

*Recherche Opérationnelle*  
*Troisième Partie*

Paul Feautrier

ENS Lyon

16 novembre 2005

# *Plan*

*Séparation et évaluation ou Branch-and-Bound*

*Programmation Dynamique*

*Recuit simulé*

*La méthode Tabou*

*Algorithmes génétiques*

*Conclusion générale*

## Séparation et évaluation I / II

- ▶ Plus communément appelée *branch and bound*. Soit un problème de la forme:

$$\begin{aligned} \min \quad & f(x) \\ g_i(x) \leq & 0, \quad i = 1, \dots, n \end{aligned}$$

- ▶ Principe. On construit un arbre de problèmes. Le problème initial est la racine.
- ▶ On s'arrange pour diviser le problème en deux (ou plus) sous-problèmes, par exemple en introduisant une contrainte supplémentaire, qui peut être satisfaite ou non.
- ▶ Le minimum peut appartenir à l'un quelconque des sous-problèmes.

## *Séparation et évaluation II / II*

- ▶ Si l'on peut prouver que l'un des sous problèmes est infaisable, on l'élimine.
- ▶ Si l'un des sous-problèmes est tellement contraint que sa solution est évidente, on note la valeur de sa solution.
- ▶ On cherche à obtenir une borne inférieure de la solution d'un sous-problème. Si elle est supérieure à la meilleure solution déjà obtenue, on élimine le sous-problème.
- ▶ Dans le cas restant, on subdivise de nouveau le sous-problème.

## *Variables bivalentes*

- ▶ La méthode est particulièrement bien adaptée à la résolution de problèmes linéaires en variables bivalentes: les inconnues ne peuvent prendre que les valeurs 0 ou 1.
- ▶ Pour séparer un problème en deux, on choisit l'une des inconnues, par exemple  $x_1$ , et on impose les contraintes  $x_1 = 0$  ou  $x_1 = 1$ .
- ▶ Pour obtenir une borne supérieure, on résout le problème en continu.
- ▶ Un sous-problèmes est résolu si toutes les variables sont fixées ou si la solution continue est entière.

## Exemple : le sac à dos

$$\max x_1 + 2x_2 + 3x_3 + 4x_4$$

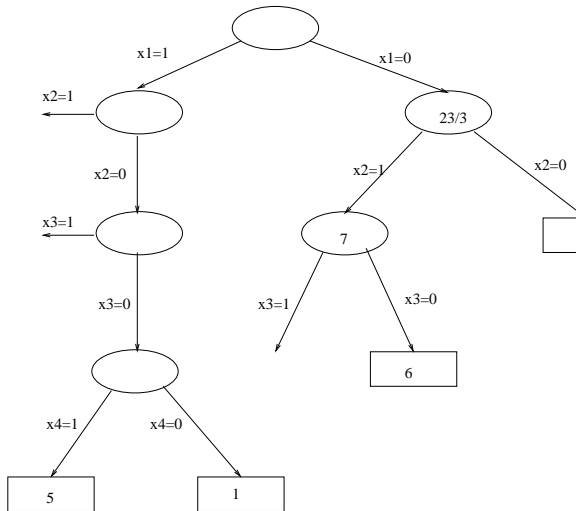
$$4x_1 + 3x_2 + 3x_3 + x_4 \leq 5$$

$$x_1, x_2, x_3, x_4 \in \{0, 1\}$$

- ▶ On veut emporter dans un sac à dos de contenance 5 une sélection de 4 objets de volumes respectifs 4, 3, 3, 1.
- ▶ Les *utilités* de ces objets sont respectivement de 1, 2, 3 et 4.
- ▶ Trouver la combinaison d'utilité maximum.

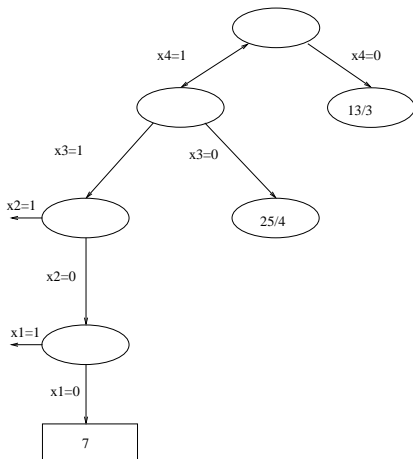
# Résolution I / II

On value les inconnues dans l'ordre  $x_1, \dots, x_4$ .



## Résolution II / II

Si on value les inconnues dans l'ordre  $x_4, \dots, x_1$ , la résolution est plus rapide.



## Méta-algorithme I / III

On doit d'abord définir une représentation des problèmes à résoudre. Par exemple, dans le cas du sac-à-dos, on notera le tableau du problème et celles des inconnues qui sont déjà valuées, dans une variable de type `pb`.

Fonctions de spécialisation :

- ▶ `is_trivial` : `pb`  $\rightarrow$  `bool` permet de savoir si le problème peut être résolu facilement (*i.e.* si toutes les inconnues, ou toutes les inconnues sauf une sont valuées).
- ▶ `trivial_solve` : `pb`  $\rightarrow$  `int` résout un problème trivial. Doit rendre  $+\infty$  si le problème trivial n'est pas faisable.
- ▶ `bound` : `pb`  $\rightarrow$  `int` donne une borne inférieure de la solution.
- ▶ `branch` : `pb`  $\rightarrow$  (`pb`, `pb`) découpe un problème en deux sous-problèmes.
- ▶ On utilise deux variables globales, `best` et `best_pb`.

## Méta-algorithme II / III

best\_pb:= ...

best:= MAXINT

**Algorithme** : BandB( $pb_0$ )

**if** is\_trivial ( $pb_0$ ) **then**

    local\_best := trivial\_solve( $pb_0$ )

**if** local\_best < best **then**

        best = local\_best

        best\_pb =  $pb_0$

**else**

    local\_best := bound( $pb_0$ )

**if** local\_best < best **then**

        ( $pb_1, pb_2$ ) := best\_pb( $pb_0$ )

        BandB( $pb_1$ )

        BandB( $pb_2$ )

## Méta-algorithme III / III

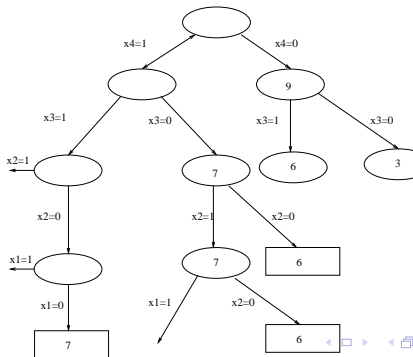
- ▶ Dans cette version, l'arbre des problèmes n'est pas représenté explicitement. Il est codé dans la suite des appels récursifs.
- ▶ La recherche se fait en profondeur d'abord. L'objectif est de trouver une solution le plus vite possible, pour pouvoir ensuite élaguer l'arbre.
- ▶ Dans le cas du sac-à-dos, on donne la priorité à la valeur  $x = 1$  pour éviter la solution triviale  $x_j = 0$ .
- ▶ La mémoire utilisée par l'algorithme est proportionnelle à la hauteur de l'arbre, *i.e.* au nombre de variables,  $n$ . D'autres versions utiliseraient une mémoire de taille  $O(2^n)$ .

## *Construction de l'arbre des problèmes*

- ▶ Que faire si les variables ne sont pas bivalentes ?
- ▶ Borner chaque variable,  $a \leq x \leq b$ , puis écrire en binaire la valeur de  $x - a$ , avec  $\log_2(b - a)$  bits. Engendre  $\log_2(b - a)$  variables équivalentes.
- ▶ De façon équivalente, partitionner à l'aide de contraintes  $x < (b + a)/2$  et  $x \geq (b + a)/2$ . Méthode plus générale, car on peut écrire des contraintes portant sur plusieurs variables.
- ▶ Il est possible de partitionner en plus de deux sous-problèmes. S'applique en particulier au cas où les variables ne sont pas numérisées.

## Évaluation I / IV

- ▶ La qualité de la fonction d'évaluation conditionne directement l'efficacité de la méthode.
- ▶ Exemple du sac à dos. On prend comme borne supérieure  $y_1 + 2y_2 + 3y_3 + 4y_4$  où  $y_i = x_i$  si  $x_i$  est évaluée et  $y_i = 1$  sinon. Il est clair que cette fonction donne bien une borne supérieure de l'utilité.



## Évaluation II / IV

- ▶ Relaxation continue. Si le problème est linéaire et en nombres entiers, on obtient une borne (inférieure ou supérieure) à partir de la solution continue. La méthode fournit une alternative à la méthode des coupes, à condition que le problème soit borné.
- ▶ Relaxation Lagrangienne. On a vu plus haut que si on sait calculer :

$$\begin{aligned}w(\lambda) &= \min_x L(x, \lambda), \\f^* &= \max_{\lambda \geq 0} w(\lambda)\end{aligned}$$

alors  $f^*$  est une borne inférieure du minimum cherché.

- ▶ Exploitation des propriétés de la fonction objectif, quand il est possible de calculer facilement son minimum en ne tenant compte que d'une partie des contraintes.

## Évaluation III / IV

- ▶ Linéarisation de la fonction objectif.
- ▶ Si les contraintes sont linéaires, on peut remplacer  $f$  par une minorante linéaire.
- ▶ Soit à minimiser une fonction contenant le produit de deux inconnues  $x$  et  $y$  dans un polyèdre (affectation quadratique).
- ▶ Si on sait d'après les contraintes que  $x \geq 0$  et  $y \geq b$ , on en déduit  $xy \geq bx$ . La solution du problème dont la fonction objectif est  $bx$  est un minorant de la solution du problème initial.

## *Évaluation IV / IV*

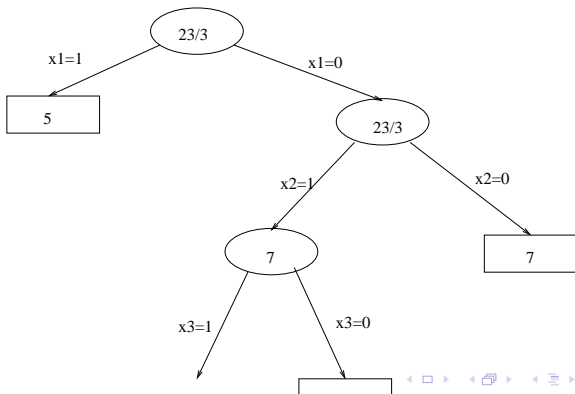
- ▶ Linéarisation des contraintes.
- ▶ On suppose la fonction objectif linéaire. On détermine un polyèdre qui contient l'ensemble des points entiers faisables. Si le domaine des points faisables est convexe, cette représentation peut être aussi précise qu'on le veut.
- ▶ La solution dans le polyèdre est un minorant de la solution du problème original.
- ▶ Exemple : Minimiser une fonction linéaire dans un disque.

## *Stratégie et Tactiques I / III*

- ▶ L'efficacité de la recherche dépend fortement :
  - ▶ De la qualité de la fonction d'évaluation.
  - ▶ De la disponibilité d'une première bonne solution. Noter qu'aucun élagage ne se produit tant qu'une solution n'est disponible.
  - ▶ De l'ordre de traitement des nœuds.
- ▶ Dans le méta-algorithme ci-dessus :
  - ▶ La fonction d'évaluation est cablée dans la fonction bound.
  - ▶ L'ordre de valuation des variables est cablé dans la fonction branch.
  - ▶ L'ordre de traitement des nœuds est cablé dans l'algorithme. C'est l'ordre en profondeur d'abord et de gauche à droite.
- ▶ Est-ce la meilleure stratégie ?

## Stratégie et Tactiques II / III

- ▶ La stratégie en profondeur d'abord fournit très vite (en  $n$  étapes) une solution, et utilise peu de mémoire. Mais elle peut s'égarer dans une région peu intéressante de l'arbre.
- ▶ Dans la stratégie en largeur d'abord, on construit l'arbre niveau par niveau.



## *Méta-algorithme (largeur d'abord)*

```
best_pb := ...;  best := MAXINT
queue :=  $\emptyset$ ;  insert(queue, pb0)
while queue  $\neq \emptyset$  do
  pbcurrent := pop(queue)
  if is_trivial(pbcurrent) then
    local_best := trivial_solve(pbcurrent)
    if local_best < best then
      best := local_best
      best_pb := pbcurrent
  else
    local_best := bound(pbcurrent)
    if local_best < best then
      branch(pbcurrent, pbleft, pbright)
      insert(queue, pbleft); insert(queue, pbright)
```

## Stratégie et Tactiques III / III

- ▶ Dans la stratégie «meilleur d'abord», on se base sur l'idée que la valeur de la borne inférieure est une estimation de l'optimum dans le sous-arbre.
- ▶ On a donc intérêt à développer d'abord le nœud qui a la meilleure borne.
- ▶ Dans l'exemple du sac-à-dos, les deux stratégies coïncident.
- ▶ Au niveau de l'implémentation, il suffit que les problèmes soient ordonnés par valeur (borne) croissante. On modifie les procédures `insert` et `pop`.
- ▶ On peut utiliser une structure de données plus adaptée, un *tas* par exemple.

## *Implémentation parallèle*

- ▶ L'algorithme par séparation et évaluation se prête bien à une implémentation parallèle, parce que le développement de chaque problème est indépendant des autres problèmes.
- ▶ La seule dépendance est celle sur la meilleure solution, best. Mais on peut accepter que cette valeur soit ajustée avec retard, la seule incidence étant un élagage moins efficace.
- ▶ Paradoxe: l'élagage peut être plus efficace si le parallélisme permet d'atteindre plus vite la région de l'arbre où se trouve le minimum.

# *Plan*

*Séparation et évaluation ou Branch-and-Bound*

*Programmation Dynamique*

*Recuit simulé*

*La méthode Tabou*

*Algorithmes génétiques*

*Conclusion générale*

## Programmation dynamique

- ▶ Méthode inventée par R. Bellman en 1956.
- ▶ Conçue sur le modèle de l'algorithme du plus court chemin dans un graphe.
- ▶ On remplace la résolution d'un problème de taille  $n$  par celle d'un certain nombre de problèmes de taille  $n - 1$ , dont on combine les résultats.
- ▶ Exemple : calcul du plus court chemin entre deux points d'un DAG,  $i$  et  $j$ .
- ▶ Si  $i = j$ , alors la longueur du plus court chemin est 0. Sinon, ce chemin passe nécessairement par l'un des successeurs de  $i$ .
- ▶ Soit  $l_{ij}$  la longueur du plus court chemin, et  $d_{ij}$  la distance de deux sommets *adjacents*. On a la récurrence :

$$l_{ij} = \min_{k \in \text{Succ}(i)} d_{ik} + l_{kj} = \min_{k \in \text{Pred}(j)} l_{ik} + d_{kj}.$$

## Exemple du sac-à-dos

Soit à résoudre :

$$\begin{aligned} \min \quad & \sum_{i=1}^n c_i x_i, \\ \sum_{i=1}^n w_i x_i & \leq W, \\ x_i & \in \{0, 1\} \end{aligned}$$

- ▶ On suppose les  $w_i$  entiers.
- ▶ On considère la famille de problèmes obtenue en faisant varier  $W$  et en ne prenant en compte que les variables  $x_i, i = k, \dots, n$ . Soit  $V_k(W)$  la valeur d'un tel problème.

## Équations de récurrence

- ▶ Soit à calculer  $V_1(W)$ . Il y a deux choix possibles :  $x_1 = 0$ ,  $x_1 = 1$ .
- ▶ Un fois la valeur de  $x_1$  fixée, on doit résoudre un autre problème de sac-à-dos.
- ▶ Si  $x_1 = 0$ , la capacité disponible est toujours  $W$ , on doit calculer  $V_2(W)$ .
- ▶ Si  $x_1 = 1$ , il ne reste plus que  $W - w_1$  unités de capacité, mais on a déjà obtenu  $c_1$  unités de valeur.
- ▶ Dans le cas général, on a la récurrence :

$$V_k(W) = \max\{V_{k+1}(W), V_{k+1}(W - w_k) + c_k\}.$$

## Conditions aux limites

- ▶ Que vaut  $V_n(W)$  ?
- ▶ On ne peut fixer que la valeur de  $x_n$ , et on doit avoir  $w_n \leq W$  si on veut pouvoir faire  $x_n = 1$ . On a donc :

$$V_n(W) = \mathbf{if } W \geq w_n \mathbf{if } c_n \mathbf{Else } 0.$$

- ▶ De plus,  $V(k, W) = 0$  si  $W \leq 0$ .
- ▶ Méthode de résolutions. On remarque que  $0 \leq W \leq B = \sum_{i=1}^n w_i$ .
- ▶ Il suffit donc de calculer  $V_k(W)$  pour les  $B$  valeurs  $[1, \dots, B]$  et pour  $k = n, \dots, 1$ .

## Algorithme

```
#define N ...
#define B ...
int w[N+1];
int c[N+1];

void main(void){
    int V[N+1][B+1];
    int W, k, c1, c2;
    for(W=1; W <= B; W++){
        if(w[N] <= W)
            V[N][W] = c[N];
        else V[N][W] = 0;
    }
    for(k=N-1; k >= 1; k--){
        for(W = 1; W <= B; W++){
            c1 = V[k+1][W];
            if(W - w[k] > 0)
                c2 = V[k+1][W - w[k]] + c[k];
            else c2 = 0;
```

## Remarques

- ▶ On lit directement la valeur du problème original dans  $V_{1W}$ .
- ▶ Noter que l'on a résolu non seulement le problème original, mais le problème pour toutes les valeurs possibles de  $W$ .
- ▶ Dans ce cas particulier, comme  $V_k(W)$  ne dépend que de  $V_{k+1}(W')$  pour  $W' < W$ , on peut ne calculer les valeurs que pour les valeurs de  $W$  au plus égale à la contrainte du problème original.

## Exemple

$$\max x_1 + 2x_2 + 3x_3 + 4x_4$$

$$4x_1 + 3x_2 + 3x_3 + x_4$$

$$x_1, x_2, x_3, x_4$$

$$\leq 5$$

$$\in \{0, 1\}$$

11	10	9	7	4
10	9	9	7	4
9	9	9	7	4
8	9	9	7	4
7	9	9	7	4
6	7	7	7	4
5	7	7	7	4
4	7	7	7	4
3	4	4	4	4
2	4	4	4	4
1	4	4	4	4
0	0	0	0	0

## Complexité

- ▶ Il est facile de voir que la complexité est  $O(n.B)$ .
- ▶ On dit que le problème est pseudo-polynomial. En effet,  $n.B$  est bien un polynôme, mais  $B$  est exponentiel en la taille des  $w_k$  (nombre de bits).
- ▶ L'ensemble  $[0, B]$  est «l'ensemble des états» de l'algorithme. Il peut être à plusieurs dimensions. Dans ce cas, la complexité devient prohibitive.

## Reconstituer la solution

- ▶ Il suffit d'introduire un nouveau tableau  $x[N][B+1]$ .
- ▶ On modifie l'algorithme comme suit :

```
    if(W - w[k] > 0){
        c2 = V[k+1][W - w[k]] + c[k];
    else c2 = 0;
    if(c1 > c2){
        V[k][W] = c1;  X[k][W] = 1;
    } else {
        V[k][W] = c2;  X[k][W] = 0;
    }
}
```

- ▶ On reconstitue la valeur des  $x_i$  par la règle :

```
W = W_0;
for(k=1; k<=N; k++){
    x[k] = X[k][W];
    W = W - x[k]*w[k];
}
```

## Économiser la mémoire

- ▶ Si l'on n'est intéressé que par la valeur de l'optimum, il est facile de voir que l'on n'a besoin que de deux lignes du tableau  $V$ ,  $V[k]$  et  $V[k+1]$ .
- ▶ Il suffit donc de  $2B$  cellules de mémoire. Les deux lignes sont permutées à chaque itération.
- ▶ Ce cas se présente quand la programmation dynamique est composante d'un autre algorithme, par exemple un *branch-and-bound*.
- ▶ Si on souhaite reconstituer la solution, il faut soit stocker le tableau  $x$ , de taille  $N.B$ , soit effectuer des calculs redondants.

## Généralisation I / III

- ▶ Soit à résoudre :

$$\begin{aligned} \min_{x \in \Omega^n} \quad & f_n(x), \\ g_n(x) \quad & \leq 0. \end{aligned}$$

- ▶ Il est évident que la méthode ne marche que parce que la fonction objectif et les contraintes ont des propriétés spéciales.
- ▶ La fonction objectif doit être séparable. Soit  $x$  un vecteur. On note  $x_{hd}$  la première composante de  $x$  et  $x_{tl}$  le vecteur des autres composantes.
- ▶ Une fonction  $f_n$  à  $n$  variables est séparable s'il existe une fonction  $h_n$  et une fonction  $f_{n-1}$  telles que:

$$\begin{aligned} f_n(x) &= h_n(x_{hd}, f_{n-1}(x_{tl})), \\ \frac{\partial h_n(x, y)}{\partial y} &\geq 0 \end{aligned}$$

## Généralisation II / III

### Théorème

$$\min_{g_n(x) \leq 0} f_n(x) = \min_{x_{hd} \in \Omega} h_n(x_{hd}, \min_{g_n(x_{hd}, x_{tl}) \leq 0} f_{n-1}(x_{tl}))$$

### Démonstration.

Soit  $x^*$  la solution du problème de droite, et soit  $y^*(x_{hd})$  le point où  $f_{n-1}(y)$  atteint son minimum sous la contrainte  $g_n(x_{hd}, x_{tl}) \leq 0$ . Par construction,  $\langle x^*, y^*(x^*) \rangle$  satisfait la contrainte, donc  $f_n(x^*, y^*) \geq \max_{x \in \Omega^n} f_n(x)$ .

Réciproquement, soit  $\bar{x}$  la solution du problème de gauche. On a  $\min_{g(\bar{x}_{hd}, x_{tl}) \leq 0} f_{n-1}(x_{tl}) \leq f_{n-1}(\bar{x}_{tl})$ , donc, par la monotonie de  $h_n$

$$h_n(x_{hd}, \min_{g(x_{hd}, x_{tl}) \leq 0} f_{n-1}(x_{tl})) \geq h_n(\bar{x}_{hd}, f_{n-1}(\bar{x}_{tl})),$$

et cette propriété s'étend au minimum. □

## Généralisation III / III

- ▶ Cette propriété ne suffit pas. Si la décomposition de  $f_n$  peut être poursuivie, elle fournit une récurrence permettant de calculer le minimum, mais sa complexité est du même ordre que celle d'une recherche exhaustive.
- ▶ Pour aller plus loin, il faut plonger le problème initial dans une famille de problèmes où les contraintes dépendent d'une *variable d'état*  $S \in \mathcal{S}$ .

$$\begin{aligned} P_n(S) : \quad & \min f_n(x), \\ g_n(x, S) & \leq 0, \end{aligned}$$

où  $g_n$  a les deux propriétés :

- ▶  $g_n(x, S) = g_{n-1}(x_{tl}, t_n(x_{hd}, S))$
- ▶  $\forall x \in \Omega, S \in \mathcal{S} : t_n(x, S) \in \mathcal{S}$ .

## Algorithmme

- ▶ On suppose que l'ensemble des valeurs possibles de  $x$ ,  $\Omega$  et celui des valeurs possible de  $S$ ,  $\mathcal{S}$  sont finis.

**for**  $S \in \mathcal{S}$

$$V_n(S) = \min_{g_n(x,S) \leq 0} f_n(x)$$

**for**  $k = n - 1$  **downto** 1

**for**  $S \in \mathcal{S}$

$$V_k(S) = \min_{x \in \Omega} h_k(x, V_{k+1}(t_k(x, S)))$$

- ▶ L'efficacité de la méthode est entièrement conditionnée par la taille de  $\mathcal{S}$  qui doit être énuméré.
- ▶ Les problèmes d'optimisation à une dimension sur  $\Omega$  peuvent être résolus par énumération (si  $|\Omega|$  est petit) ou par toute autre méthode.

## Application au sac-à-dos

- ▶ La fonction objectif se met bien sous la forme :

$$\sum_{i=1}^n c_i x_i = c_1 x_1 + \sum_{i=2}^n c_i x_i.$$

- ▶ La fonction  $h$  est l'addition, qui est bien monotone croissante en son deuxième argument.
- ▶ La contrainte se met sous la forme :

$$\sum_{i=1}^n w_i x_i - W \leq 0,$$

et on peut écrire :  $\sum_{i=2}^n w_i x_i - (W - w_1 x_1) \geq 0$ .

- ▶ La fonction de transition est  $t_k(x, W) = W - w_k x$ .
- ▶ L'espace des états est  $\mathcal{S} = [0, B]$  avec  $B = \sum_{i=1}^n w_i$ , l'espace des valeurs est  $\Omega = [0, 1]$ .

## *Autres exemples*

- ▶ On peut envisager d'autres formes de la fonction objectif, comme :

$$\prod_{i=1}^n x_i^{c_i}.$$

- ▶ Ici la fonction de combinaison est la multiplication, qui est bien non décroissante en son deuxième argument à condition que le premier argument soit non négatif.

## *Le voyageur de commerce*

- ▶ On se donne  $n$  villes et une matrice des «distances»  
 $\{d_{ij} \mid 1 \leq i \leq n, 1 \leq j \leq n\}$ . Certaines distances peuvent être infinies.
- ▶ On demande de trouver une tournée de longueur minimale.
- ▶ Une tournée est un circuit hamiltonien, *i.e.* qui passe une fois et une seule par chaque ville.
- ▶ Le problème est très difficile, parce que le nombre de circuits (hamiltoniens) est très élevé.

## Équation de récurrence I / II

- ▶ On peut choisir une ville arbitraire, par exemple la ville 1, comme point de départ de la tournée.
- ▶ Soit  $S$  un sous ensemble de  $[1, n]$  contenant 1. On considère les chemins hamiltoniens de  $S$ , c'est-à-dire les chemins partant de 1 et passant une fois et une seule par chaque ville de  $S$ .
- ▶ On note  $Pred(k)$  l'ensemble des prédécesseurs de  $k$ , c'est-à-dire l'ensemble :

$$\{i \in [1, n] \mid d_{ik} < \infty\}.$$

- ▶ On note  $F(S, k)$  la longueur du plus court chemin de 1 à  $k$  qui passe une et une seule fois par toutes les villes de  $S$ .

## Équation de récurrence II / II

- ▶ Considérons la ville  $k'$  qui précède  $k$  dans le plus court chemin cherché. Cette ville fait évidemment partie de  $Pred(k) \cap S$ .
- ▶ D'autre part, le chemin de 1 à  $k'$  passe par toute les villes de  $S - \{k\}$ . Si sa longueur est supérieure à  $F(S - \{k\}, k')$ , il est possible de l'améliorer. On a donc la récurrence :

$$F(S, k) = \min_{k' \in Pred(k) \cap S} F(S - \{k\}, k') + d_{k'k}.$$

- ▶ On a d'autre part la condition initiale  $F(\{1\}, 1) = 0$ .
- ▶ On peut donc résoudre le problème en tabulant la fonction  $F$  pour tous les éléments de  $S$ , puis pour tous les sous-ensembles de villes à 2, 3, ... éléments.
- ▶ On lit le résultat en  $F([1, n], 1)$ .
- ▶ Mais la méthode n'est pas très efficace car il y a en tout  $2^{n-1}$  sous-ensembles.

## *Relaxation*

- ▶ On peut simplifier le problème en demandant que le chemin passe par  $|S|$  villes sans exiger qu'il soit hamiltonien.
- ▶ On peut alors traiter en une seule fois tous les sous-ensembles de même cardinal.
- ▶ On obtient une borne inférieure de la longueur de la tournée, qui peut être utilisée, par exemple, dans un algorithme *branch and bound*.

# *Plan*

*Séparation et évaluation ou Branch-and-Bound*

*Programmation Dynamique*

*Recuit simulé*

*La méthode Tabou*

*Algorithmes génétiques*

*Conclusion générale*

## Position du problème

Soit à résoudre :

$$\max_{x \in S} f(x).$$

- ▶ L'ensemble  $S$  est l'ensemble des configurations, ou ensemble des états. On suppose qu'il est trop grand pour être énuméré, mais qu'il existe un procédé efficace pour tirer une configuration au hasard dans  $S$ .
- ▶  $f$  est la fonction objectif.
- ▶ On suppose que tout point  $x$  de  $S$  a un ensemble fini de voisins,  $V(x)$ . Étant donné  $x$ , il existe un procédé efficace pour énumérer  $V(x)$ . Enfin, l'espace des configurations est connexe, *i.e.* on peut aller de  $x$  à  $y$  quelconque par des déplacements de voisin à voisin.
- ▶ Un maximum global de  $f$  est un point  $x^*$  de  $S$  tel que  $\forall x \in S : f(x) \leq f(x^*)$ .
- ▶ Un maximum local est un point  $\bar{x}$  tel que  $\forall x \in V(\bar{x}) : f(x) \leq f(\bar{x})$ .

## *Amélioration itérative*

### **Algorithme : Hill-Climbing**

best :=  $-\infty$

**for**  $i = 1$  to  $n$  **do**

$x := \text{random}(S)$

$z := x$

**repeat**

**foreach**  $y \in V(x)$  **do**

**if**  $f(y) > f(x)$  **then**

$x := y$

**break**

**until**  $x = z$

**if**  $f(x) > \text{best}$  **then**

$\text{best} := f(x)$

$\text{xbest} := x$

## *Bassin d'attraction*

- ▶ Autour de chaque maximum local  $a$  il existe un bassin d'attraction  $A$  défini par :

$$a \in A, \\ (\forall y \in V(x) : f(y) > f(x) \Rightarrow y \in A) \Rightarrow x \in A.$$

- ▶ Si le point initial est dans le bassin d'attraction du maximum global, l'algorithme Hill-Climbing trouve le maximum global.
- ▶ La probabilité d'atteindre le minimum global tend donc vers 1 quand  $n \rightarrow \infty$ .
- ▶ Mais il est impossible d'avoir une certitude.

## Recuit simulé I / III

- ▶ Algorithme de Métropolis, 1953.
- ▶ Voir aussi P. J. M. van Larrhoven ans E.H. Aarts, *Simulated Annealing, Theory and Applications*, Kluwer, 1987.
- ▶ Analogie avec la physique statistique. Lorsqu'un objet physique pouvant exister dans plusieurs états atteint l'équilibre thermodynamique, la population d'un état d'énergie  $E$  est donnée par la loi de Boltzmann :

$$p(E) = \frac{1}{Z} e^{-\frac{E}{kT}},$$

où  $k$  est la constante de Boltzmann,  $T$  la température, et  $Z$  un facteur de normalisation, la fonction de partition.

- ▶ Cet équilibre est dynamique. Il résulte de multiples transitions qui se compensent. Une transition faisant varier l'énergie de  $\Delta E$  a une probabilité proportionnelle à  $e^{-\frac{\Delta E}{kT}}$ .
- ▶ On modélise un problème d'optimisation par un système thermodynamique dont on simule l'évolution.

## Recuit simulé II / III

- ▶ Les éléments de  $S$  sont les divers états possibles du système. L'«énergie» de l'état  $x$  est  $f(x)$ .
- ▶ Si le système est dans l'état  $x$ , il peut passer aléatoirement dans l'un des états de  $V(x)$ .
- ▶ La probabilité pour le système de passer dans l'état  $y \in V(x)$  est :
  - ▶  $\frac{1}{Z}$  si  $f(y) > f(x)$  ;
  - ▶  $\frac{1}{Z} e^{\frac{f(y)-f(x)}{T}}$  si  $f(y) < f(x)$ .
- ▶  $Z$  est le facteur de normalisation.
- ▶  $T$  est une pseudo-température.
- ▶ Si le problème a des contraintes qui ne sont pas intégrées dans la définition de  $S$ , il suffit de faire  $f(x) = -\infty$  quand  $x$  n'est pas faisable.

## *Recuit simulé III / III*

- ▶ Au bout d'un certain temps, le système se stabilise.
- ▶ On réduit la valeur de la «température» et on continue.
- ▶ La motivation est de permettre au système simulé de s'échapper du bassin d'attraction d'un minimum local pour atteindre le minimum global.

## Algorithm Simulated-Annealing

```
 $T := \dots$   
 $x := x_{best} := \text{random}(S)$   
 $best = f(x)$   
for  $i := 1$  to  $n$  do  
  for  $j := 1$  to  $m$  do  
     $y := \text{random}(V(x))$   
    if  $f(y) > f(x)$  then  
       $x := y$   
      if  $f(x) > best$  then  
         $best := f(x)$   
         $x_{best} := x$   
      else  $v = \text{random}([0, 1])$   
      if  $v < e^{\frac{f(y)-f(x)}{T}}$  then  $x := y$   
   $T := 0.99T$ 
```

## *Analyse du recuit simulé*

- ▶ Quand doit-on baisser la température (*i.e.*, quel est la valeur de  $m$ ) ?
- ▶ Combien de fois doit on baisser la température (*i.e.* quelle est la valeur de  $n$ ) ?
- ▶ A quelle vitesse faire baisser la température (la valeur 0.99 est elle la bonne) ?

## Chaîne de Markov

- ▶ Une chaîne de Markov uniforme, discrète et à temps discret est un système qui peut exister dans plusieurs états formant un ensemble  $S$ . Soit  $x_n$  l'état du système à l'état  $n$ . La chaîne ne change d'état qu'aux instants de valeur entière.
- ▶ La probabilité de transition de l'état  $x$  à l'état  $y$  ne dépend que de  $x$  et de  $y$ . Elle ne dépend ni du temps, ni des états antérieurs occupés par le système. Soit  $P_{xy}$  la probabilité de transition de l'état  $x$  à l'état  $y$ .
- ▶ On doit avoir  $\sum_{y \in S} P_{xy} = 1$ . La matrice  $p_{xy}$  est une «matrice stochastique».
- ▶ Il est clair qu'un algorithme de recuit simulé fonctionne comme une chaîne de Markov.

## Calcul des probabilités de transition

- ▶ Chaque tirage peut avoir trois résultats :
  - ▶ On tire un point  $y$  tel que  $f(y) > f(x)$ .
  - ▶ On tire un point  $y$  tel que  $f(y) < f(x)$  et un nombre  $v$  tel que  $v < e^{\frac{f(y)-f(x)}{T}}$ .
  - ▶ Dans le cas restant, on recommence le tirage.
- ▶ Fonction de partition :
$$Z = \sum_{f(y) \geq f(x)} 1 + \sum_{f(y) < f(x)} e^{\frac{f(y)-f(x)}{T}}.$$
- ▶ Probabilité de la transition  $x \rightarrow y$  :
  - ▶  $\frac{1}{Z}$  si  $f(y) \geq f(x)$  ;
  - ▶  $\frac{1}{Z} e^{\frac{f(y)-f(x)}{T}}$  sinon.

## *Graphe d'une chaîne de Markov*

- ▶ Le graphe d'une chaîne de Markov a pour sommets les états de la chaîne. Il y a un arc de  $x$  vers  $y$  si et seulement si la transition  $x \rightarrow y$  est de probabilité non nulle.
- ▶ On peut construire les composantes fortement connexes (cfc) de ce graphe et le graphe réduit.
- ▶ En vertu du principe que tout évènement de probabilité non nulle finit par se produire au bout d'un temps suffisamment long, l'état de la chaîne finit toujours par parvenir dans l'une des cfc terminale.
- ▶ On dit que la chaîne est simple s'il n'y a qu'une seule cfc terminale.
- ▶ Une chaîne simple finit par atteindre sa cfc terminale. A partir du moment où elle entre dans la cfc terminale, elle passe une infinité de fois par chaque sommet de celle-ci.

## *Distribution limite*

- ▶ Si  $p_x^0$  est la distribution de probabilité initiale sur la cfc terminale, la distribution après  $n$  étapes est égale à :

$$p^n = p^{n-1}.P$$

- ▶ Le comportement de  $p^n$  dépend donc des valeurs propres de  $P^T$ . On montre :
  - ▶ Que 1 est valeur propre de  $P^T$ .
  - ▶ Que les composantes de  $p^n$  sont positives, que leur somme est égale à 1, et par conséquent, qu'elles sont comprises entre 0 et 1.
  - ▶ En conséquence,  $P^T$  ne peut avoir de valeur propre de module  $> 1$ .
  - ▶ On est dans le cas régulier quand 1 est valeur propre simple de  $P^T$ . Une condition suffisante est que tous les coefficients de  $P$  soient non nuls.
  - ▶ Dans ce cas, la distribution de probabilité converge vers le vecteur propre de  $P^T$  associé à 1.

## *Application au recuit simulé*

- ▶ Dans le cas du recuit simulé, aucune probabilité de transition n'est nulle. On est donc toujours dans le cas régulier.
- ▶ Quand la température décroît, certaines probabilités de transition tendent vers 0. A la limite, chaque cfc est un ensemble contigu de minimum locaux. La ou les cfc terminales sont associées à l'extremum global.
- ▶ Pour arrêter la recherche, on peut attendre que la distribution de probabilité soit stable (par exemple en estimant l'espérance mathématique de  $x$ , si cela à un sens, ou celle de  $f(x)$ .)

## *Exemple, Sac-à-dos*

- ▶ L'ensemble des états est celui des suites binaires de taille  $n$  qui vérifient la contrainte.
- ▶ Voisinage : deux suites qui diffèrent par un seul bit.
- ▶ Sur un petit exemple, le résultat n'est pas très satisfaisant.

## *Exemple, Voyageur de Commerce*

- ▶ On suppose que la matrice des distances est symétrique.
- ▶ Ensemble des états : ensemble des circuits hamiltoniens du graphe.
- ▶ Voisinage.
  - ▶ On choisit deux villes  $a$  et  $b$  visitées dans l'ordre  $a \rightarrow b$ . Soit  $a'$  la ville qui précède  $a$  et  $b'$  celle qui suit  $b$ .
  - ▶ On construit le circuit  $a' \rightarrow b \rightarrow a \rightarrow b' \rightarrow a'$ .
- ▶ Les résultats expérimentaux sont excellents.

# *Plan*

*Séparation et évaluation ou Branch-and-Bound*

*Programmation Dynamique*

*Recuit simulé*

*La méthode Tabou*

*Algorithmes génétiques*

*Conclusion générale*

## *La méthode Tabou*

- ▶ On conserve les notions d'espace de configuration et de voisinage.
- ▶ Pour éviter de rester bloqué autour d'un optimum local, on conserve la liste des derniers points visités, la liste tabou.
- ▶ Quand on explore un voisinage, on choisit le meilleur voisin à l'exclusion des points de la liste tabou.
- ▶ La liste tabou est gérée comme une FIFO.
- ▶ Sa longueur est un paramètre crucial. Folklore : la valeur  $L = 7$  est presque toujours suffisante!

# Algorithmme

## Algorithmme : Tabou

$tabou := \emptyset$

$x := \dots$

**for**  $i := 1$  to  $n$  **do**

    add\_FIFO ( $tabou, x$ )

$x_{best} := \perp$ ;  $best := +\infty$

**foreach**  $y \in V(x)$  **do**

**if**  $y \notin tabou$  **then**

**if**  $f(y) < best$  **then**

$x_{best} := y$ ;  $best := f(y)$

**if**  $x_{best} \neq \perp$  **then**

$x := x_{best}$

        add\_FIFO ( $tabou, x$ )

**else return**

## *Evaluation*

- ▶ Contrairement au recuit simulé, il n'y a pas de théorie de la méthode Tabou.
- ▶ Elle est cependant réputée plus efficace et fiable que le recuit simulé.
- ▶ Le choix de la longueur de la liste Tabou (NB- qui est cachée dans la fonction `add_FIFO`) est le plus important.
- ▶ L'autre paramètre est le nombre d'essais ( $n$  ci-dessus) qui ne peut guère être choisi que de façon expérimentale.

# *Plan*

*Séparation et évaluation ou Branch-and-Bound*

*Programmation Dynamique*

*Recuit simulé*

*La méthode Tabou*

*Algorithmes génétiques*

*Conclusion générale*

## *Algorithmes génétiques I / II*

- ▶ Principe : on cherche à imiter ce que l'on sait du fonctionnement de l'évolution biologique.
- ▶ Chaque configuration doit pouvoir être codée par une chaîne de caractères.
  - ▶ Pour le sac-à-dos, on prend la chaîne 0-1 des valeurs des variables.
  - ▶ Pour le voyageur de commerce, on prend la liste des villes dans l'ordre de la tournée.
- ▶ On travaille non pas sur une configuration, mais sur une *population* de configurations, que l'on tire au hasard au début.

## Algorithmes génétiques II / II

- ▶ On évalue  $f$  pour tous les individus de la population (parallélisme) on classe et on retient les meilleurs 20% (par exemple).
- ▶ On complète la population à l'aide de 2 mécanismes :
  - ▶ Mutation : on choisit un individu au hasard et on modifie (avec une probabilité très faible) l'une de ses lettres.
  - ▶ Croisement : On choisit deux individus  $x$  et  $y$ , on sélectionne un point de croisement  $x = a.b, y = c.d$ , et on forme les individus  $a.d$  et  $c.b$  (la longueur de  $a$  et celle de  $c$  doivent être égales).
- ▶ On itère jusqu'à stabilisation de la population.

## Algorithme

**Algorithme :** *Genetic*( $P, N, L, Q$ )

**for**  $i := 1$  to  $P$  **do**  $x[i] := \text{random}(S)$

**for**  $k := 1$  to  $N$  **do**

$\text{sort}(x, P, f)$

$l := m := p/5$

**for**  $q = 1$  to  $Q$  **do**

$i := \text{random}([1, m])$

$x[l + 1] := \text{mutate}(x[i]); l += 1$

**while**  $l < P$  **do**

$i = \text{random}([1, m])$

$j = \text{random}([1, m])$

**if**  $i \neq j$  **then**

$\text{cut} := \text{random}([2, L - 1])$

$(u, v) := \text{crossover}(x[i], x[j], \text{cut})$

$x[l] := u; \quad x[l + 1] := v; \quad l := l + 2$

## *Evaluation*

- ▶ L'originalité essentielle est de travailler sur une population et non sur un individu unique. Similaire à l'idée du redémarrage aléatoire, mais permet une évaluation parallèle si c'est possible.
- ▶ L'idée des mutations est similaire à l'exploration aléatoire d'un voisinage.
- ▶ L'idée de recombinaison peut avoir du sens ou non.
  - ▶ Elle repose sur l'idée que les gènes ont un effet cumulatif et non positionnel.
  - ▶ C'est faux pour le sac-à-dos, pour le voyageur de commerce, et peut-être aussi dans la nature.
- ▶ La méthode ne s'applique donc efficacement que dans des cas particuliers.

# *Plan*

*Séparation et évaluation ou Branch-and-Bound*

*Programmation Dynamique*

*Recuit simulé*

*La méthode Tabou*

*Algorithmes génétiques*

*Conclusion générale*

## Conclusion générale

- ▶ Tout un ensemble de méthodes, des plus particulières (la programmation linéaire et les méthodes de gradient) aux plus générales (les méta-heuristiques).
- ▶ Les méthodes particulières sont plus difficiles à programmer, mais plus efficaces.
- ▶ Les méthodes générales sont faciles à programmer et d'un champ d'application plus vaste.
- ▶ Donc choisir toujours la méthode la plus spécialisée compatible avec la définition du problème (il existe beaucoup de logiciels tout faits, libres et commerciaux).
- ▶ La méthode *Branch and Bound* demande une étude préliminaire du problème. Elle est très générale et pas trop difficile à programmer. Elle justifie à elle seule le principe de Minoux : «En optimisation combinatoire, la linéarité n'est pas importante».