Complexité (Quatrième année)

TD9

Maud MARCHAL

21 avril 2011

Plan

Introduction

- Les algorithmes de recherche en profondeur et en largeur n'utilisent que la structure du graphe comme information sur le problème traité.
- Les algorithmes basés sur une recherche par heuristique utilisent une nouvelle information appelée **fonction heuristique** h.
- Cette fonction h associe à un nœud e un entier qui est une estimation de la "distance" entre e et la première solution du sous-arbre de recherche issu de e.
- Remarque : dans la recherche en profondeur avec retour arrière, on a déjà utilisé des heuristiques (fonctions d'estimations dans l'algorithme Min-Max).

Recherche par heuristique

- **Heuristique** : information sur le "futur", c'est à dire sur la partie non développée de l'arbre de recherche.
- Heuristique : du grec euristikê : [art] de découvrir
- Ce type d'information **peut** permettre de conclure qu'il est inutile de développer un sous-arbre car on est sûr qu'il ne contient aucune (meilleure) solution (qu'une solution déjà connue).
- La notion de recherche heuristique est une des notions de base de l'intelligence artificielle.

Recherche par heuristique

- **Objectif** : diriger la recherche dans l'arbre de façon à réduire le temps de résolution du problème. Une heuristique introduit des mécanismes qui dérivent de connaissances spécifiques au problème.
- Il est courant de mixer l'utilisation d'une heuristique avec des méthodes de recherche ne garantissant plus la complétude de la solution afin d'améliorer l'efficacité.
- Justification : on ne cherche pas toujours la meilleure solution à un problème mais une solution satisfaisante (ex : problème de l'emploi du temps).
- Inconvénient : une heuristique ne suffit pas toujours à résoudre un problème.

Comment choisir une heuristique ?

- Supposons qu'on cherche à évaluer une fonction e croissante le long de chaque branche dont on cherche une solution minimale.
- Au niveau du nœud courant N, on connaît la valeur partielle e(N) ; Le problème est de savoir s'il est intéressant de parcourir le sous-arbre de racine N.
- Si on possède une heuristique d(N) qui minore ce qui peut être ajouté (dans le sous-arbre de racine N) à e(N) pour obtenir une solution, alors :
  $epd(N) = e(N) + d(N)$
  avec epd(N) évaluation par défaut. epd(N) est un minorant de toute solution calculée dans le sous-arbre de racine N.

Comment choisir une heuristique ? (2)

- Si on cherche une solution minimale et si la meilleure solution déjà rencontrée est de valeur nbopt, alors :

  $epd(N) \geq nbopt$ indique qu'il est inutile de développer le sous-arbre.

  Remarque : pour une fonction décroissante et recherche d'une solution maximale, il faut un majorant.

Recherche par heuristique

- Deux types d'heuristique :
  - Heuristique statique : connue avant de commencer la recherche par essais sucessifs ;
  - Heuristique dynamique : évaluée au cours de la recherche.
- Prise en compte des heuristiques dans le modèle général :
  - PP(X,i,y) prend en considération les éventuelles heuristiques : PP(X,i,y) : ... et epd(N) $<$ nbopt
  - Si une heuristique est évaluée dynamiquement, modifier(y) et restituer(y) participent à cette évaluation.

Recherche par heuristique : exemple

	1	2	3	4
1	0	4	17	16
2	8	19	15	14
3	12	4	17	11
4	20	10	13	1

- Soit la matrice T :
- On veut se déplacer de (1,1) à (n,n) en n'employant que les mouvements $\rightarrow$ et $\downarrow$. Le coût d'un chemin est la somme des valeurs contenues dans les cases empruntées.

Recherche par heuristique : exemple

- Le problème revient à la recherche d'un chemin de valeur minimale du sommet x_{11} au sommet x_{nn}.

  1. Combien faut-il traverser de cases pour aller de (1,1) à (n,n) ?

  2. Combien faut-il traverser de cases pour aller d'une case (i,j) à (n,n) ?

Recherche par heuristique : exemple

Propositions d'heuristique (pour retenir la meilleure) :

- $d_1(i,j)$: $c_{ij}\times$ plus petite case non nulle du tableau.
- $d_2(i,j)$: $c_{ij}\times$ plus petite case non nulle de D_{ij}.
- $d_3(i,j)$: somme des c_{ij} plus petites valeurs non nulles de D_{ij}.
- $d_4(i,j)$: $\sum_{k=i+j+1}^{2n-1} min_{D_{ij}}(diagonaleDroite(k))$

Exemples pour les heuristiques :

- $d_1(2,2) =$
- $d_2(2,2) =$
- $d_3(2,2) =$
- $d_4(2,2) =$

Recherche par heuristique : exemple

- Soit N un nœud de l'arbre de recherche caractérisé par la case (i,j) atteinte et le coût $e(N)$ du chemin parcouru de (0,0) à (i,j).
- Pour tout heuristique I, on a :

$$\begin{aligned} epd_I(N) &= e(N) + d_I(i,j) \\ &\leq e(F)\ \forall F \text{ feuille du sous-arbre de racine N} \end{aligned}$$

- On a : $\forall N$ nœud de l'arbre :
  $epd_1(N) \leq epd_2(N) \leq epd_3(N) \leq epd_4(N)$
- On a intérêt à choisir l'heuristique qui donne le plus grand des minorants d_4.

Recherche par heuristique : exemple

- Remarque :

  On constate que $\forall k\ d_k(N)$ ne dépend pas vraiment du nœud N mais seulement de la case (i,j) qui est attachée à N ; on peut donc calculer d_k :
  - statiquement : tableau de n^2 valeurs dont certaines ne seront pas utilisées par l'algorithme :
  - dynamiquement : pas de perte de place, on ne calcule que les valeurs utiles ... mais on peut être appelé à calculer plusieurs fois la même valeur.
  - un compromis : calcul dynamique avec tableau : on stocke les valeurs calculées dans le tableau : évite un recalcul.

Recherche par heuristique : exemple

Arbre de recherche avec l'heuristique d_4 :

Recherche par heuristique : exemple

Algorithme :

Recherche par heuristique : Complexité

Complexité des calculs des heuristiques :

Heuristique ou Métaheuristique ?

Un algorithme d'approximation peut être :

- basé sur un critère propre au problème : on parle **d'heuristique** : elles sont souvent basées sur des règles empiriques tirées de l'expérience (ex : heuristiques gloutonnes) ;
- l'instanciation d'une **métaheuristique** : algorithmes génétiques, recuit simulé, ... avec une notion de voisinage d'opérateur adaptée au problème, souvent basée sur des analogies avec des phénomènes physiques ou naturels.

Plan

Principe des heuristiques gloutonnes

- Le fonctionnement d'une heuristique gloutonne est similaire à celui d'un algorithme glouton exact (cf. précédent TD) ;
- La différence réside dans le fait qu'on n'impose pas que la solution obtenue soit optimale (plus besoin de prouver l'optimalité !) ;
- On obtient donc un **algorithme d'approximation**.

Premier exemple : problème du sac à dos

- Le problème consiste à remplir au mieux un sac à dos de capacité C avec n produits $P_1 \cdots P_n$ qui prennent un volume $v_1 \cdots v_n$ et rapportent $a_1 \cdots a_n$ par unité.
- On aimerait le bénéfice maximum.
- On suppose que l'on a autant d'unités que l'on veut pour chaque produit.
- **Formalisation du problème :**
  Ecriture d'un programme linéaire en nombres entiers.
  Soit x_i le nombre d'unités du produit P_i que l'on choisit de prendre ($x_i \in \mathbb{N}$). Il s'agit de calculer $max \sum_{i=1}^{n} a_i x_i$ sous la contrainte $\sum_{i=1}^{n} v_i x_i \leq C$.

Problème du sac à dos : application numérique

On prend a=(4,5,6,2) et v=(33,49,60,32) avec C=130.
Le problème à résoudre :
$max(4x_1 + 5x_2 + 6x_3 + 2x_4)$
et $33x_1 + 49x_2 + 60x_3 + 32x_4 \leq 130$

Arbre de recherche avec une solution globale (énumération de toutes les solutions) :

Problème du sac à dos : quel critère glouton choisir ?

Il existe plusieurs critères gloutons possibles :

- prendre le plus volumineux d'abord ?
- prendre celui qui a le plus de valeur ?
- prendre celui qui a le meilleur rapport valeur/volume ?

Aucun ne donnera la solution optimale dans tous les cas...

Problème du sac à dos : exemple d'heuristique

- On choisit de trier les objets par a_i/v_i décroissants :
  Dans notre exemple :

$$\frac{4}{33} > \frac{5}{49} > \frac{6}{60} > \frac{2}{32}$$

- Schéma glouton :

```
trier les objets par a/v decroissant
J=EnsVide; V=0; A=0;
pour chaque objet i faire :
    si il reste de place // V+vi <= C
      prendre cet objet // J=J+i; A=A+ai; V=V+vi
```

- Avec notre exemple, nous obtenons :

Algorithmes d'approximation et garanties

- Un **algorithme d'approximation** est un algorithme qui donne toujours une solution mais pas forcément optimale.
- Dans le cas des algorithmes déterministes, on peut vouloir **mesurer la qualité** de l'algorithme.
- Pour estimer la qualité de l'approximation, l'idée est de comparer la solution proposée par l'algorithme et la solution optimale.

Garantie d'un algorithme d'approximation

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.

- **Garantie absolue :** c'est la borne supérieure de $|f(A(I)) - f(Opt(I))|$ pour toutes les instances I du problème.
  - La borne n'est pas forcément finie et la notion est rarement utilisée.
- **Ratio de garantie :** c'est la borne supérieure de $\frac{f(A(I))}{f(Opt(I))}$ pour toute instance I si la fonction objectif est à minimiser, le ratio inverse si la fonction objectif doit être maximisée.
  - Plus ce ratio est proche de 1, meilleure est la garantie de l'approximation.
  - Attention : le ratio ne mesure que le comportement dans le pire des cas (qui peut être différent du comportement réel de l'approximation).

Exemple du sac à dos

- **Question** : l'heuristique utilisée précédemment a-t-elle un ratio ?
- Soit l'instance I_n : $v_1 = 1$, $v_2 = n$, $a_1 = 2$, $a_2 = n$, $C = n$.

Deuxième exemple : sac à dos binaire

- **Problème du sac à dos binaire (mise en sachets)** :
  On a n objets, de volumes respectifs $v_1, \cdots v_n$ et la capacité d'un sac est C.
  On souhaite mettre en sachets en utilisant le moins de sacs, i.e. trouver une application $g : [1 \cdots n] \rightarrow [1 \cdots n]$ telle que $\sum_{g(i)=s} v_i <= C$ pour tout sac s.
  On recherche donc le nombre de sacs minimum : $max(g(i))$.

Sac à dos binaire : application numérique

- **Application du sac à dos binaire** :
  On a $n = 5$ objets, de volumes respectifs $v_1 = 2, v_2 = 3, v_3 = 3, v_4 = 2, v_5 = 2$ et la capacité d'un sac est $C = 6$.
- La solution optimale est
- Le problème est NP-dur !

Sac à dos binaire : heuristique de la ménagère

```
n=1; // premier sachant
cap=C;
pour i de 1 a n faire :
  si vi>cap  // on ouvre un nouveau sac
       n++;
       cap=C;
  g(i)=n;
  cap=cap-vi;
```

L'algorithme est en $O(n)$.

Sac à dos binaire : garantie

- Soit k le nombre de sacs utilisés par l'heuristique sur une instance.
  Le poids des objets est supérieur à $C * k/2$.
  La solution optimale utilise $k/2 + 1$ sacs, le ratio de garantie est donc : $\frac{f(A(I))}{f(Opt(I))} \leq 2$ pour toute instance I.
- Montrons que le ratio est égal à 2 :
  Soient $p_1 = k, p_2 = 1, p_3 = k, p_4 = 1, p_5 = k, \cdots p_{4k} = 1$ et $C = 2k$.
  L'algorithme glouton va utiliser $2k$ sacs (k+1 ; k+1, etc.) alors que la solution optimale va utiliser $k + 1$ sacs (k+k ; k+k ; ... ;1 ...1).
  Le ratio est donc supérieur à $\frac{2k}{k+1}$ pour tout k. Comme le ratio est au plus 2, le ratio est donc 2.

Existe-t-il toujours un algorithme polynômial avec garantie pour problème NP-dur ?

Question : Peut-on toujours trouver un algorithme polynômial déterministe d'approximation avec une garantie ?
Non, il y a plusieurs situations possibles :

- **Sans garantie** : pour certains problèmes, on montrer que quelque soit le ratio, on ne peut pas trouver d'algorithme polynômial qui offre ce ratio de garantie (sauf si P=NP). Exemple : voyageur de commerce si on ne suppose pas que la distance vérifie l'inégalité triangulaire.
- **Avec une certaine garantie** : pour certains problèmes, on connaît des algorithmes polynômiaux qui offrent une certaine garantie. Exemple : voyageur de commerce si on suppose que la distance vérifie l'inégalité triangulaire.
- **Utilisation d'un schéma d'approximation**.

Schéma d'approximation

Pour certains problèmes, on peut montrer que quelque soit le ratio de garantie, on peut trouver un algorithme polynômial qui offre cette garantie. On a alors deux types de schémas :

- dans le cas le moins favorable, le degré du polynôme dépend de la qualité de l'approximation souhaitée (Cas du sac à dos binaire).
- dans les autres cas, on a un schéma complet polynômial d'approximation : on a un algorithme polynômial qui prend en entrée une instance du problème et une qualité de solution souhaitée et qui sort une solution ayant cette qualité (Cas du sac à dos non fractionnable).

En pratique : attention car le degré du polynôme peut être grand !

Conclusion sur les heuristiques gloutonnes

- Les heuristiques gloutonnes sont souvent simples et efficaces.
- Nous avons également introduit la notion de **ratio de garantie** pour les algorithmes d'approximation :
  - Le ratio garantit une certaine qualité mais ne représente qu'une indication dans le pire des cas.
  - Certains problèmes NP-durs n'ont pas d'algorithmes polynômiaux d'approximation avec garantie.

Remarque sur les algorithmes d'approximation

Un algorithme d'approximation peut être :

- **déterministe** : pour une entrée donnée, il donnera toujours la même solution (heuristiques gloutonnes, optimum local, tabou, etc.) ;
- **stochastique ou non déterministe** : différentes exécutions sur une même instance pourront produire des solutions différentes (recuit simulé, algorithme génétique, etc.).

Plan

Recherche guidée : différentes méthodes

Il existe deux grandes classes de recherche guidée :

- les méthodes utilisant des heuristiques pour **trouver les plus courts chemins entre l'état initial et un état solution** (Branch and Bound, A*).
- les méthodes utilisant des heuristiques pour **accélerer la recherche d'un état solution** (parcours en profondeur ou en largeur avec fonctions d'estimation). On distingue :
  - Méthodes par voisinage (Recherche d'un optimum local, Méthode taboue, Recuit Simulé)
  - Méthodes à base de population (Algorithmes génétiques, Colonie de fourmis)

Introduction aux méthodes exactes

- **Principe** : La famille de méthodes "Branch and Bound/A^*" consiste à faire une recherche en largeur en gardant les chemins générés dans la file. A chaque étape, on choisit le chemin le plus prioritaire et on l'étend par les différentes alternatives présentes au niveau du dernier sommet du chemin. Les nouveaux chemins sont rajoutés à la file.
- Pour ordonner les chemins dans la file, on utilise la fonction d'estimation $f(e)$ où e est le dernier sommet du chemin. Dans les méthodes Branch and Bound (ou procédure par Séparation et Evaluation progressive) ou bien dans l'algorithme A^*, la fonction d'estimation est décomposée en deux parties.

Fonction d'estimation

- On définit : $f(e) = g(e) + h(e)$.
  - $g^*(e)$: minimum du coût des chemins de l'état initial vers e ;
  - $h^*(e)$: minimum du coût des chemins de e vers un état terminal qelconque ;
  - $f^*(e) = g^*(e) + h^*(e)$: minimum du coût des chemins passant par e et menant de l'état initial à un état terminal.
- Remarques :
  1. $f^*(e)$, $g^*(e)$ et $h^*(e)$ sont infinies par convention si les chemins n'existent pas.
  2. $f^*(i) = g^*(i)$ avec i l'état initial correspond au coût du chemin optimal solution du problème.

Fonctions d'estimation (2)

- En général, g^* et h^* ne sont pas connues. On définit alors $h(e)$ qui est une estimation de $h^*(e)$ (information heuristique).
- On adopte également une estimation de g^* : $g(e)$ ="minimum du coût des chemins de i vers e connus explicitement à un moment donné de la recherche.
  On a : $f(e) = g(e) + h(e)$.
- Pour que le chemin trouvé soit toujours optimal, on impose à h^* de ne jamais surestimer le coût du trajet restant réel. On dit que h^* est une fonction de sous-estimation. Ainsi, si $h^*(e) = v$ alors le coût réel du trajet entre e et l'état final est forcément supérieur ou égal à v.
  - Si $h^*(e) = 0$ quelque soit e, la méthode est appelée "Branch and Bound" pure.
  - Si $h^*(e)$ est une fonction de sous-estimation du trajet restant entre e et l'état final, la méthode est appelée "Branch and Bound" avec sous-estimation.

Branch and Bound vs. Algorithme A*

- **Variante** de l'algorithme "Branch and Bound" avec programmation dynamique :
  Comme le problème du plus court chemin vérifie le principe d'optimalité (i.e. dans un chemin optimal, tout sous-chemin doit aussi être optimal), on peut réorganiser la file de priorité pour éliminer tous les sous-chemins menant vers un même sommet pour ne garder que le plus court d'entre eux. C'est une application de la programmation dynamique qui permettra de rendre l'algorithme encore plus efficace.
- **La méthode** A^* est tout simplement un algorithme Branch and Bound avec sous-estimation et programmation dynamique.
- Les algorithmes "Branch and Bound" et A* sont des **méthodes exactes** (on essaie de faire la plus grande coupure dans l'arbre de recherche tout en ne surestimant pas l'évaluation).

Procédure par Séparation et Evaluation Progressive

Méthode Branch and Bound :

1. Initialement, l'arbre est composé d'une seule feuille : la racine B et on connaît $e_{pd}(N)$.
2. On choisit parmi les feuilles, une feuille F dont on pense (par rapport à la valeur de $e_{pd}(F)$) qu'elle est la plus proche de la valeur optimale.
3. Séparer F en exhibant tous ses fils.
4. Evaluer e et e_{pd} sur les nouvelles feuilles.
5. On arrête l'itération lorsqu'il n'y a plus de feuille améliorante : $\forall F.\, e_{pd}(F) \geq vmin$ si on cherche une solution minimale.

Remarque : on ne considère dans (2) que les feuilles développables (solution non atteinte)

PSEP : exemple avec la traversée de la matrice

PSEP : Mise en oeuvre

- Les feuilles en attente sont gérées en une liste ordonnée E par valeurs croissantes de e_{pd}.
- On choisit l'élément de tête auquel on attribue un numéro de séparation, on le retire de la liste et on ajoute ses fils à la liste que l'on réordonne.
- Les états successifs de la liste sont pour la traversée de la matrice :
  - 1. [(1,1),0,41]
  - 2. [(2,1),8,45], [(1,2),4,46]
  - 3. [(3,1),20,45], [(1,2),4,46], [(2,2),27,52]
  - ...
- On obtient ainsi la valeur optimale mais si on désire calculer le vecteur solution optimale, on doit conserver l'arbre en mémoire.

PSEP : Algorithme

- Etat initial : liste E := racine de l'arbre
- Une solution admissible : sol_cour
- La valeur de cette solution : opt_cour

```
while E non vide do
    n = E.extract();
    if (EstUneSolution(n)){
      if (VraieValeur(n) > opt_cour){
        opt_cour = VraieValeur(n);
        sol_cour =n;
      }
    }
    if (Eval(n) > opt_cour)
      E.ajouter(Branch(n));
    else
      Elaguer(n);
}
```


Problème du sac à dos : application de la méthode Branch and Bound

Application numérique :
On prend a=(4,5,6,2) et v=(33,49,60,32) avec C=130.

Première étape :

- On trouve une première solution : la première évaluation est un majorant de toutes les solutions.
  Pour cela, on trie les produits selon leurs valeurs par unité de volume.
- Dans notre exemple :

$$\frac{4}{33} > \frac{5}{49} > \frac{6}{60} > \frac{2}{32}$$

- On divise E en 4 sous-ensembles. Le premier sommet évalué est $x_1 = 3$: on trouve une évaluation exacte : 12.

Problème du sac à dos : application de la méthode Branch and Bound

Deuxième étape : Algorithme :

```
Branch([X(1)... X(k-1)], Gain_Cour, Vrest){
    for i = PartieEntiere(Vrest/Vk) jqa 0
      Noeud[X(1)... X(k)=i), Gain_Cour + i*Gain[k], Vrest -
          i*Vol[k]];
}
```

```
Eval([X(1)... X(k)], Gain_Cour, Vrest){
    if Vrest < min{V(k+1)... Vn}{
        sol_cour = (X(1) ... X(k),0 ... 0);
        opt_cour = Gain_Cour;
    } else {
        eval = Gain_Cour + a(k+1) * Vrest/V(k+1);
        if (eval > opt_cour +1)
          Branch([X(1)... X(k)], Gain_Cour, Vrest);
        else
          Elaguer([X(1)... X(k)]);
    }
}
```

Problème du sac à dos : application numérique

Problème du sac à dos : application numérique

Application du problème du sac à dos

Le problème du sac à dos peut être généralisé à la relaxation de programmes linéaires en nombres entiers :

$$z = max\ cX$$

avec :

$$Ax \leq B$$

avec X positif et à composantes entières.

Plan

Méthodes par voisinage

- **Principe** : L'idée des méthodes par voisinage ou parcours de l'espace des solutions est de transformer une solution petit à petit, explorer l'espace des solutions en partant d'un point et en se "déplaçant" de proche en proche.
- Différentes méthodes :
  - L'algorithme de l'alpiniste (ou recherche d'un optimum local) ;
  - La méthode taboue ;
  - Le recuit simulé.
- Pour ce type de méthodes, il faut définir une notion de voisinage : permuter deux tâches dans un problème d'affectation de tâches, échanger un certain nombre d'objets (à fixer) dans le problème du sac à dos, etc.

Hill Climbing (algorithme de l'alpiniste)

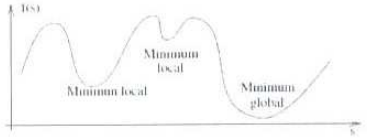
- **Principe** (Hill Climbing ou recherche d'un optimum local) : on part d'une solution "bonne" (donnée par une heuristique par exemple) et on balaie l'ensemble des voisins de cette solution.
  - s'il n'existe pas de voisin meilleur que notre solution, on a trouvé un optimum local et on arrête ;
  - sinon, on choisit le meilleur des voisins et on recommence.
  - une autre implémentation consiste à prendre le premier meilleur voisin trouvé à chaque étape (cf. algorithme "Best First Search").
- L'algorithme de l'alpiniste est analogue à la recherche en profondeur mais on développe les fils dans l'ordre croissant de la fonction heuristique h (cf. recherche en escalade).

Hill Climbing : algorithme

- Algorithme :

```
Sol = solution obtenue avec une heuristique
// variante non deterministe : tirage aleatoire
tant que Sol a un meilleur voisinage
    Sol := meilleur voisin de Sol;
```

- On obtient un **optimum local**.
- Le choix du voisinage est un important :
  - si le voisinage est trop petit, on risque de bloquer très vite dans un optimum qui ne donne pas une "bonne" solution.
  - si le voisinage est trop grand, chaque itération est très longue.



Hill Climbing : bilan

- Inconvénients de la méthode :
  - La solution obtenue est un optimum **local**. On peut lancer plusieurs recherches avec plusieurs solutions pour améliorer le résultat.
  - La convergence peut être lente : on peut fixer un nombre maximal d'itérations.
- La méthode de l'alpiniste correspond à la méthode de descente du gradient dans le cas discret.
  - on a alors : minimiser $y = f(x)$ à chaque pas, en mettant à jour $x := x - \epsilon * f'(x)$.
  - La méthode de descente du gradient est très utilisée, notamment pour trouver des paramètres (réseaux de neurones, etc.).

Best First Search

- **Principe :** c'est une recherche en largeur avec fonction d'estimation.
- **Algorithme** :
  - A chaque étape, on génère les successeurs d'un sommet, on les rajoute à une file de priorité contenant tous les sommets générés mais pas encore explorés.
  - La file est ordonnée par une fonction d'estimation afin que le prochain sommet à visiter soit toujours le meilleur.
  - La seule différence avec la recherche en largeur concerne le fait d'utiliser une file de priorité ordonnée par les valeurs de la fonction d'estimation donnée.

Méthode taboue

- **Contexte** : l'existence de minima locaux impose l'utilisation de méthodes d'exploration efficaces pour éviter de se retrouver bloqué aux alentours de ces minima. Plusieurs méthodes ont été proposées et ont souvent été inspirées par des phénomènes naturels.
- **Principe de la méthode taboue** : on veut pouvoir s'échapper de l'optimum local. Pour cela, on permet à la méthode de choisir 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 voisins visités.

Méthode taboue : liste taboue

La gestion de la liste taboue est la base de la méthode taboue.

- La liste est gérée en file FIFO ;
- Si la liste est trop courte, on n'arrive pas à s'échapper d'un minimum local ;
- 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 effectuées ;

La méthode taboue peut être vue comme une généralisation de la recherche d'un optimimum local.

Méthode du recuit simulé : un peu de métallurgie...

- Quand on chauffe un métal, il passe à l'état liquide, ce qui laisse beaucoup de liberté pour les atomes.
- Quand on abaisse la température, il y a donc de moins en moins de degrés de liberté, jusqu'à l'état solide. Suivant la manière dont on abaisse la température, on obtient différents solides :
  - Baisse brutale (trempe) : on obtient un "verre" structure amorphe (minimum local d'énergie).
  - Baisse progressive de la température : si on contrôle la température pour obtenir un minimum absolu d'énergie, on obtient un cristal.
  - Baisse trop rapide de la température : on obtient des défauts dans le cristal. Le "recuit" permet, si l'on réchauffe un peu, de redonner de la liberté aux atomes.

Méthode du recuit simulé

- Méthode découverte en 1982 par S. Kirkpatrick à partir de la méthode de Métropolis (1953) qui remarque que le problème des minima locaux est de même nature dans les deux cas.
- La méthode du recuit simulé est donc inspirée de la physique pour obtenir des états de faible énergie.
- **Idée :** on part d'un état instable avec une température très haute. On laisse refroidir lentement et l'organisation des différentes solutions s'effectue au fur et à mesure jusqu'à un état stable.
- La méthode du recuit simulé est une méthode non-déteministe.

Méthode du recuit simulé : principe

- On part d'une solution quelconque (a priori, il n'est pas forcément intéressant de partir d'une "bonne" solution).
- 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 (en analogie avec l'état instable), puis on accepte plus difficilement une solution plus coûteuse.

Méthode du recuit simulé

- Pour choisir une autre solution, on utilise la mécanique statistique : la distribution de Boltzmann mesure la probabilité $P(x)$ de visiter l'état X en fonction de son énergie $E(X)$ et de la température T :

$$P(X) = e^{-\frac{E(X)}{kT}}/N(T)$$

  avec k la constante de Bolzmann.
- On calcule l'augmentation de coût Δ ;
- Puis on calcule $accept = E^{-\Delta/T}$, T étant une quantité "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.

Méthode du recuit simulé : réglage de la température

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

- choisir la température initiale : elle doit être assez élevée pour que $accept$ soit assez grand au départ même si l'augmentation de coût est grande. La température initiale est un paramètre réglé 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 est aussi réglé par l'utilisateur.

Méthode du recuit simulé : applications

- La méthode du recuit simulé permet de donner des solutions pour le problème du voyageur de commerce avec 10000 villes.
- Domaines d'applications de la méthode :
  - Optimisation combinatoire (ordonnancement) ;
  - CAO (conception de circuits) ;
  - traitement d'images (restitution d'images floues) ;

Refroidissement rapide

Refroidissement lent

(Wikipedia)

Plan

Algorithmes génétiques

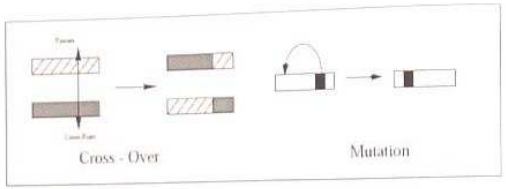
- Les algorithmes génétiques ont été proposés par J.H Holland ("Adaptation in Natural and Artifical Systems", 1976) puis développés par D. Goldberg.
- **Principe :** Le fonctionnement des algorithmes génétiques est calqué sur les critères de sélection naturelle (survie des individus les mieux adaptés et recombinaison génétique).
  Partant d'une population initiale constituée de solutions admissibles, on produit itérativement une population de taille raisonnable en ne conservant que les meilleurs individus (solutions).

Algorithmes génétiques : schéma algorithmique

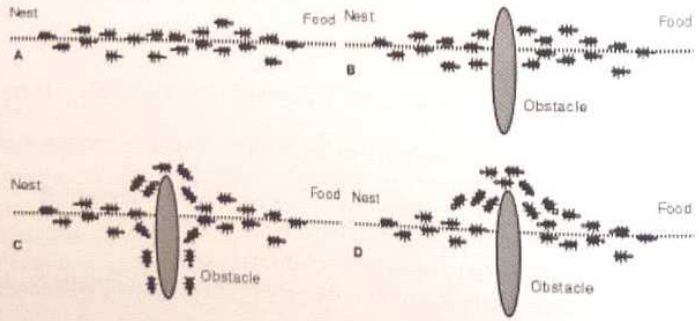
- Initialisation : générer un ensemble de solutions initiales ;
- Boucle principale :
  1. Evolution :
     - Sélection : choisir avec une probabilité proportionnelle à la qualité une liste d'individus ;
     - Reproduction : générer à partir de cette liste de nouveaux individus à l'aide des opérateurs génétiques ;
     - Remplacement : éliminer avec une probabilité inversement proportionnelle à leur qualité certains individus.
  2. Réactualisation de la meilleure solution ;
  3. Allez en 1 tant que le nombre de générations est inférieur ou égal à la valeur pré-déterminée.

Algorithmes génétiques : reproduction

- **La mutation** permet de diversifier les solutions en altérant de façon non déterministe un individu.
- **Le cross-over** consiste à générer à partir de deux individus deux nouvelles solutions composées chacune d'une partie des caractéristiques de leurs parents.



Colonie de fourmis



Colonie de fourmis

On place des fourmis sur les différentes solutions et chaque fourmi à son tour choisit une nouvelle ville de manière probabiliste en fonction de :

- la quantité de phéromone !
- une fonction de la distance à la prochaine solution ;
- sa mémoire interne qui lui indique les solutions déjà parcourues.

A chaque fois qu'une fourmi se déplace, elle laisse une trace. Celle qui a trouvé le chemin de longueur minimum laisse une trace plus importante.