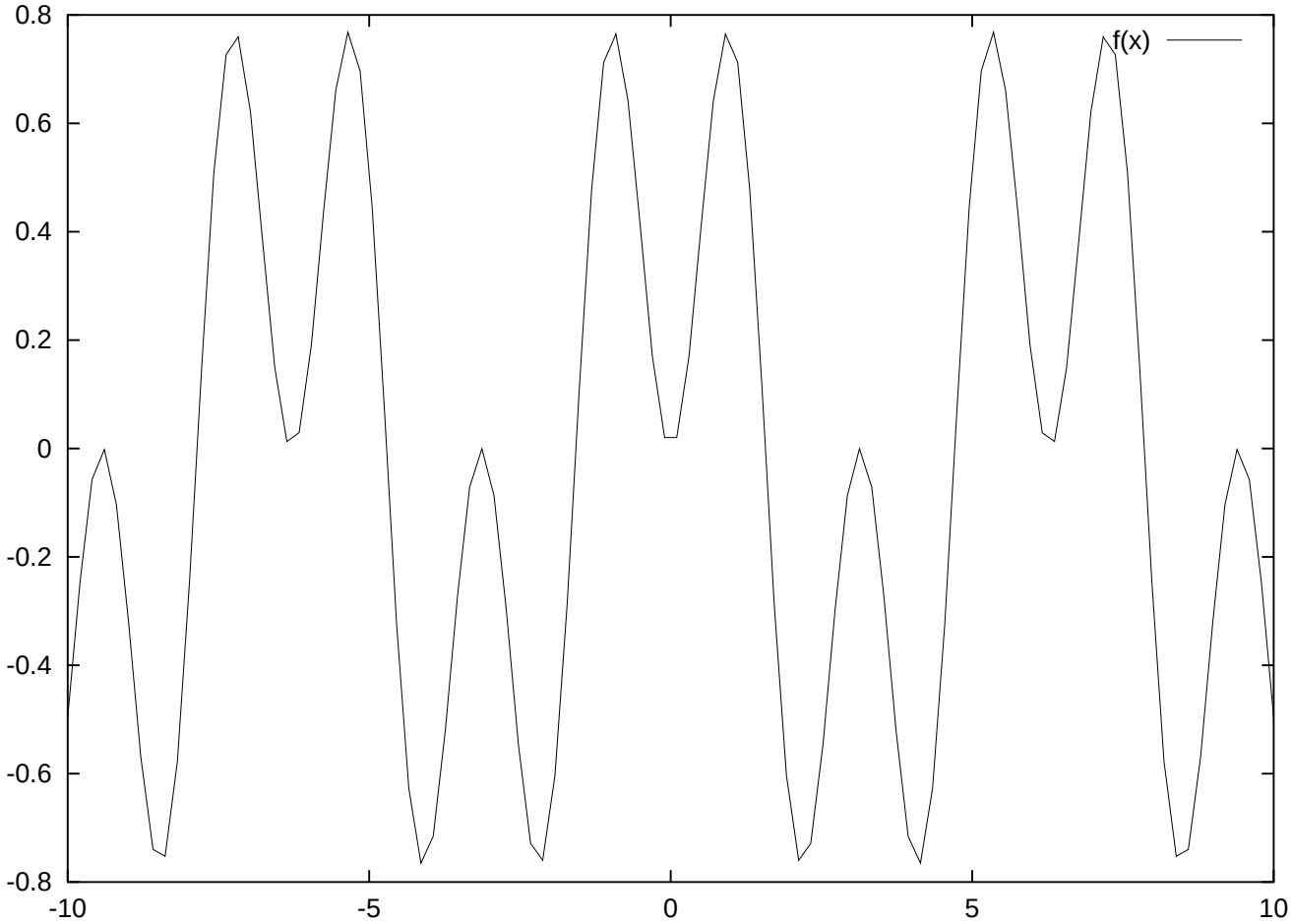


Illustration (3)

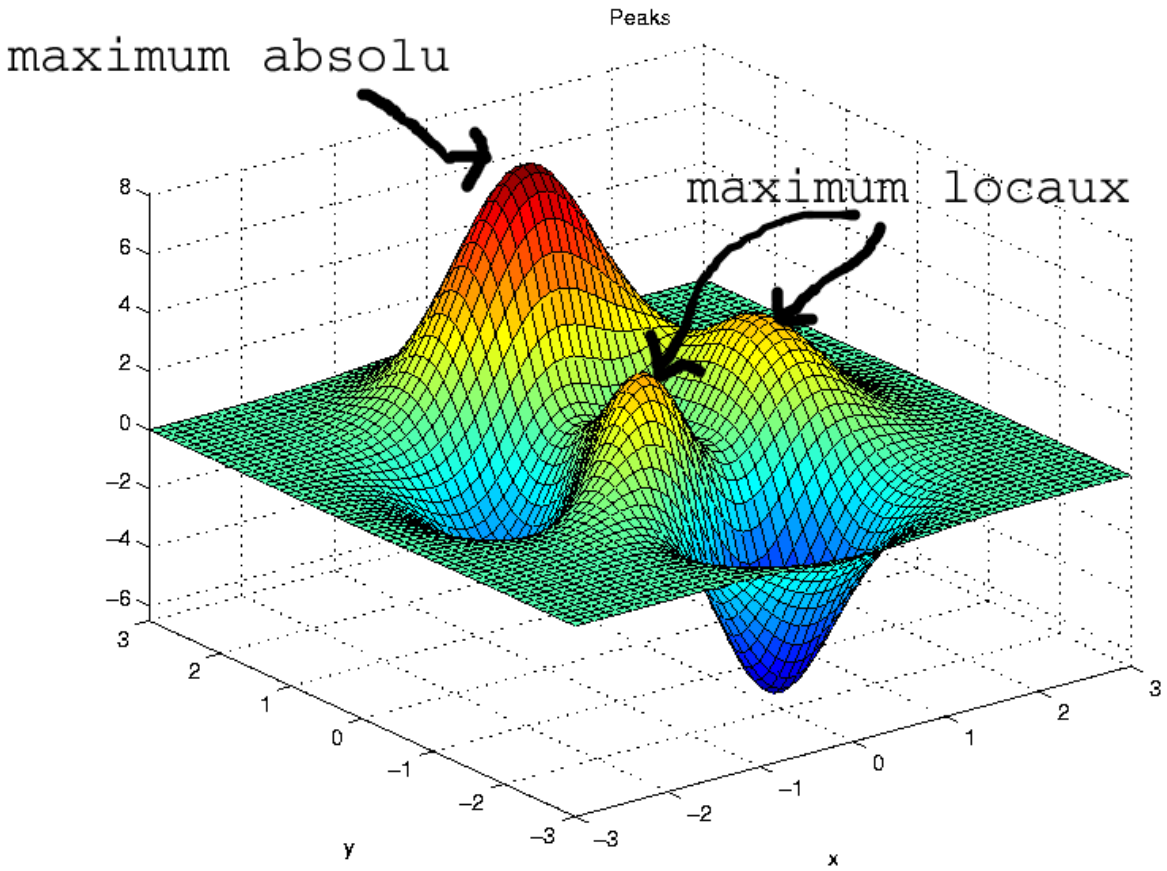


Illustration (4)

en montant le plus tôt possible (hill-climbing simple)

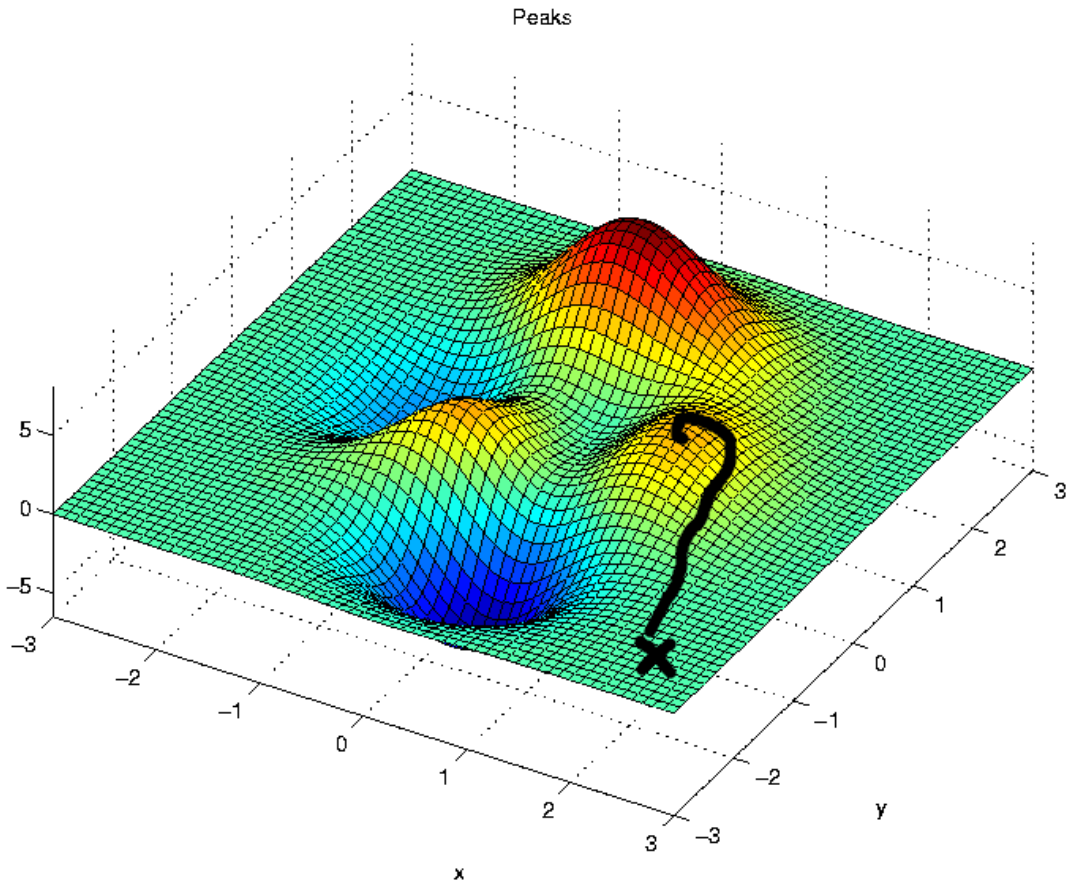


Illustration (5)

en suivant la pente la plus forte

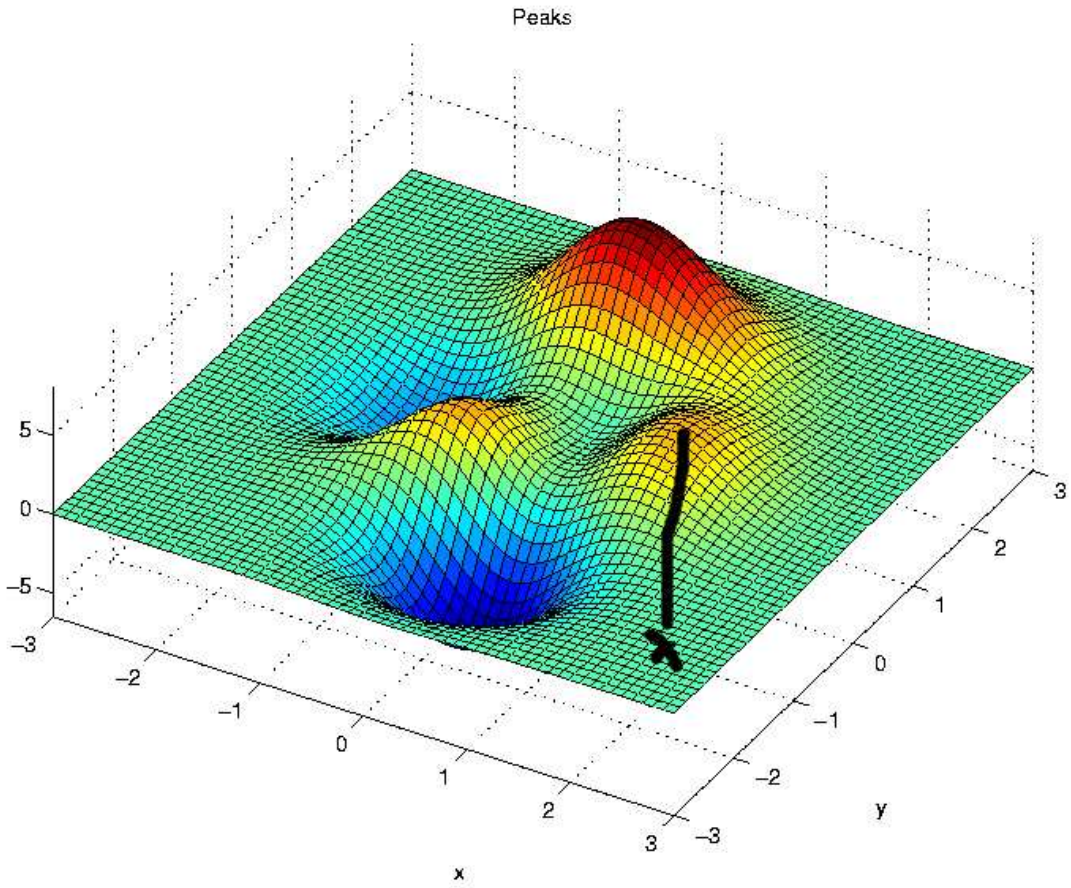
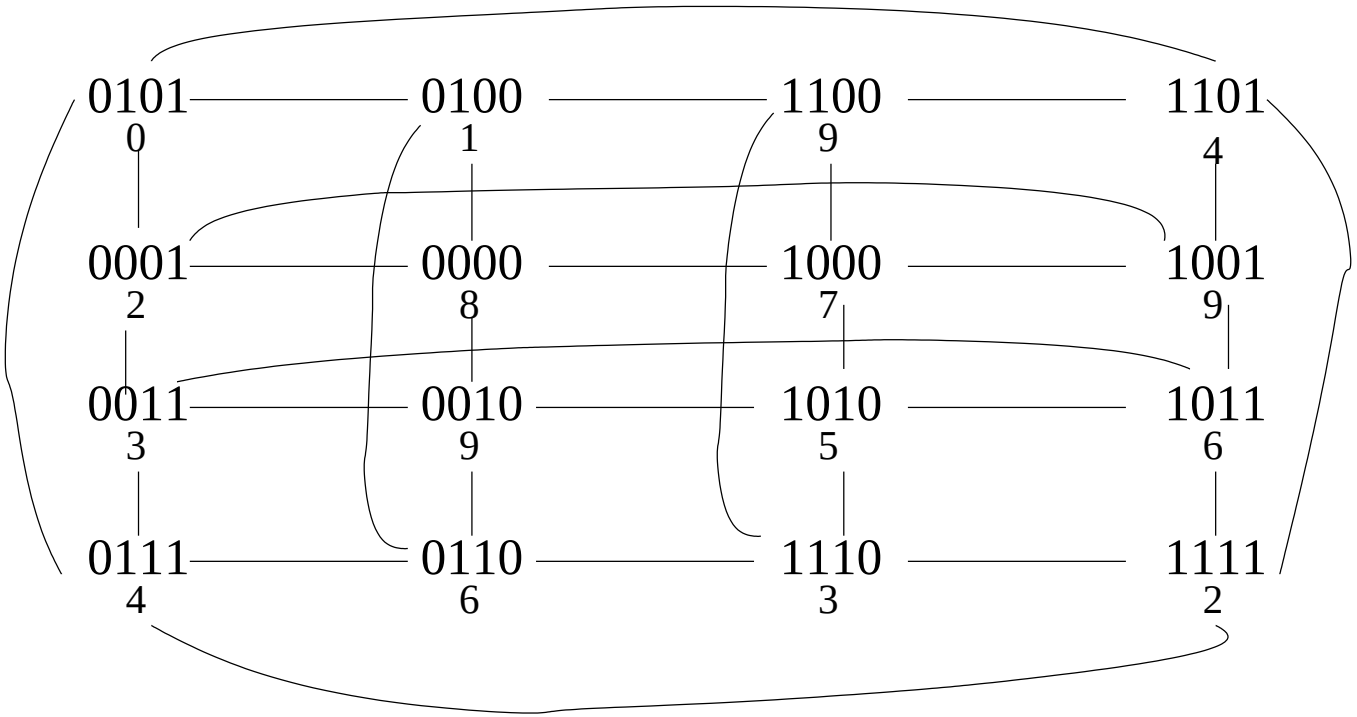


Illustration dans le cas discret

Exercice : Faire un essai en partant de 1000 sur l'espace de recherche suivant :



Les questions à se poser (2)

Pour le critère d'arrêt :

- pour l'essai en cours : nb de mouvements max ?
  comment détecter un optimum local ?
- pour la recherche elle-même : le nb max d'essai ?
  coût de la solution convenable ?
  limite de temps atteinte.

Avidité ou opportuniste :

- si aucune recherche d'amélioration : comment trouver les bonnes solutions ?
- si recherche d'amélioration, comment éviter optimum local ?

Exercice : adaptation à quelques problèmes NP-complets

- qu'est-ce qu'une solution ? combien y'en a-t-il ?
- quelle initialisation ?
- quel voisinage ? quel est sa taille ? échange d'arcs)
- complexité selon le choix opportuniste/gourmand

Exercice : adaptation à quelques problèmes NP-complets

Problèmes à voir :

- sac à dos (knapsack problem)
- déménagement (binpacking problem)
  avec éventuellement une fonction objectif différente, eg maximiser
  $f = \sum_{i=1}^{i=N} \frac{F_i^k}{N^\alpha}$
  où N est le nombre de boîtes, et F_i la somme des volumes dans la boîte i
  (ex, k=2, $\alpha = 1$)
- couverture d'ensemble
- couverture des sommets d'un graphe
- SAT
- voyageur de commerce
- integer logic programming

Amélioration des classiques : l'approche probabiliste

Pour éviter d'être coincé dans un optimum local, on peut ajouter du **bruit** : une part de mouvements aléatoires pendant une recherche classique.
Principe :

- avec une proba p, faire un mouvement aléatoire.
- avec une proba $1 - p$, suivre la méthode originale.

On peut jouer aussi au niveau de la fonction d'acceptabilité : si il y a une dégradation, l'accepter avec une probabilité p.
Reste le problème : comment régler p ? compromis entre exploration globale/locale

Amélioration des classiques : le Tabou

basée sur une recherche par gradient

- choix dans le voisinage
- possibilité de détérioration de la solution courante
- Pour améliorer la recherche, on mémorise les k dernières solutions courantes et on interdit le retour sur une de ces solutions.
- on autorisera en plus toujours un mouvement conduisant à une meilleure solution que la meilleure obtenue auparavant.

En pratique, au lieu de mémoriser les k dernières solutions courantes (parfois trop coûteux en temps et mémoire), on mémorise les k derniers mouvements (plus restrictif mais peu coûteux en temps)

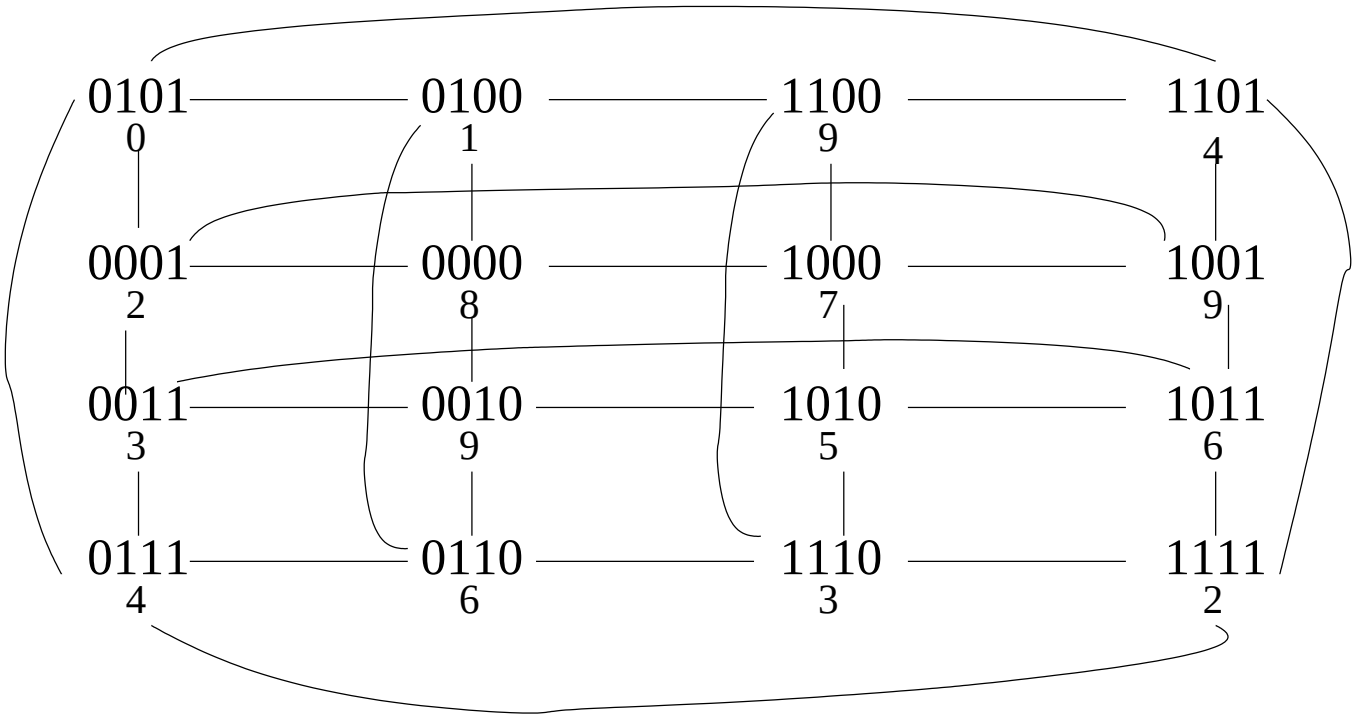
L'algorithme

```
A ← solution_initiale()
Am ← A
T ← ∅
while non(condition de fin) do
    C ← voisinage(A)-T
    Y ← min pour f sur C
    if f(Y)≥f(A) then
        T ← T ∪{A}
    end if
    if f(Y)<f(Am) then
        Am ← Y
    end if
    A ← Y
end while
retourner Am, f(Am)
```

La condition de fin est soit une limite en temps ou en itération, soit une condition de non amélioration de la solution.

Exercice

Développer l'algorithme avec $|T] = 1$, puis $|T| = 3$, puis en conservant les mouvements au lieu des états, en partant de 1111



avec taille 1 bouclage : 2-3-5-6-2-3-5-6

cycle 1111 1110 1010 1011
(sol courante (1111 2)) (tabou liste ())
(c ((1110 3) (0111 4) (1011 6) (1101 4))) (c' ((1110 3) (0111 4) (1011 6) (1101 4)))
(sol courante (1110 3)) (tabu liste ((1111 2)))
(c ((1111 2) (0110 6) (1010 5) (1100 9))) (c' ((0110 6) (1010 5) (1100 9)))
(sol courante (1010 5)) (tabu liste ((1110 3)))
(c ((1110 3) (1011 6) (0010 9) (1000 7))) (c' ((1011 6) (0010 9) (1000 7)))
(sol courante (1011 6)) (tabu liste ((1010 5)))
(c ((1111 2) (1010 5) (0011 3) (1001 9))) (c' ((1111 2) (0011 3) (1001 9)))
(sol courante (1111 2)) (tabu liste ((1010 5)))
(c ((1110 3) (0111 4) (1011 6) (1101 4))) (c' ((1110 3) (0111 4) (1011 6) (1101 4)))
(sol courante (1110 3)) (tabu liste ((1111 2)))
(c ((1111 2) (0110 6) (1010 5) (1100 9))) (c' ((0110 6) (1010 5) (1100 9)))
(sol courante (1010 5)) (tabu liste ((1110 3)))
(c ((1110 3) (1011 6) (0010 9) (1000 7))) (c' ((1011 6) (0010 9) (1000 7)))

(sol courante (1011 6)))

Correction

(sol courante (1111 2)) (meilleur (1111 2)) (tabu liste ()) (c ((1110 3) (0111 4) (1011 6) (1101 4))) (c' ((1110 3) (0111 4) (1011 6) (1101 4)))
(sol courante (1110 3)) (meilleur (1111 2)) (tabu liste ((1111 2))) (c ((1111 2) (0110 6) (1010 5) (1100 9))) (c' ((0110 6) (1010 5) (1100 9)))
(sol courante (1010 5)) (meilleur (1111 2)) (tabu liste ((1111 2) (1110 3))) (c ((1110 3) (1011 6) (0010 9) (1000 7))) (c' ((1011 6) (0010 9) (1000 7)))
(sol courante (1011 6)) (meilleur (1111 2)) (tabu liste ((1111 2) (1110 3) (1010 5))) (c ((1111 2) (1010 5) (0011 3) (1001 9))) (c' ((0011 3) (1001 9)))
(sol courante (0011 3)) (meilleur (1111 2)) (tabu liste ((1111 2) (1110 3) (1010 5))) (c ((0111 4) (1011 6) (0010 9) (0001 2))) (c' ((0111 4) (1011 6) (0010 9) (0001 2)))
(sol courante (0001 2)) (meilleur (1111 2)) (tabu liste ((1111 2) (1110 3) (1010 5))) (c ((0011 3) (1001 9) (0000 8) (0101 0))) (c' ((0011 3) (1001 9) (0000 8) (0101 0)))
(sol courante (0101 0)) (meilleur (0101 0))

Amélioration des classiques : le recuit simulé

méthode inspirée par une analogie avec un phénomène de physique statistique, l'obtention d'un état stable d'un métal.
En termes de recherches locales, la probabilité d'acceptation d'un mouvement de A à A' :

$$p(\phi(A') - \phi(A)) = 1 \text{ si } \phi(\mathrm{A}') \leq \phi(\mathrm{A})$$

$$p(\phi(A') - \phi(A)) = e^{\frac{-(\phi(A') - \phi(A))}{T}} \text{ si } \phi(\mathrm{A}') > \phi(\mathrm{A})$$

(issu de lois thermodynamiques)
Principe : on simule le processus de recuit des métaux :

- on part d'une température T élevée (instabilité/ loin de l'optimum).
- on refroidit lentement par plateaux (réduction du bruit).

Au départ on choisit T assez élevée pour que chaque mouvement ait une probabilité d'acceptation de plus de 50% (mouvement très chaotique)
Après chaque plateau, T est baissée (réduction du bruit jusqu'à ne plus accepter de solutions plus mauvaises que la solution courante).

```
A* := creer_une_solution()
A := creer_une_solution()
T := T_initiale
while (T > T_min) do
    for m :=1 a m :=longueur_des_plateaux do
        A' :=choisir_voisin(A)
        d := (f(A')-f(A))
        if d≤0 then
            A :=A'
        else
            if vrai avec proba p = e^{-d/T} then
                A :=A'
            end if
        end if
    end for
    if f(A*)>f(A) then
        A* :=A
    end if
    T :=T*α
```

Réglage des paramètres

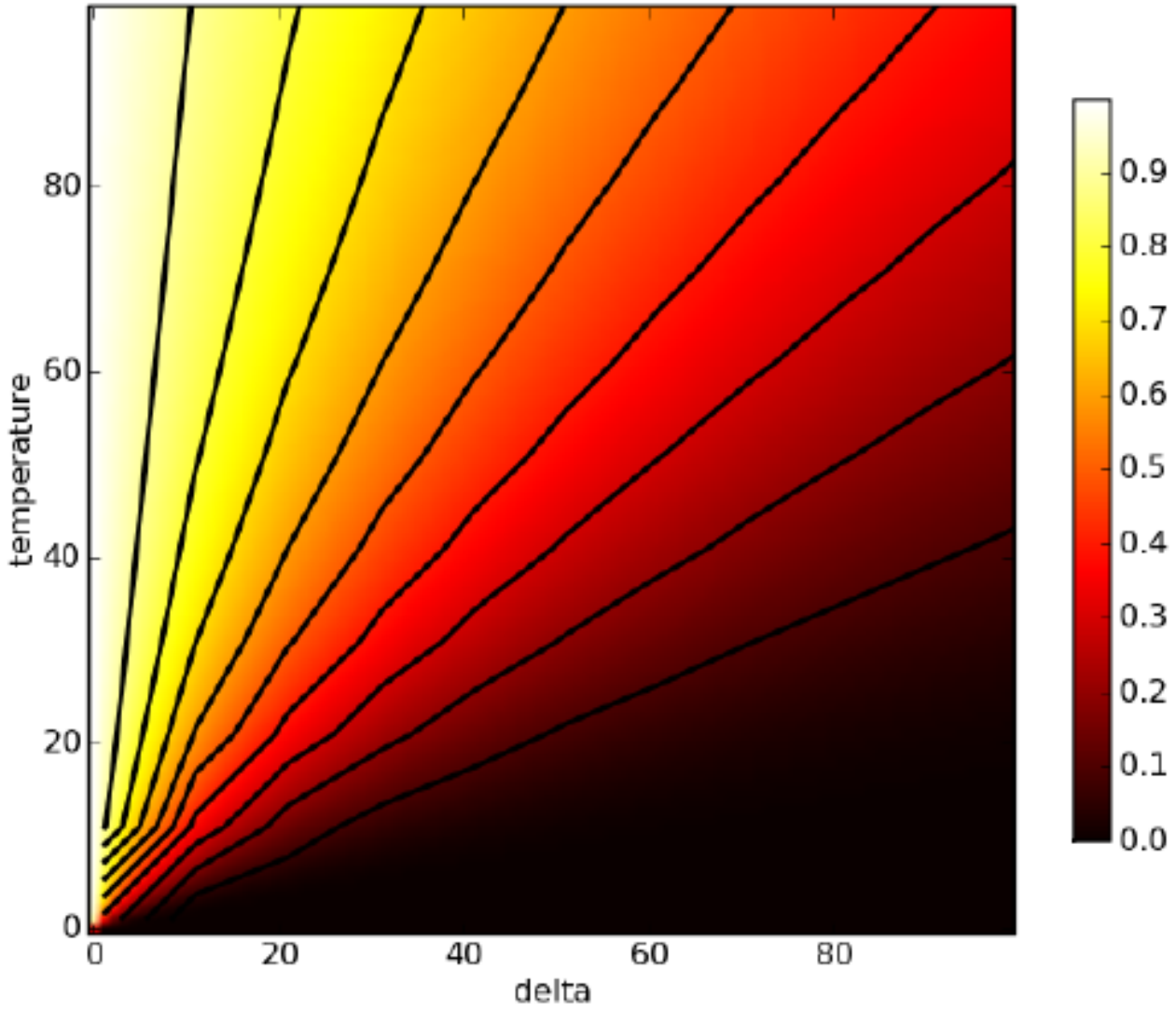
- valeur de T_initiale : si la valeur T_init permet d'accepter x% des configurations de départ, alors on la garde sinon on la double et on recommence (x au moins plus de 50, certains auteurs : 80).
- la durée des paliers (longueur du plateau) : le plus simple est de le borner par le nombre de configurations atteignables en un mouvement élémentaire. là encore subtilités possibles
- la décroissance de la température : le plus simple est facteur constant, de l'ordre de 0.9 / 0.95 (décroissance lente). pour raffiner, le réglage se joue entre ce paramètre (+/- rapide) et le précédent (plateau +/- long).
- condition d'arrêt : la température mini ; soit 0 mais pas très efficace, soit quand améliorations très petites (très empirique).

Exercice : retour sur l'espace 0-15

départ 1111
delta 3 $\rightarrow$ T=5 pour 50%
t=15 pour 80%
paliers longueur 4
décroissance T dépend (0.9)

d	T	p
-5	20	0.77
-5	2	0.08
-0.1	200	0.99

probabilité$=f(\delta, T)$



Adaptation TSP

Avantages des recherches locales

- méthodes rapides (et temps paramétrable) ;
- faciles à implémenter ;
- donnent souvent de bonnes solutions ;
- fonctionnement intuitif

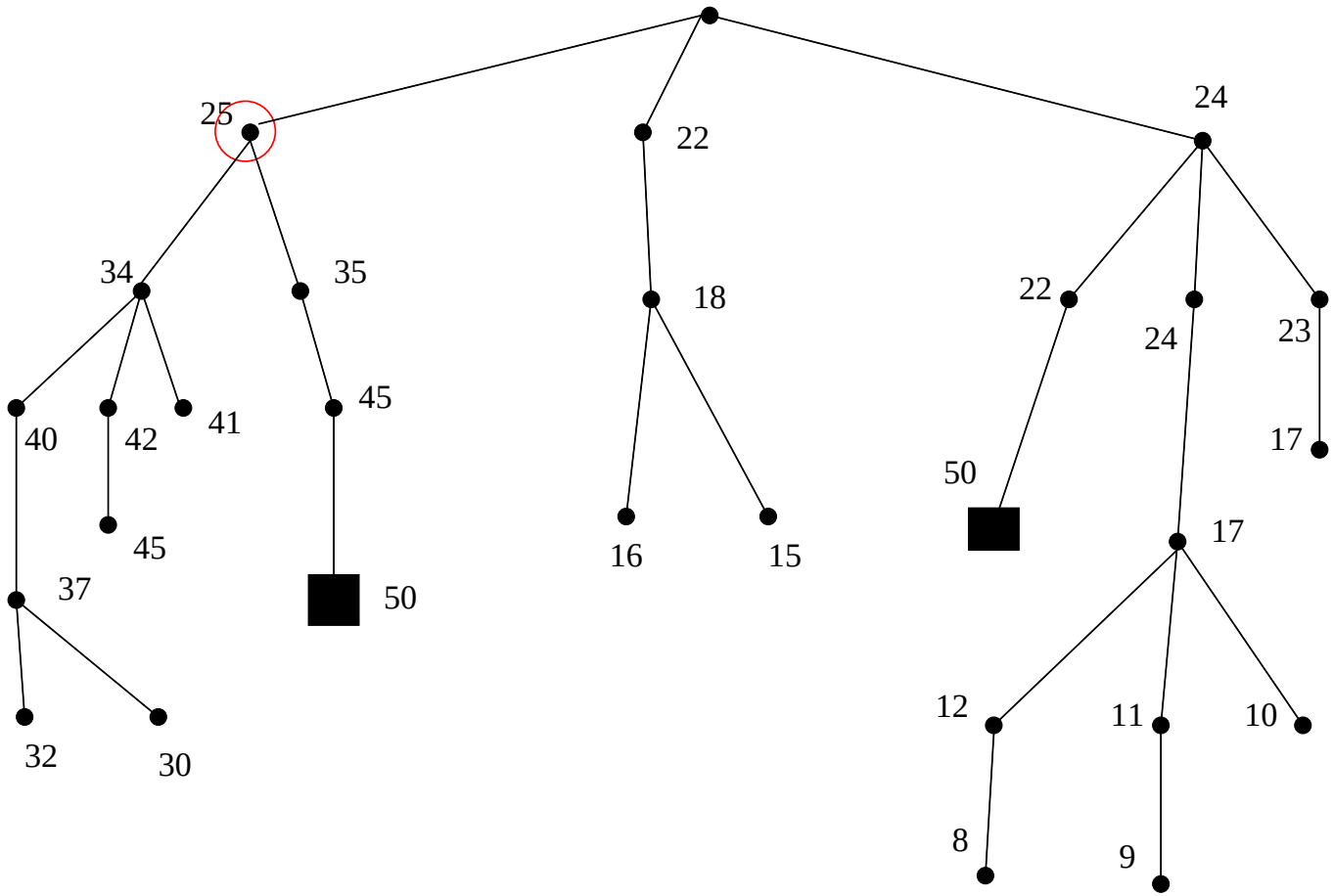
Inconvénients des recherches locales

- manque de modélisation mathématiques (chaque cas est différent : il faut adapter la recherche à chaque problème particulier)
- difficiles à paramétrer (=bidouille !)
- aucune évaluation de la distance à l'optimum (pas une approximation, trouve au pire un optimum local qui n'a rien à voir)

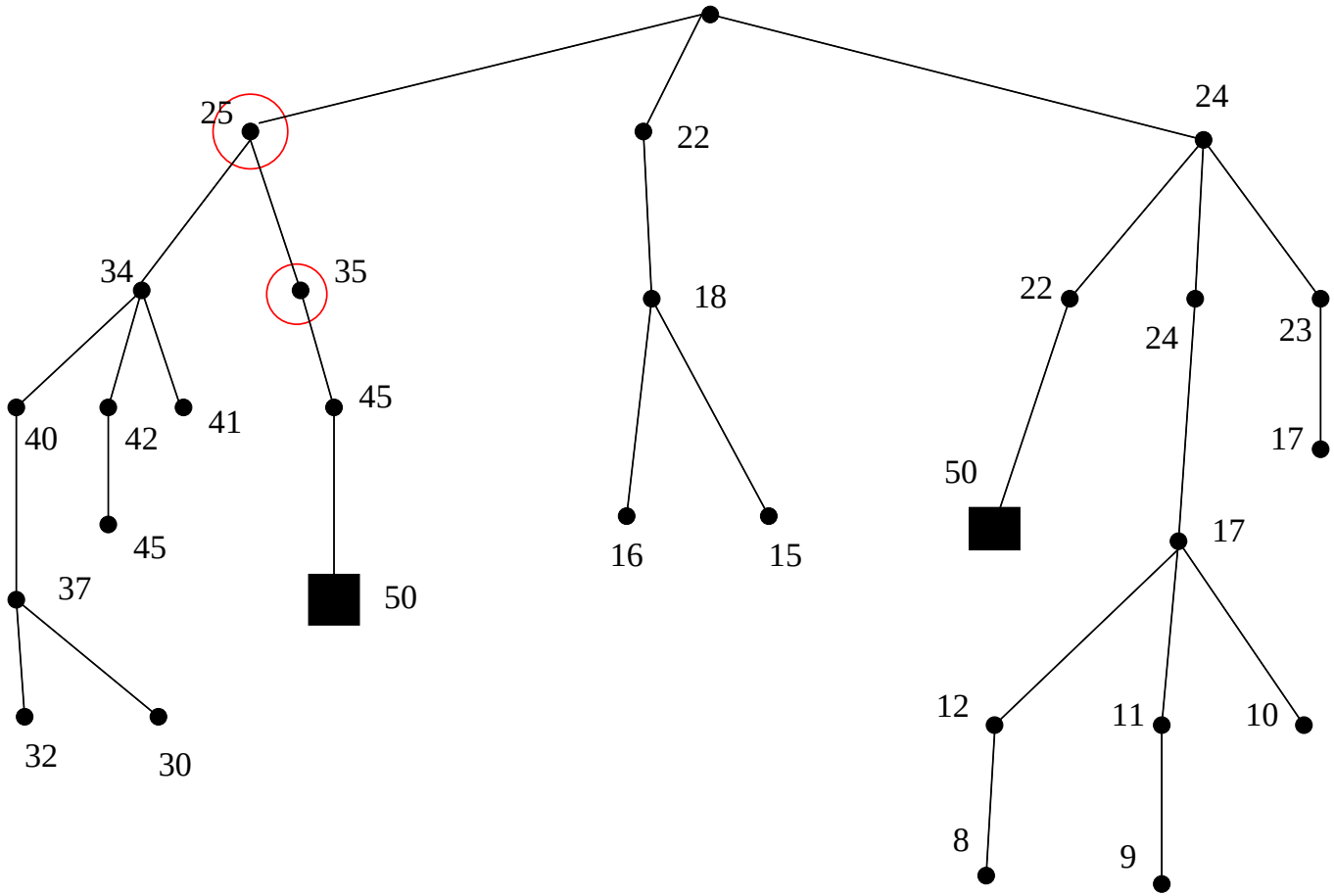
Autres méthodes incomplètes

- exploration arborescente par meilleur d'abord sans retour arrière
- généralisation aux p-meilleures solutions : beam search
- hybride local/global : les algorithmes génétiques

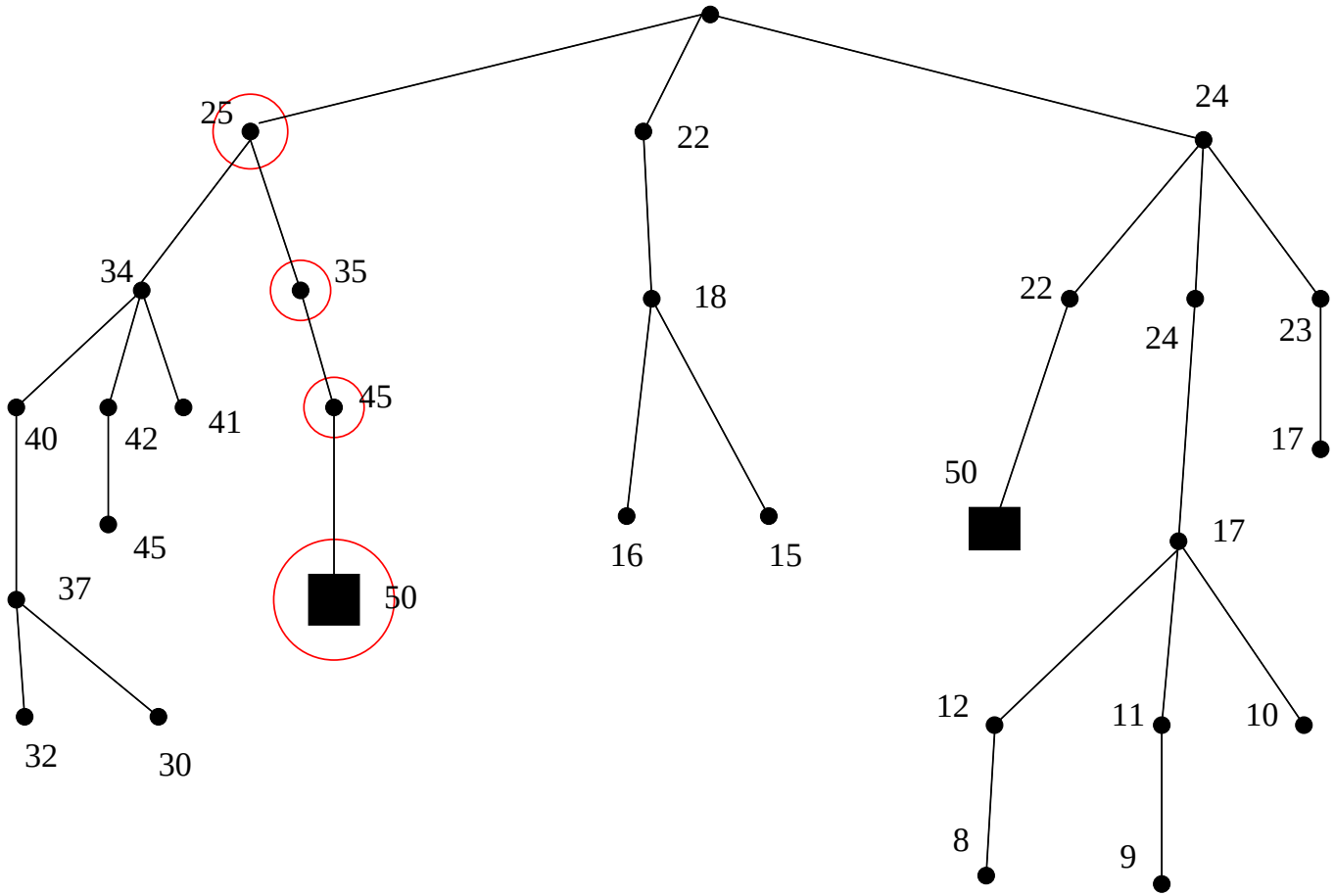
Recherche par meilleur d'abord sans retour arrière



Recherche par meilleur d'abord sans retour arrière



Recherche par meilleur d'abord sans retour arrière



Recherche en rayon "beam search"

- principe identique, mais on garde les p-meilleurs solutions rencontrées
- à chaque étape on développe le meilleur des p solutions, et on garde les p meilleurs entre ses descendants et les précédents

1. on met dans le noeud racine dans Q, la liste des noeuds en attente.
2. tant que Q n'est pas vide et que l'on a pas atteint un noeud solution, faire :
   1. tester si un élément de Q est solution, si oui, on arrête avec succès.
   2. sinon on développe tous les successeurs des éléments de Q.
   3. on les trie avec la fonction à optimiser ou une heuristique et on garde les p meilleurs.
3. si on a trouvé une solution, succès, sinon échec.

méthode simple à mettre en oeuvre, ajustable.
peut être combiné à méthode locale pour servir de point de départ

Les Algorithmes génétiques

Les algorithmes génétiques sont une forme de recherche locale qui se démarque un peu des méthodes précédentes. Le principe est inspiré d'une analogie biologique : on considère un ensemble de solutions comme une **population d'individus** susceptible d'évoluer et de se croiser, en suivant certaines règles proches de la sélection naturelle.
L'intérêt principal est de couvrir l'espace de recherche plus largement.
Trois opérations sont réalisées successivement sur la population :

- la sélection des meilleures solutions/individus en proportion/ de leur adaptattion ou avec une proba proportionnelle ;
- le croisement d'individus entre eux ;
- la mutation éventuelle d'un ou de plusieurs individus : changements aléatoires ;

L'algorithme

population initiale P avec n solutions

for g :=1 a max générations **do**
  C := vide
  for m :=1 a n par pas de 2 **do**
    parent1 := selection(P)
    parent2 := selection(P)
    enfant1,enfant2 :=croisement(parent1,parent2)
    C := C + mutation(enfant1) + mutation(enfant2)
  end for
  P := C
end for

Quelques caractéristiques des Algorithmes génétiques

– Historiquement, un individu était codé par une chaîne de bits (son "patrimoine génétique"). En pratique chaque problème appelle un codage particulier.
– La sélection se base sur l'adaptation (ou **fitness**) d'un individu, donnée par une fonction d'utilité (on cherche à maximiser l'adaptation des individus).
– La sélection permet de focaliser la recherche sur les zones intéressantes, alors que mutations et croisements permettent de diversifier la recherche (on retrouve, plus lâchement, le paradigme des recherches locales).
– Les algorithmes génétiques sont semblables à des recherches locales effectuées +/- en parallèle.
– Utilité dans le cas de problèmes dont on ignore la structure (espace de recherche mal connu par exemple

Exercice : traiter le TSP avec un AG

- encodage d'un parcours
- recombinaison : comment avoir un parcours valide ?
- mutation

Quelques inconvénients des algorithmes génétiques

Les problèmes techniques importants sont :

- celui du codage des configurations en binaire
- celui de la recombinaison de solutions pour garder quelque chose de pas trop loin de l'optimum
- performances (temps, mémoire)

Fondamentalement, l'hypothèse de base qui sous-tend ces méthodes n'est pas toujours fondée : la recombinaison des solutions isole-t-elle toujours les parties optimales pour faire une solution optimale ?

Conclusion sur les méthodes locales

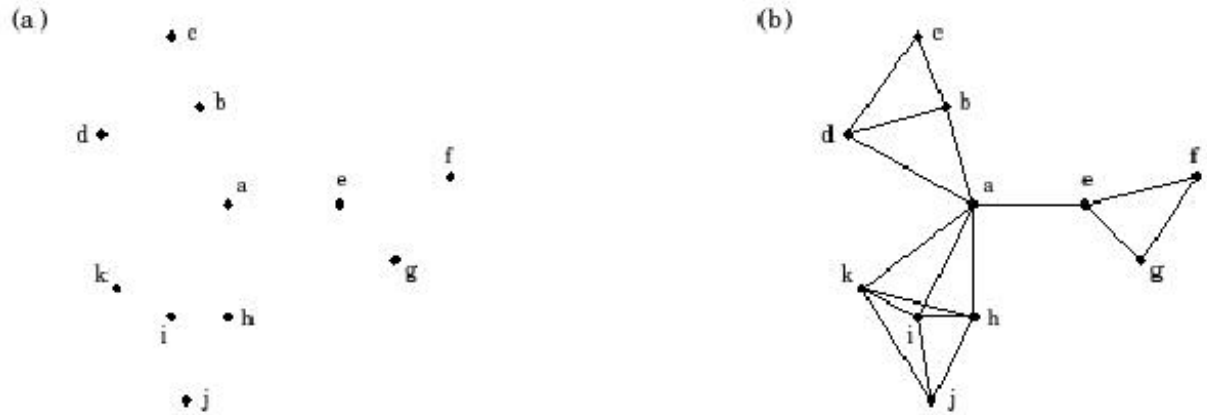
les méthodes incomplètes fournissent un ensemble de résolutions empiriques variées, parfois trop, car l'absence de modélisation les rend difficiles à adapter d'un problème à un autre (bricolage...).
Par contre elles sont à peu près indifférentes à la taille du problème, et donnent de bons résultats (approchés) à condition de bien les paramétrer.
$\rightarrow$ peuvent fournir un bon complément aux méthodes complètes.

Méthodes incomplètes et approximations

- méthodes incomplètes finissent généralement sur optimum local
- besoin de garantir la qualité de la solution : estimation de la différence à l'optimum
- selon les cas on peut encadrer la valeur
- plus généralement : jusqu'où peut-on aller dans l'approximation, et à quel prix ?

Exemple

ensembles de villes à couvrir avec une école, pour que tout le monde soit à moins de 30km d'une école.



Un algo approché

input : ensemble de villes V = v1,v2, ...v_n
pour chaque ville, on a un ensemble de villes voisines (moins de 30km), donc R= S1, S2,... S_n etc inclus dans V
sortie : un sélection de S_i tels que l'union fasse V fonction à optimiser : le nb d'ensembles choisis

U $\leftarrow \emptyset$
solution $\leftarrow \emptyset$
while U different de V **do**
    S $\leftarrow argmax_{S_i \in R/solution} \|S_i\|$
    U $\leftarrow U \cup S$
    solution += S
end while

Ecart à l'optimum

supposons $\|V\| = n$
et optimal = k
alors l'algo précédent trouve au pire k*ln n
soit nt = nombre de villes non couvertes à l'itération t. n0=n
ces éléments sont couverts par une partition de k elets donc il y a un ensemble avec au moins nt/k d'entre eux (pire cas avec répartition uniforme ds ts les ensembles)

$$n_{t+1} \leq nt - nt/k = nt(1 - 1/k)$$

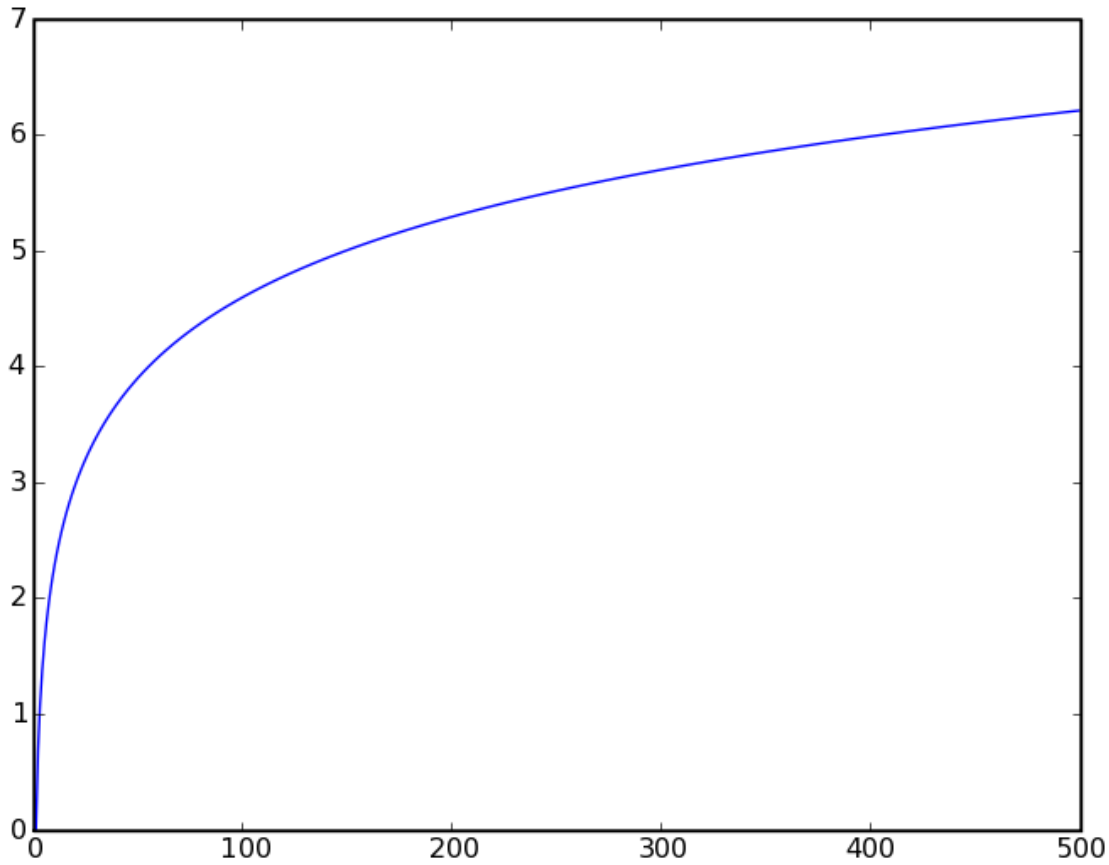
$$n_t \leq n0 * (1 - 1/k)^t$$

sachant $1 - x \leq e^{-x}$

$$nt \leq n0 * (e^{-1/k})^t = n * e^{-t/k}$$

pour $t = k * \ln n$, $n_t < n * e^{(-lnn)} = n/n = 1$ donc $n_t = 0$
et on a pris k*ln n ensemble soit ln n fois plus que l'optimum.

pas si bien que ça en a l'air :



Cas général

- une instance $\mathcal{I}$ d'un problème, de taille n
- $OPT(\mathcal{I}) =$ valeur de la solution optimale
- $A(\mathcal{I}) =$ une approximation de la solution par un algorithme incomplet A.
- le ratio d'approximation est défini par

$$\alpha_A = \max_{\mathcal{I}} \frac{A(\mathcal{I})}{OPT(\mathcal{I})}$$

- NB : le ratio est le pire cas

Exemple constant : TSP avec inégalité triangulaire

- soit TSP* l'optimum, et MST le coût de l'arbre couvrant minimal du graphe.
- on a MST$\leq$ TSP*
- en parcourant les arcs de cet arbre dans un sens puis l'autre on obtient un cycle C avec répétition de sommets, de longueur 2*MST
- on obtient un vrai cycle sans répétition de sommets à partir de C en les prenant dans l'ordre de parcours, et en sautant au suivant à chaque fois qu'on rencontre un sommet déjà rencontré.
  on remplace donc 2 arcs (i,j)+(j,k)
  par un arc (i,k), plus court par l'inégalité triangulaire.
- au total le cycle de longueur C' $\leq$ C = 2MST, et TSP* est par définition $\leq$ à ce cycle.

donc TSP* $\leq C' \leq 2 * MST \leq 2 * TSP^*$
C' a donc un ratio d'approximation borné par 2

Exemple d'approximabilité arbitraire polynomiale : sac à dos

NB : il existe un algorithme en O(n*Vt), avec n= nb objets, Vt=capacité max du sac
(mais Vt est potentiellement exponentiel par rapport à n).

D'où un algorithme approché :

- diviser toutes les valeurs par un facteur fixé à l'avance,
- arrondir à l'entier inférieur,
- résoudre exactement
- la solution au problème approché est aussi une solution au problème initial, avec une différence que l'on peut estimer.

Preuve I

- on prend comme diviseur : $\alpha = \frac{n}{\epsilon v_{max}}$
- chaque volume approximé est donc : $\hat{v}_i = \alpha v_i$
- chaque volume est maintenant $\leq n/\epsilon$ donc l'algorithme en $O(nV'_t)$ est maintenant en $O(n * n * n/\epsilon) = O(n^3/\epsilon)$ et est donc polynômial
- on considère pour simplifier la démonstration que le gain d'un objet est aussi son volume.
- soit $\hat{S}$ la solution optimale au problème approché et S la solution au problème exact.

$$\sum_{i \in \hat{S}} v_i \geq (\sum_{i \in \hat{S}} \hat{v}_i) * \frac{\epsilon v_{max}}{n} \quad (1)$$

car on a arrondi à l'entier inférieur.

Preuve II

la solution approchée est meilleur que la solution exacte, sur le problème approché :

$$\sum_{i \in \hat{S}} \hat{v}_i \geq \sum_{i \in S} \hat{v}_i$$

or

$$\sum_{i \in S} \hat{v}_i = \sum_{i \in S} v_i \frac{n}{\epsilon v_{max}} \geq \sum_{i \in S} (v_i \frac{n}{\epsilon v_{max}} - 1) = K^* \frac{n}{\epsilon v_{max}} - \|S\| \geq K^* \frac{n}{\epsilon v_{max}} - n$$

d'où avec (1) :

$$\sum_{i \in \hat{S}} v_i \geq (K^* \frac{n}{\epsilon v_{max}} - n) \frac{\epsilon v_{max}}{n} = K^* - \epsilon v_{max} \geq K^* - K^* \epsilon = K^*(1 - \epsilon)$$

donc la solution au problème approché rapporte au pire $K^*(1 - \epsilon)$ avec les volumes exacts, soit un ratio d'approximation de $(1 - \epsilon)$ qui est donc aussi proche de 1 que l'on veut.
on parle alors d'approximabilité arbitraire

Bonus track : solution exacte du Knapsack

p contient le sac à dos.
algorithme en O(n*Vt), avec n= nb objets, Vt=capacité max du sac
(mais Vt est potentiellement exponentiel par rapport à n).
table contient les valeurs optimales de remplissage pour des paires (p,q) ou p est le volume à remplir, et q indique que l'on considère seulement les q premiers objets de l'ensemble des objets. pour chaque objet j on doit décider si on le met ou non. donc

```
table[w,j]=max(table[w,j-1],(table[w-wj,j-1]+cj))
```

Bonus track : solution exacte du Knapsack

code (à peine simplifié) :

```
table={}
for j in range(p.nb_obj()):
    table[0,j+1]=0

for j in range(1,p.nb_obj()+1):
    for w in range(1,p.capacity()+1):
        wj,cj=p[j]
        if wj > w:
            table[w, j]= table.get((w, j - 1),0)
        else:
            table[w,j]=max(table[w,j-1],
                           (table[w-wj,j-1]+cj))

print table[p.capacity(),p.nb_obj()]
```

Exemple de non approximabilité polynomiale

TSP dans le cas général n'a pas de borne d'approximation

La hiérarchie d'approximabilité

Pour tout problème, on cherche l'algorithme avec la meilleure approximation.
On essaie aussi de trouver la borne minimale d'approximation polynomiale.
On peut maintenant distinguer les problèmes :

- qui n'ont pas d'approximation polynômiale (TSP) ;
- ceux pour qui on peut trouver un ratio d'approximation, mais avec une limite (clusterisation, TSP avec inégalité triangulaire)
- ceux qui sont arbitrairement approximable en temps polynomial (sac à dos) (ratio tend vers 1)
- ceux qui ont une approximation intermédiaire, par exemple en (log n), comme la couverture d'ensemble.

Attention aux conclusions hâtives / pire cas / $P \stackrel{?}{=} NP$

Conclusion Générale

- méthode incomplètes
- problèmes NP