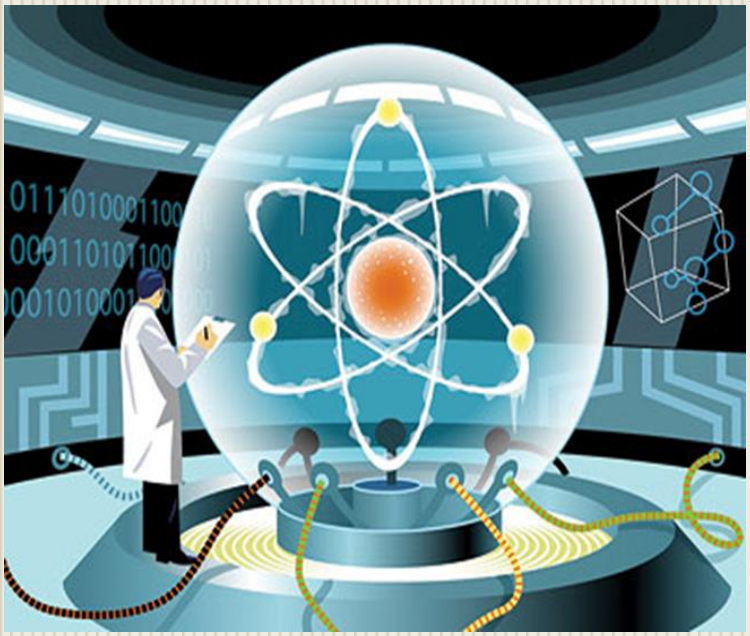
Recherche opérationnelle Avancée

Cours & Exercices



Authored by: Abdesslem LAYEB

1.1 Introduction

L'**optimisation** est une branche des mathématiques consistant à rechercher des conditions ou des configurations optimales pour des systèmes variés. Ce mot nous vient du latin optimum qui signifie le meilleur.

Elle est très importante en analyse numérique et dans les mathématiques appliquées, fondamentales pour l'industrie et l'ingénierie. En effet, lorsqu'un phénomène économique, physique, chimique... est exprimé par des équations, il est nécessaire d'optimiser le système afin d'obtenir un rendement maximal ou une configuration idéale. Pour cela on utilise des outils mathématiques. Aujourd'hui, tout est optimisé : Le fonctionnement d'un moteur, la gestion des lignes ferroviaires, les investissements économiques, les réactions chimiques... etc. Les exemples sont multiples, il est donc crucial de posséder les outils pour résoudre ces problèmes. L'objectif consiste à maximiser une fonction f appelée *la fonction objectif.* Les éléments de l'espace de définition de la fonction f sont appelées les solutions admissibles. Dans le cas où les solutions admissibles appartiennent à l'ensemble des entiers N, on parle de l'**optimisation combinatoire**. L'optimisation combinatoire est un outil indispensable combinant diverses techniques des mathématiques discrètes et de l'informatique afin de résoudre des problèmes de la vie réelle. D'une manière simple, résoudre un problème d'optimisation combinatoire consiste à trouver l'optimum d'une fonction, parmi un nombre fini de choix, souvent très grand. Il s'agit, en général, de maximiser (problème de maximisation) ou de minimiser (problème de minimisation) une fonction objectif sous certaines contraintes. Le but est de trouver une solution optimale dans un temps d'exécution raisonnable. Néanmoins, ce but est loin d'être concrétisé pour plusieurs problèmes vu la leurs complexités grandissantes.

La théorie de la NP-complétude a permis de classifier les problèmes d'optimisation selon leurs complexités et elle fournit des informations pertinentes sur le genre de méthodes à choisir en fonction de la difficulté intrinsèque des problèmes. Lorsqu'une solution est associée à une seule valeur, on parle de problèmes mono-objectifs, et lorsqu'elle est associée à plusieurs valeurs, on parle de problèmes multi-objectifs (ou multi-critères). Il faut noter que, l'optimisation d'un problème multi-objectif est souvent plus difficile que l'optimisation des problèmes mono-objectifs. En effet, L'optimisation multi-objectif permet de modéliser des problèmes réels faisant concourir de nombreux critères (souvent conflictuels) et contraintes. Dans ce contexte, la solution optimale recherchée n'est plus un simple point, mais un ensemble de bons compromis satisfaisant toutes les contraintes. Bien que les problèmes d'optimisation combinatoire soient souvent faciles à définir, ils sont généralement pénibles à résoudre. En effet, la plupart de ces problèmes appartiennent à la classe des problèmes NP-difficiles et ne possèdent pas encore de solutions algorithmiques efficaces et acceptables pour toutes les données.

Les techniques pour résoudre les problèmes mathématiques dépendent de la nature de la fonction objectif de l'ensemble contraint. Les sous-domaines majeurs suivants existent :

La programmation linéaire étudie les cas où la fonction objectif et les contraintes sont linéaires.

La programmation linéaire en nombres entiers étudie les programmes linéaires dans lesquels certaines ou toutes les variables sont contraintes à prendre des valeurs entières.

La programmation quadratique concerne les problèmes dont la fonction objectif contient des termes quadratiques (ex :$f(x)= x^2+x$), tout en conservant les contraintes linaires .

La programmation non-linéaire étudie le cas général dans lequel la fonction objectif ou les contraintes (ou les deux) contiennent des parties non-linéaires.

La programmation stochastique concerne les problèmes avec des contraintes dépendant de variables aléatoires.

La programmation dynamique ce type de méthodes est utilisé dans le cas où les le problème est décomposables en petites entités facilement résolubles. Elle n'est utilisable que lorsque la fonction objectif est monotone croissante. De ce fait, la résolution d'un problème en programmation dynamique est basée sur une décomposition du problème en sous-problèmes plus simples. A chaque sous-problème correspond un ensemble d'options, représentant chacune un coût en terme de fonction objectif. Un ensemble de choix doit donc être effectué pour les différents sous-problèmes dans le but d'arriver à une solution optimale.

1.2 La théorie de la complexité

La théorie de la complexité consiste à estimer la difficulté ou la complexité d'une solution algorithmique d'un problème posé de façon mathématique. Elle se concentre sur les problèmes de décision qui posent la question de l'existence d'une solution comme le problème de satisfiabilité booléenne. La théorie de la complexité repose sur la notion de classes de complexité qui permettent de classer les problèmes en fonction de la complexité des algorithmes utilisés pour les résoudre. Parmi les classes les plus courantes, on distingue:

- La *classe P* (Polynomial time) qui englobe les problèmes pour lesquels il existe un algorithme déterministe de résolution en temps polynomial,
- La *classe NP* (Nondeterministic Polynomial time) qui contient des problèmes de décision pour lesquels la réponse *oui* peut être décidée par un algorithme non-déterministe en un temps polynomial par rapport à la taille de l'instance. Les problèmes NP-complets sont définis comme suit:

Définition 1.1 (Problème NP-complet) *Un problème de décision n est NP-complet s'il satisfait les deux conditions suivantes : $n \in NP$, et tout problème NP se réduit à n en temps polynomial.*

L'une des questions ouvertes les plus fondamentales en informatique théorique est vraisemblablement la question si « P=NP ? » (Figure 1.1). Ceci revient à trouver un algorithme polynomial pouvant résoudre un problème NP-complet. Trouver un tel algorithme, pour un seul problème appartenant à la classe NP-complet, signifierait que tous les problèmes de cette classe pourraient être résolus en temps polynomial (voir Définition 1.1) et en conséquence, que P=NP.

Cependant, il est commun de penser que P≠NP, mais aucune preuve n'a encore été trouvée jusqu'à aujourd'hui.

Il est important de préciser que tous les problèmes d'optimisation ne peuvent pas être classifiés comme problèmes NP-complets, puisqu'ils ne sont pas tous des problèmes de décision, même si pour chaque problème d'optimisation on peut définir un problème de décision qui a une complexité équivalente.

Définition 1.2 (Problème NP-difficile) *Un problème P quelconque (de décision ou non) est NP-difficile si et seulement s'il existe un problème NP-complet* P′ *qui est réductible à lui polynomialement.*

La définition d'un problème NP-difficile est donc moins étroite que celle de la NPcomplétude. De cette définition on peut observer que pour montrer qu'un problème d'optimisation est NP-difficile, il suffit de montrer que le problème de décision associé à lui est NP-complet. De cette façon un grand nombre de problèmes d'optimisation ont été prouvés NP-difficiles. C'est notamment le cas des problèmes du Voyageur de Commerce, de Partitionnement de Graphes et d'Affectation Quadratique.

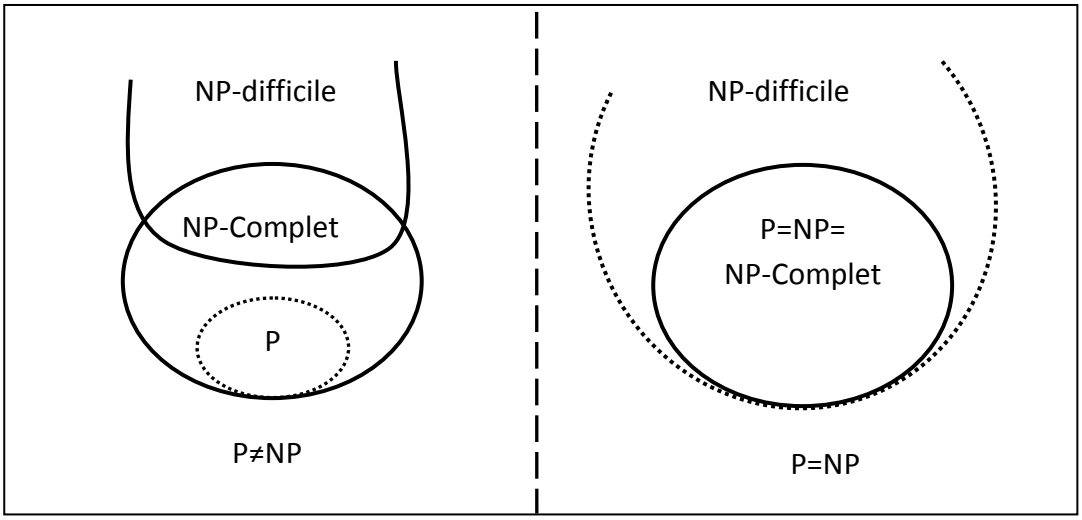


Figure.1.1 – classe P, NP, NP-complet, NP-difficile.

1.3 Formulation mathématique des problèmes d'optimisation

Les problèmes d'optimisation combinatoire peuvent être formulés comme suit :

Définition 1.3 (Problèmes mono-objectifs) : *Un problème d'optimisation est généralement formulé comme un problème de minimisation ou de maximisation, et écrit sous la forme suivant:*

$$\begin{cases} \min_x f(x), \text{ Tel que,} \\ g_i(x) \leq 0 \ , \ i = 1, \ldots, m \ , \\ h_j(x) = 0 \ , \ j = 1, \ldots, p \ , \\ x \in S \subset R^n, \end{cases} \qquad (1.1)$$

Où f est la fonction (scalaire) à minimiser, appelée fonction coût ou fonction objectif, x représente le vecteur des variables d'optimisation, g_i sont les contraintes d'inégalité et h_j les contraintes d'égalité, et S est l'espace des variables (appelé aussi espace de recherche). S indique quel type de variables sont considérées : réelles, entières, mixtes (réelles et entières dans un même problème), discrètes, continues, bornées, etc.

Un point x_A est appelé un point admissible si $x_A \in S$ et si les contraintes d'optimisation sont satisfaites : $gi(xA) \leq 0$, $i = 1, \ldots, m$ et $hj(xA) = 0$, $j = 1, \ldots, p$. La solution de (1.1) est l'ensemble des optima $\{x^*\}$.

x^* est un minimum global de f si et seulement si $f(x^*) \leq f(x) \ \forall x \in S$, et x^* est un minimum local de f si et seulement si $f(x^*) \leq f(x) \ \forall x \in S / ||x - x^*|| \leq \varepsilon, \varepsilon > 0$. La Figure 1.2 présente un exemple d'une fonction à une variable, avec des minima locaux et un minimum global. Parmi les minima locaux, celui qui possède la plus petite valeur de f est le minimum global. Par ailleurs, une fonction multimodale présente plusieurs minima (locaux et globaux), et une fonction unimodale n'a qu'un minimum, le minimum global. La Figure 1.3 montre une fonction multimodale à deux variables.

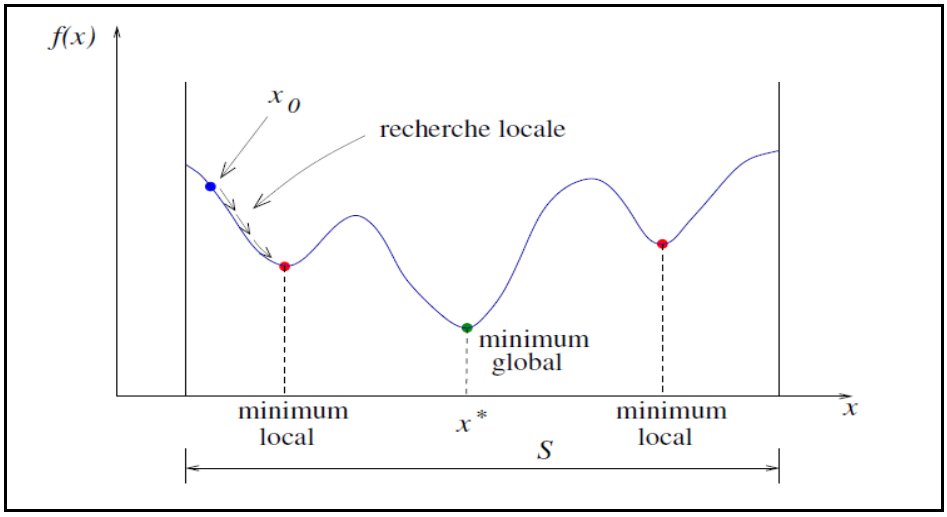


Figure 1.2 – Minima locaux et minimum global d'une fonction à une variable.

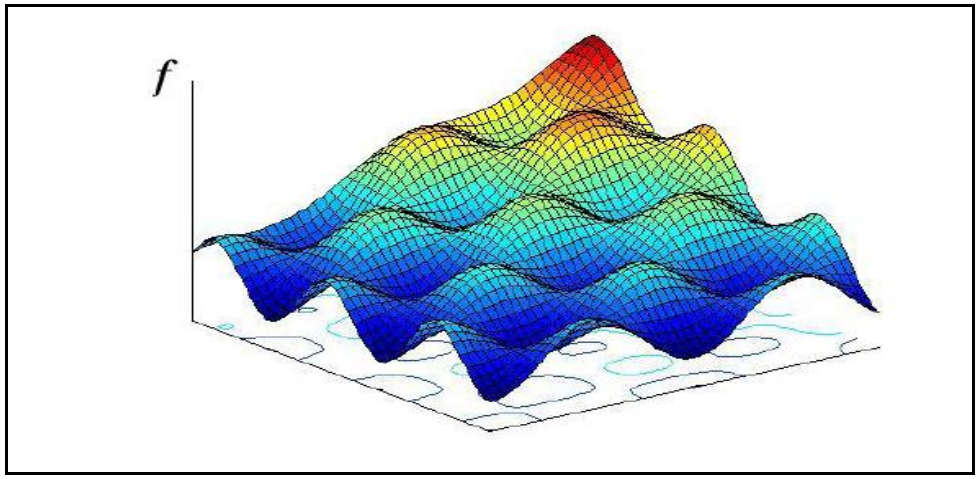


Figure 1.3 – Une fonction multimodale à deux variables.

Définition 1.4 (Problèmes multi-objectifs) : *D'un point de vue mathématique, un problème d'optimisation multi-objectif, se présente, dans le cas où le vecteur* $\vec{f}$ *regroupe k fonctions objectif, de la façon suivante:*

$$\begin{cases} min_x \ \vec{f}(x) \text{ , Tel que,} \\ \vec{g_i}(x) \leq 0 \text{ , } i = 1, \ldots, m \text{ ,} \\ \vec{h_i}(x) = 0 \text{ , } j = 1, \ldots, p \text{ , } \ x \in S \subset R^n, \end{cases} \qquad (1.2)$$

De ce fait, résoudre un problème d'optimisation combinatoire nécessite l'étude de trois points suivants :

– la définition de l'ensemble des solutions réalisables,

– l'expression de l'objectif à optimiser,

– le choix de la méthode d'optimisation (exacte ou approchée) à utiliser.

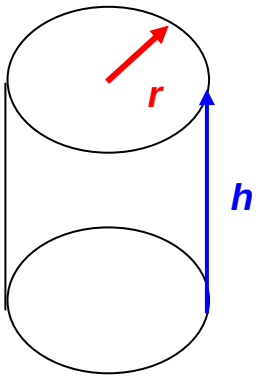
Les deux premiers points relèvent de la modélisation du problème, le troisième de sa résolution.

Exemple 1.

Lait-de-Constantine est une petite entreprise spécialisée dans les produits laitiers. Elle vient de créer une gamme de soupes à la crème qu'elle désire mettre en marché, dans des boîtes de conserve. Mais elle se demande quel devrait être le format à adopter pour ces boîtes afin d'optimiser cette nouvelle gamme.

Le département de la production suggère tout simplement le format de base retrouvé sur le marché, soit des boîtes de 1 L. Mais comme il existe toute une panoplie de dimensions de conserve pouvant satisfaire ce volume fixé, ce département a entrepris une étude sur les coûts de production qui devraient permettre la déduction des dimensions optimales. Ces coûts sont établis comme suit : chaque millilitre de soupe revient à 0,1DA, tandis que les coûts de fabrication de la boîte en tant que tels reviennent à 0,4DA/cm^2 pour les bases de la boîte et à 0,3DA/cm^2 pour la surface latérale de la boîte (incluant le coût de l'étiquette de la compagnie). Trouver les dimensions de la boîte de conserve qui minimisent les coûts. (Note : un cm^3 contient 1 ml.)

Le modèle mathématique



Variables de décision

r = rayon de la boîte de conserve (en cm)

h = hauteur de la boîte de conserve (en cm)

Fonction-objectif :

Min $c(r,h) = 0{,}001\pi r^2 h + 0{,}008\pi r^2 + 0{,}006\pi r h$

Contraintes :

$\pi r^2 h = 1000$

$r > 0, h > 0$

1.4 Notions de base en optimisation mathématique

Définition 1.5 *L'ensemble S est* ***convexe****, si quels que soient les deux points P, Q ∈S, tout le segment PQ ∈ S.*

Définition 1.6 *Une fonction $f(x, y)$ à deux variables est dite* ***convexe*** *sur un ensemble convexe si et seulement si tout segment de droite reliant deux points sur la surface de $f(x, y)$ est toujours au-dessus de cette surface.*

Définition 1.7 *Une fonction $f(x, y)$ à deux variables est dite* ***concave*** *sur un ensemble convexe si et seulement si tout segment de droite reliant deux points sur la surface de $f(x, y)$ est toujours au-dessous de cette surface.*

Définition 1.8 *Un programme mathématique est dit* ***convexe*** *s'il s'agit de la* ***minimisation d'une fonction convexe*** *sur une région réalisable convexe, soit de la* ***maximisation d'une fonction concave*** *sur une région réalisable convexe.*

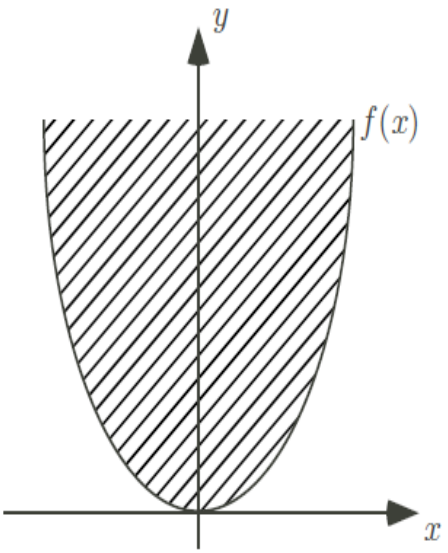


Figure 1.4– exemple de fonction convexe.

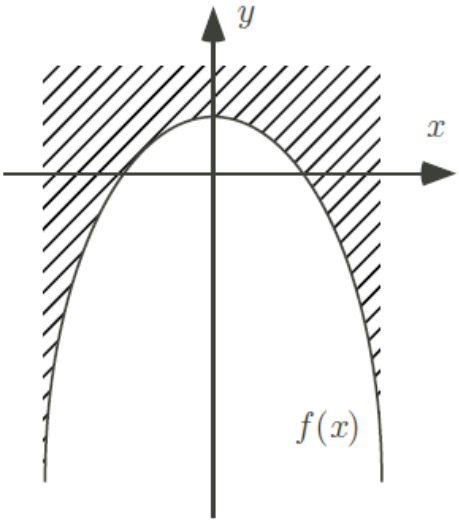


Figure 1.5– exemple de fonction concave.

Définition 1.9 (Voisinage et mouvement) : *Soit une solution s, on dit que s* est une solution voisine de s, si on peut obtenir s* en modifiant légèrement s. On dit aussi qu'on peut passer de s à s* en effectuant un mouvement. Le voisinage V(s) de s est l'ensemble des solutions voisines de s. La fonction $V : S \rightarrow S^2, \forall s \in S, V(s) \subset S$ est associe à chaque point de S un sous-ensemble de S .*

Définition 1.10 (Optimum local) *:* soit S un ensemble convexe, f est une fonction définie sur S. *Une solution $s \in S$ est un optimum local si et seulement si il n'existe pas de solution $s_0 \in V(s)$ dont l'évaluation est de meilleure qualité que s, soit :*

$$\forall\ s_0 \in v(s) \begin{cases} f(s) \leq f(s_0) & \textit{Dans le cas d'un problème de minimisation} \\ f(s) \geq f(s_0) & \textit{Dans le cas d'un problème de maximisation} \end{cases}$$

Avec V (s) l'ensemble des solutions voisines de s.

Définition 1.11 (Optimum global) : *Une solution est un optimum global à un problème d'optimisation s'il n'existe pas d'autres solutions de meilleure qualité. La solution s* ∈ S est un optimum global si et seulement si:*

$$\forall\ s \in S \begin{cases} f(s^*) \leq f(s) & \textit{Dans le cas d'un problème de minimisation} \\ f(s^*) \geq f(s) & \textit{Dans le cas d'un problème de maximisation} \end{cases}$$

Donc, l'optimum global est la solution s* qui vérifie la propriété précédente pour toutes les structures de voisinage du problème. La figure 1.6 schématise la courbe d'une fonction d'évaluation en faisant apparaître l'optimum global (local) dans le cas d'un problème de minimisation.

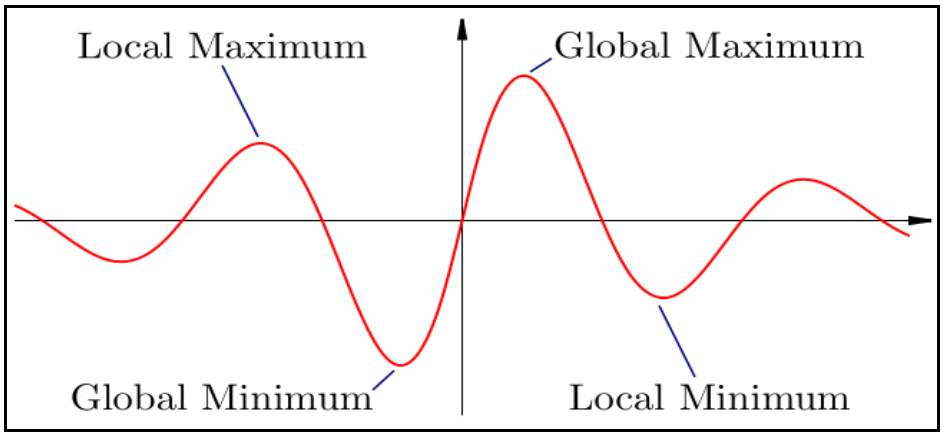


Figure 1.6 – L'optimum global et local.

Du point de vue pratique, les définitions des minima ne sont pas très utiles, pour s'assurer que s0 est bien un minimum local, il faudrait parcourir tous les s dans le voisinage de s0 ce qui est trop coûteux.

1.5 Les méthodes de l'optimisation combinatoire

Les méthodes de l'optimisation combinatoire peuvent être classées en méthodes heuristiques et méthodes exactes (figure 1.7) :

Les algorithmes exacts sont utilisés pour trouver au moins une solution optimale d'un problème. Les algorithmes exacts les plus réussis dans la littérature appartiennent aux paradigmes de la programmation dynamique, de la programmation linéaire en nombres entiers, ou des méthodes de recherche arborescente (Branch & Bound). Généralement, pour un problème donné, ces algorithmes procèdent par énumération de toutes les solutions possibles afin de trouver la meilleure solution. Lorsque les ampleurs du problème deviennent importantes, une énumération explicite s'avère irréalisable. Pour cela, On a recours aux procédures classiques par séparation et évaluation ("Branch and Bound"). Ces algorithmes réalisent une construction partielle d'un arbre de recherche des

solutions en essayant de trouver une solution exacte pour un problème donné. Bien que les méthodes exactes trouvent des solutions optimales, leur grand inconvénient est l'explosion combinatoire ce qui rend leur utilisation pratique difficile.

D'un autre coté, Une méthode heuristique ou approchée est une méthode d'optimisation qui a pour but de trouver une solution réalisable de la fonction objectif en un temps raisonnable, mais sans garantie d'optimalité. L'avantage principale de ces méthodes est qu'elles peuvent s'appliquer à n'importe quelle classe de problèmes, faciles ou très difficiles, bien ou mal formulés, avec ou sans contrainte. En plus, elles ne nécessitent pas une spécification mathématique du problème. D'un autre côté, Les algorithmes heuristiques tels que les algorithmes de recuit simulé, les algorithmes tabous et les algorithmes génétiques ont démontré leurs robustesses et efficacités face à plusieurs problèmes d'optimisation combinatoires.

Il faut noter qu'il n'existe pas une méthode sésame qui peut résoudre tous les problèmes, parfois c'est intéressant de faire hybrider plusieurs algorithmes de résolution au sein d'un seul algorithme afin de résoudre des problèmes très compliqués. Indépendamment de l'approche, une bonne méthode d'optimisation doit considérer les points suivants :

• **robustesse :** la méthode devrait fonctionner pour une large classe de problèmes, indépendamment des paramètres et des valeurs initiales,

• **efficacité:** elle ne devrait pas être prohibitivement chère en temps CPU et en mémoire vive,

• **précision:** le résultat numérique devrait bien approcher la vraie solution du problème de minimisation

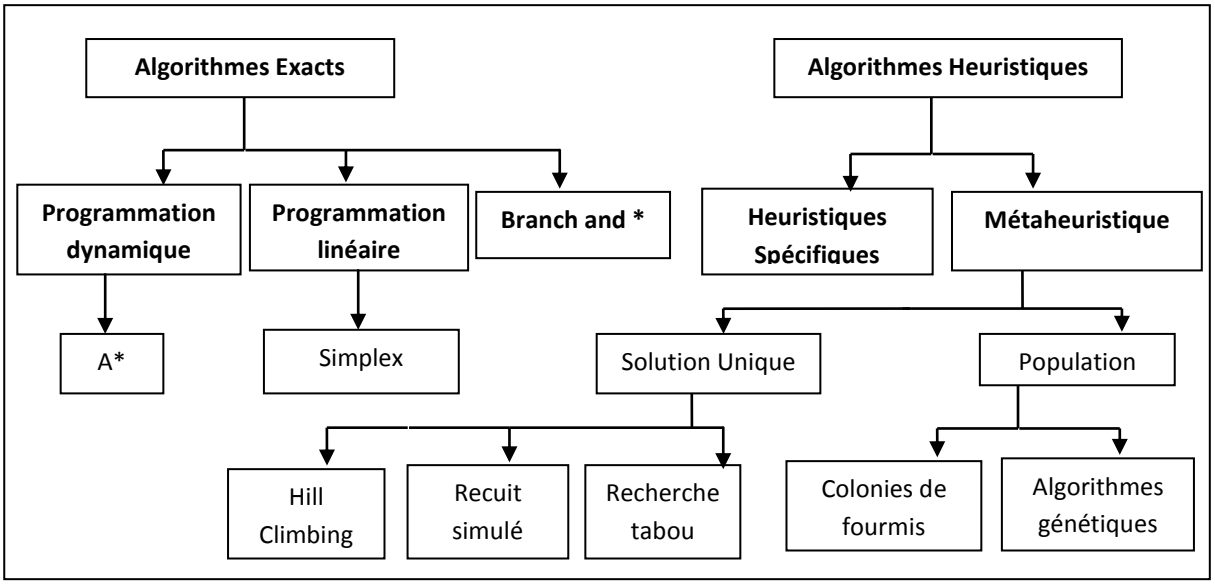


Figure 1.7 – Taxonomie des méthodes d'optimisation combinatoire.

Chapitre 2 : Optimisation non-linéaire sans contraintes

2.1 Introduction

De nombreuses applications industrielles de l'optimisation impliquent une modélisation plus proche de la physique. En optimisation de la conception par exemple, on optimise des méta-modèles ou surfaces de réponse issus d'un modèle statistique qui ne sont pas linéaires. De même l'optimisation topologique ou l'optimisation de forme fait largement appel à des modèles non linéaires et de plus de très grande taille dans cet exemple. Formellement un programme mathématique non linéaire sans contraintes s'écrit :

> **Étant donné :** une fonction $f : A \to \mathbb{R}$ d'un ensemble *A* dans l'ensemble des nombre réels
> **Rechercher :** un élément x_0 de *A* tel que $f(x_0) \geq f(x)$pour tous les *x* en *A*
> (« maximisation ») ou tel que $f(x_0) \leq f(x)$pour tous les *x* en *A* (« minimisation »).

x vecteur de nombre réels représentant les variables de décision

Exemple : Déterminer les valeurs extrémales de $f(x, y) = x^3 + y^3 - 6xy$

Les techniques pour résoudre les problèmes mathématiques et la performance des algorithmes d'optimisation dépendent de la nature de la fonction objectif .

Pour les fonctions dérivables deux fois, des problèmes sans contraintes peuvent être résolus en trouvant les points où le gradient de la fonction est 0 et en utilisant la matrice Hessienne pour classer la nature de point. Si le Hessien est défini positif, le point est un minimum local ; s'il est un défini négatif, un maximum local.

Si la fonction est convexe sur l'ensemble des solutions admissibles alors tout minimum local est aussi un minimum global.

2.2 Rappel mathématiques

Rappel mathématiques 1: le gradient d'une fonction *f* à *n* variables est un vecteur n-dimensionnel contenant toutes les dérivées partielles de f par rapport à chacune de ses variables

$$\nabla f(x1, x2, ..., xn) = (\frac{\partial f}{x1}, \frac{\partial f}{x2}, ..., \frac{\partial f}{xn})$$

Si $\nabla f(x1, x2, ..., xn) = 0$ pour le point (x_1', x_2',.., x_n') alors ce point est un minimum (maximum) local. Ce point est appelé également point stationnaire.

Rappel mathématiques 2: Le **Hessien** est une façon d'écrire les dérivées d'un gradient à l'aide d'une matrice symétrique. pour une fonction f(x1,x2,…,xn), le Hessein est calculé en utilisant la formule suivante :

$$H(x1, , x2, ..., xn) = \nabla^2 f(x1, , x2, ..., xn) = \left(\frac{\partial f}{\partial xi \partial xj} \right) \text{ pour i=1..n,j=1..n.}$$

$$\mathrm{H(x)} = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\ \vdots & \vdots & & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix}$$

Pour le cas d'une fonction à deux variables : Soit $f(x, y)$ une fonction deux fois dérivable, alors le Hessien est :

$$H(x, y) = \nabla^2 f(x, y) = \begin{bmatrix} f_{xx}^{(2)}(x, y) & f_{xy}^{(2)}(x, y) \\ f_{yx}^{(2)}(x, y) & f_{yy}^{(2)}(x, y) \end{bmatrix}.$$

On a toujours $f_{xy}^{(2)}(x, y) = f_{yx}^{(2)}(x, y)$.

Exemple $f(x, y) = 5x^2 + 3y^2 + 8xy - 3x + 7y$ où $(x, y) \in \Re^2$.

Calculer le gradient de f : $f_x'(x, y) = 10x + 8y - 3$; $f_y'(x, y) = 6y + 8x + 7$.

Calculer le Hessien de f : $\frac{\partial}{\partial x} f_x'(x, y) = f_{xx}^{(2)}(x, y) = 10$; $\frac{\partial}{\partial y} f_x'(x, y) = f_{xy}^{(2)}(x, y) = 8$;

$\frac{\partial}{\partial x} f_y'(x, y) = f_{yx}^{(2)}(x, y) = 8$; $\frac{\partial}{\partial y} f_y'(x, y) = f_{yy}^{(2)}(x, y) = 6$.

Le Hessien est $H(x,y)=\begin{bmatrix}10 & 8\\ 8 & 6\end{bmatrix}$ et dans cet exemple, il est le même (constant) en tout point du domaine de $f(x,y)$, ce qui n'est pas toujours le cas.

- Soit $A=\begin{bmatrix}a & b\\ c & d\end{bmatrix}$, le **déterminant** de *A* est donné par $|A|=\det(A)=ad-bc$.
- Une matrice A est **symétrique** si $b = c$; le Hessien est toujours symétrique.

2.3 Définitions

Soit A une matrice n*n symétrique et X un vecteur de R^n X^T est le transposé de X.

- *A* est **définie positive** si et seulement si $X^T.A.X > 0 \ \forall\ X \in S$

  Pour le cas d'une fonction f à deux variables il faut vérifier seulement $\det(A)>0, a>0, d>0.$
- *A* est **définie négative** si et seulement si $X^T.A.X < 0$

  Pour le cas d'une fonction f à deux variables il faut vérifier seulement $\det(A)>0, a<0, d<0.$
- *A* est **semi-définie positive** si et seulement si $X^T.A.X \geq 0$

  Pour le cas d'une fonction f à deux variables il faut vérifier seulement $\det(A)\geq 0, a\geq 0, d\geq 0$
- *A* est **semi-définie négative** si et seulement si $X^T.A.X \leq 0$

  Pour le cas d'une fonction f à deux variables il faut vérifier seulement $\det(A)\geq 0, a\leq 0, d\leq 0$
- Note : $\det(H(x,y))\geq 0$ signifie que le déterminant $\det(H(x,y))$ peut s'annuler pour une ou plusieurs valeurs de (*x*,*y*).

Exemples

- $H_1=\begin{bmatrix}10 & 8\\ 8 & 6\end{bmatrix}\rightarrow \det(H_1)=10\cdot 6-8\cdot 8=-4<0$; H_1 est non définie.
- $H_2=\begin{bmatrix}-2 & 3\\ 3 & -6\end{bmatrix}\rightarrow \begin{cases}\det(H_2)=(-2)(-6)-3\cdot 3=3>0\\ \text{diagonale négative}: a=-2;\ d=-6.\end{cases}$ H_2 est définie négative.
- $H_3=\begin{bmatrix}10 & 10\\ 10 & 10\end{bmatrix}\rightarrow \begin{cases}\det(H_3)=10\cdot 10-10\cdot 10=0\\ \text{diagonale positive}: a=10,\ d=10\end{cases}$ H_3 est semi–définie positive.
- $H_4(x,y)=\begin{bmatrix}x^2 & 0\\ 0 & y^2\end{bmatrix}\rightarrow \begin{cases}\det(H_4(x,y))=x^2y^2\geq 0\\ a=x^2\geq 0;\ b=x^2\geq 0\end{cases}$ H_4 est semi–définie positive.

- $H_5(x,y)=\begin{bmatrix}0 & 0\\ 0 & 0\end{bmatrix}\rightarrow\begin{cases}\det(H_5(x,y))=0\\ a=b=0\end{cases}$

- H_5 est à la fois semi–définie positive et semi-définie négative; c'est en fait le Hessien d'une fonction linéaire.

Théorème : Soit $X^*=(x^*,y^*)$ un point stationnaire de la fonction $f(x,y)$.

- **Si le Hessien de *f* est défini positif en X^*, alors X^* est un *minimum local*.**
- **Si le Hessien de *f* est défini négatif en X^*, alors X^* est un *maximum local*.**
- **Si $\det(H(X^*))<0$, alors X^* est un *point singulier* .**
- **Si $\det(H(X^*))=0$, il faut utiliser d'autres techniques basées sur les dérivées d'ordre supérieur pour pouvoir se prononcer.**

Exemple $f(x,y)=2x^2-8xy^2-\frac{16}{3}y^3+8y^2-8,$ où $(x,y)\in\Re^2$.

a) Donner le **gradient** $\nabla f(x,y)$ et le **Hessien** $H(x,y)$.

b) Identifier la nature des points stationnaires de la fonction $f(x,y)$.

a) $\nabla f(x,y)=(4x-8y^2;-16xy-16y^2+16y)$; $H(x,y)=\begin{bmatrix}4 & -16y\\ -16y & -16x-32y+16\end{bmatrix}$.

b) $\nabla f(x,y)=(0;0)\Leftrightarrow x=2y^2$ et $xy+y^2-y=0=2y^3+y^2-y=y(2y-1)(y+1)$.

Les points stationnaires sont : (0, 0) (1/2, 1/2) (2, -1).

$H(0,0)=\begin{bmatrix}4 & 0\\ 0 & 16\end{bmatrix}$; $\det(H(0,0))>0$; $4>0$ $\Rightarrow$ (0,0) est un minimum local.

$H(\frac{1}{2},\frac{1}{2})=\begin{bmatrix}4 & -8\\ -8 & -8\end{bmatrix}$; $\det(H(\frac{1}{2},\frac{1}{2}))<0$ $\Rightarrow$ $(\frac{1}{2},\frac{1}{2})$ est un point singulier.

$H(2,-1)=\begin{bmatrix}4 & 16\\ 16 & 16\end{bmatrix}$; $\det(H(2,-1))<0$; $\Rightarrow$ $(2,-1)$ est un point singulier.

Exemple Un problème de marketing

La fonction de profit est $f(x,y)=300x+400y-x^2-2y^2-2xy$

et le seul point stationnaire est $(x^*,y^*)=(100,50)$.

Quelle est la nature de ce point ?

Puisque $\nabla f(x,y) = (300-2x-2y; 400-4y-2x)$, le Hessien est $H(x,y) = \begin{bmatrix} -2 & -2 \\ -2 & -4 \end{bmatrix}$.

$\det(H(100,50)) = (-2)(-4)-(-2)(-2) = 4 > 0$ avec $a = -2$ et $b = -4$. $H(x^*, y^*)$ est *défini négatif*. $(x^*, y^*) = (100, 50)$ est donc un *maximum local*.

Pour une fonction convexe ou concave, la condition du premier ordre (gradient nul) est nécessaire et suffisante pour caractériser la nature d'un point stationnaire.

Théorème : Soit $X^* = (x^*, y^*)$ un point stationnaire de la fonction $f(x,y)$.

- **Si $f(x,y)$ est convexe, alors X^* est un minimum absolu (global).**
- **Si $f(x,y)$ est concave, alors X^* est un maximum absolu (global).**

Si la fonction $f(x,y)$ est deux fois dérivable, on peut la caractériser par le Hessien :

Théorème : Soit $f(x,y)$ dérivable deux fois sur un ensemble *E*.

$f(x,y)$ est convexe si et seulement si $H(x,y)$ est semi-défini positif $\forall (x,y) \in E$.

$f(x,y)$ est concave si et seulement si $H(x,y)$ est semi-défini négatif $\forall (x,y) \in E$.

Si $\det(H(x,y)) > 0, \ \forall (x,y) \in E$ (H est défini), alors on dira que la fonction $f(x,y)$ est *strictement* convexe ou *strictement* concave.

Exemple $f(x,y) = x\ln(x) + y\ln(y), \quad x > 0, y > 0$.

$\nabla f(x,y) = (1+\ln(x), \quad 1+\ln(y)), \quad x > 0, y > 0$.

$H(x,y) = \begin{bmatrix} 1/x & 0 \\ 0 & 1/y \end{bmatrix}, \quad x > 0, y > 0.$ $\det(H(x,y)) = (1/xy) > 0;$ et $1/x > 0, \forall (x,y).$

$H(x,y)$ est défini positif $\forall (x,y)$, donc $f(x,y)$ est *strictement* **convexe.**

Exemple $f(x,y) = 4x^2 + 2y^2 - 8x - 2y + 1$.

Identifier la nature des points stationnaires de la fonction $f(x,y)$.

$$\nabla f(x,y) = (8x-8; \quad 4y-2); \ \nabla f(x^*, y^*) = 0 \Leftrightarrow \begin{cases} 8x^* - 8 = 0 \Leftrightarrow 8x^* = 8 \Rightarrow x^* = 1 \\ 4y^* - 2 = 0 \Leftrightarrow 4y^* = 2 \Rightarrow y^* = 1/2 \end{cases}$$

(1 , 1/2) est le seul point stationnaire de $f(x, y)$.

$H(x,y) = \begin{bmatrix} 8 & 0 \\ 0 & 4 \end{bmatrix}$. $\det(H(x^*, y^*)) = 8 \cdot 4 - 0 \cdot 0 = 32 > 0$ avec $a = 8$, $\forall(x, y)$.

Le Hessien $H(X^*)$ est défini positif $\forall(x, y)$, donc $f(x, y)$ est *strictement* **convexe.**

Par conséquent, $(x^*, y^*) = (1, 1/2)$ est un *minimum absolu*.

Exemple **Problème de marketing** (suite)

On a déjà vu que l'unique point stationnaire $(x^*, y^*) = (100, 50)$ est un maximum local. Mais il est possible d'avoir une information plus complète sur ce point stationnaire, puisque le Hessien $H(x,y) = \begin{bmatrix} -2 & -2 \\ -2 & -4 \end{bmatrix}$ est le même partout.

$\det(H) = (-2)(-4) - (-2)(-2) = 4 > 0$ avec $a = -2$ et $b = -4$. $H(x, y)$ est *défini négatif* pour tout point (x, y) et donc $f(x, y)$ est *strictement* **concave**. Par conséquent, $(x^*, y^*) = (100, 50)$ est un *maximum absolu*.

Propriétés des fonctions convexes et concaves

- Si f_1 et f_2 sont convexes (concaves), alors $f_1 + f_2$ est aussi convexe (concave).
- Si *f* est convexe (concave), alors $g(X) = k \cdot f(X)$ est convexe (concave), $\forall k > 0$.
- Si *f* est convexe (concave) alors $-f$ est concave (convexe).
- $f(x, y) = a + bx + cy$ est dite une **fonction affine** sur $\Re^2$; elle est à la fois convexe et concave parce que son Hessien est nul et qu'il est alors à la fois semi-défini positif et semi-défini négatif.

Références

1. Initiation aux techniques classiques de l'optimisation, Michèle Breton et Alain Haurie, Éditeur Modulo.
2. Calcul 1 : Théorie, exemples, problèmes, Gilles Ouellet, Éditions Le Griffon d'Argile (Révision sur les dérivées).
3. Initiation aux techniques classiques de l'optimisation - solutions, Michèle Breton et Alain Haurie, Éditeur Modulo.

Chapitre 3 : Optimisation non-linéaire avec contraintes

3.1 Introduction

Nous avons traité des techniques d'optimisation d'une fonction non linaire d'une ou de plusieurs variables sans contraintes. Cependant, Dans les situations plus réalistes, il faut également considérer des restrictions communes sur les variables données par des **contraintes** du modèle étudié. On distingue de type de problème d'optimisation avec contraintes.

Nous ne pouvons pas appliquer les règles vues dans le cas des problèmes sans contraintes. Examinons cet exemple :

Min $f(x)=x^2$ sous contrainte $x \geq 1$, la solution est 1 alors que $f'(1) = 2 \neq 0$

3.2 Optimisation avec contraintes d'égalité

De façon générale, les modèles étudiés avec contraintes d'égalité sont de la forme suivante où, sans perte de généralité, le côté droit de l'égalité peut toujours prendre une valeur nulle :

$$\max\ f(X)\ \text{ s.c. } \left.\begin{array}{l} h_1(X) = 0 \\ \vdots \\ h_m(X) = 0 \end{array}\right\} \ m \text{ contraintes d'égalité.}$$

Définition : Les concepts d'optimum local et absolu sont encore les mêmes, à l'exception maintenant que tout point X_0 considéré doit être **admissible**, c'est-à-dire qu'il doit satisfaire toutes les contraintes $h_i(X_0) = 0,\ \forall i = 1,\ldots m$.

Nous verrons, dans les sections suivantes, un survol des méthodes d'optimisation utilisées pour résoudre les problèmes sous contraintes d'égalité comportant seulement deux variables et une seule contrainte.

3.2.1 Résolution par substitution

Lorsqu'il y a une variable de plus que le nombre de contraintes et que ces contraintes sont simples (par exemple, linéaires), on peut résoudre le modèle par substitutions successives. En fait, on se ramène à un problème *d'une fonction d'une seule variable*. On peut ainsi utiliser les techniques d'optimisation non linéaire sans contraintes.

Exemple 3.1

Résoudre le problème $\max f(x,y) = -x^2 - y^2$ s.c. $3x + 4y - 25 = 0$.

Par la contrainte, on isole $y = \dfrac{25-3x}{4}$ que l'on substitue dans $f(x,y)$:

$f(x,y) = -x^2 - \left(\dfrac{25-3x}{4}\right)^2 = g(x)$. $g(x)$ est donc une fonction à une seule variable dont on trouve le point stationnaire : $g'(x^*) = 0 \Rightarrow -2x_1^* - 2\left(\dfrac{-3}{4}\right)\left(\dfrac{25-3x^*}{4}\right) = 0 \quad \Rightarrow x^* = 3$.

La dérivée seconde étant toujours négative ($g^{(2)}(x) = -25/8, \forall x$), la fonction $g(x)$ est strictement concave et le point stationnaire trouvé est un maximum absolu.

L'évaluation de y est immédiate : $y^* = \dfrac{25-3(3)}{4} = 4$, d'où $(x^*, y^*) = (3, 4)$ est le point maximisant l'objectif du problème original, sous la contrainte imposée.

3.2.2 Conditions d'optimum du premier ordre

Considérons le problème suivant à deux variables : max $f(x,y)$ s.c. $h(x,y) = 0$.

Théorème 4.1 : Conditions (nécessaires) d'optimum du premier ordre

Soit $f(x,y)$ **et** $h(x,y)$ **deux fonctions dérivables; si** $f(x,y)$ **admet un optimum local en** $X^* = (x^*, y^*)$ **sous l'hypothèse que** $\nabla h(X^*) \neq (0,0)$**, alors :**

1. $h(X^*) = 0$ **;**
2. **il existe un** $\lambda^* \in \Re$ **tel que**

$$\left.\begin{aligned} f'_x(X^*) - \lambda^* h'_x(X^*) = 0 \\ f'_y(X^*) - \lambda^* h'_y(X^*) = 0 \end{aligned}\right\} \Leftrightarrow \nabla f(X^*) - \lambda^* \nabla h(X^*) = (0,0).$$

λ^* est le ***multiplicateur de Lagrange***

La première condition indique que le point considéré X^* doit être admissible, c'est-à-dire qu'il doit satisfaire la contrainte imposée, alors que la seconde généralise la condition sur le gradient nul, en tenant compte de cette même contrainte. Un point X^* qui satisfait ces conditions d'optimum du premier ordre s'appelle un **point extrémal**.

Si l'une des dérivées partielles $h'_x(X^*)$ ou $h'_y(X^*)$ est nulle, on utilisera l'autre ratio pour calculer λ^*. Ceci est possible car, par hypothèse, on a $\nabla h(X^*) = (h'_x(X^*), h'_y(X^*)) \neq (0,0)$.

Retour sur l'exemple 3.1

Considérons à nouveau $\max f(x,y) = -x^2 - y^2$ s.c. $3x + 4y = 25$.

Vérifions le calcul de λ^* au point $(x^*, y^*) = (3, 4)$:

$$\lambda^* = \frac{f'_x(X^*)}{h'_x(X^*)} = -\frac{2x}{3} = -\frac{2(3)}{3} = -2 \text{ et } \lambda^* = \frac{f'_y(X^*)}{h'_y(X^*)} = -\frac{y}{2} = -\frac{4}{2} = -2.$$

Ce résultat vérifie aussi **que** $\nabla f(x^*, y^*) - \lambda^* \nabla h(x^*, y^*) = (-6, -8) - (-2)(3, 4) = (0, 0)$**.**

3.2.3 La méthode du Lagrangien

La méthode que nous étudions dans cette séance est, en fait, une généralisation de la règle du multiplicateur de Lagrange pour résoudre des problèmes d'optimisation avec des contraintes d'égalité. Nous examinerons le cas d'une seule contrainte à deux variables $X = (x, y)$.

Définition :

Soit le problème : $\max\ f(X)$ s.c. $h(X) = 0$.

On appelle **Lagrangien** associé à ce problème la fonction :

$$L(x, y, \lambda) = L(x, y, \lambda) = f(x, y) - \lambda h(x, y) \ .$$

Le Lagrangien permet d'introduire la contrainte dans la fonction-objectif avec une certaine pénalité λ. On se retrouve ainsi à maximiser une *fonction à trois variables sans contrainte.*

Si, maintenant, on cherche les points stationnaires (x^*, y^*, λ^*) du Lagrangien $L(x, y, \lambda)$, cela veut dire que nous cherchons (x^*, y^*, λ^*) tel que $\nabla L(x^*, y^*, \lambda^*) = (0, 0, 0)$.

Nous déduisons alors le système suivant comportant trois équations:

$$\begin{aligned} L'_x(x^*, y^*, \lambda^*) = \quad & f'_x(x^*, y^*) - \lambda^* h'_x(x^*, y^*) = 0 \\ L'_y(x^*, y^*, \lambda^*) = \quad & f'_y(x^*, y^*) - \lambda^* h'_y(x^*, y^*) = 0 \\ L'_\lambda(x^*, y^*, \lambda^*) = \quad & -h(x^*, y^*) = 0 \end{aligned}$$

Dans les deux premières équations, on reconnaît les conditions d'optimum du premier ordre : $\nabla f(x^*, y^*) - \lambda^* \nabla h(x,^* y^*) = (0,0)$**, alors que la troisième est équivalente à la contrainte de départ imposée par le modèle :** $h(x, y) = 0$**, au point** (x^*, y^*) **(point admissible).**

La **méthode du Lagrangien** transforme donc un problème d'optimisation d'un modèle avec contraintes d'égalité à celui de recherche des points stationnaires d'une fonction sans contrainte. De plus, puisque $h(x^*, y^*) = 0$ pour tout point optimal (x^*, y^*), la valeur du Lagrangien est la même que celle de l'objectif en ce point :

$$L(x^*, y^*, \lambda^*) = f(x^*, y^*) - \lambda^* h(x^*, y^*) = f(x^*, y^*).$$

Remarque pour un problème de minimisation remplacer – par +.

Remarque sur la nature du point stationnaire du Lagrangien

Il faut faire attention car il n'est pas certain que (x^*, y^*) soit un maximum (sous contraintes) pour l'objectif. Pour les contraintes d'égalité, le seul cas facile à résoudre au complet est celui

impliquant des contraintes linéaires (affines) parce qu'elles définissent automatiquement un domaine convexe. On peut alors vérifier la convexité/concavité de la fonction-objectif. En pratique, on choisit généralement de construire des modèles mathématiques que l'on est capable de résoudre !

Dans les autres cas, il faut avoir recours à des outils mathématiques plus complexes, par exemple, les conditions du second ordre qui utilisent la notion de **Hessien bordé**

Retour sur l'exemple 3.1

Résoudre par la méthode du Lagrangien :
$\max\ f(X) = -x^2 - y^2$ s.c. $h(X) = 3x + 4y = 25$.

On forme d'abord **le Lagrangien** : $L(x, y, \lambda) = -x^2 - y^2 - \lambda(3x + 4y - 25)$.

On optimise ensuite $L(x, y, \lambda)$, une fonction à **trois** variables pour laquelle on cherche les **points stationnaires** (x^*, y^*, λ^*) tel que $\nabla L(x^*, y^*, \lambda^*) = (0,0,0)$:

$$L'_x(x^*, y^*, \lambda^*) = -2x^* - 3\lambda^* = 0 \qquad \Rightarrow\ x^* = -\frac{3}{2}\lambda^*.$$

$$L'_y(x^*, y^*, \lambda^*) = -2y^* - 4\lambda^* = 0 \qquad \Rightarrow\ y^* = -\frac{4}{2}\lambda^*.$$

$$L'_\lambda(x^*, y^*, \lambda^*) = -(3x^* + 4y^* - 25) = 0.$$

Avec x^* et y^* dans la troisième équation, on a : $-\left(3\left(-\frac{3}{2}\lambda^*\right) + 4\left(-\frac{4}{2}\lambda^*\right) - 25\right) = 0$,

d'où $\frac{9}{2}\lambda^* + \frac{16}{2}\lambda^* = -25 \qquad \Rightarrow \qquad \lambda^* = -2$.

Reportant à son tour la valeur de λ^* dans les deux premières expressions, on retrouve :
$x^* = -\frac{3}{2}(-2) = 3$ et $y^* = -\frac{4}{2}(-2) = 4$.

Donc $(x^*, y^*, \lambda^*) = (3, 4, -2)$ est le seul point stationnaire du Lagrangien, et par conséquent, $(x^*, y^*) = (3, 4)$ est le seul point extrémal du problème original avec contrainte (trouvé sans avoir à énoncer les conditions d'optimum du premier ordre!).

Le fait que le point $(x^*, y^*, \lambda^*) = (3, 4, -2)$ soit un point stationnaire du Lagrangien ne garantit pas qu'il s'agisse d'un optimum. Il faut vérifier sa **nature**. Puisque la contrainte $h(x, y)$ est linéaire, le domaine est convexe. Il suffit alors de vérifier si la fonction-objectif est convexe/concave par l'analyse du signe de son Hessien. Dans le cas présent, le hessien est le suivant :

$$H(x, y) = \begin{bmatrix} -2 & 0 \\ 0 & -2 \end{bmatrix}$$

Le hessien est donc toujours défini négatif et la fonction-objectif $f(x, y)$ est strictement concave.
Par conséquent, le point stationnaire du Lagrangien correspond bien au maximum absolu du modèle sous contrainte.

3.2.4 Interprétation de λ^*

Considérons le problème d'optimisation suivant, où le côté droit de la contrainte d'égalité est exprimé par la quantité b : $\max \quad f(x,y) \quad$ s.c. $\quad h(x,y)=b$.

Le Lagrangien associé à ce problème est donné par $L(x,y,\lambda)=f(x,y)-\lambda\,(h(x,y)-b)$. À l'optimalité, nous avons déjà mentionné que $L(x^*,y^*,\lambda^*)=f(x^*,y^*)$. Ainsi,

$$L(x^*,y^*,\lambda^*)=f(x^*,y^*)-\lambda^*\,h(x^*,y^*)+\lambda^*\,b.$$

Le coefficient de *b* est λ^*. Ainsi, ce multiplicateur de Lagrange λ^* représente l'effet sur la valeur optimale de la fonction objectif $f(x^*,y^*)$ si l'on modifie d'une unité la valeur de *b*. Il faut bien s'entendre que c'est un calcul approximatif.

> λ^* **donne une approximation de la variation de** $f(x^*,y^*)$ **lorsqu'on augmente d'une unité le côté droit** *b* **de la contrainte** $h(x,y)=b$**.**

Ainsi, si f est une fonction de profit (en DZ) et *b* représente la main-d'œuvre en nombre d'employés, alors $\lambda^*=200$ signifie que le profit augmenterait (approximativement) de 200 DZ si on augmentait la main-d'œuvre d'un employé additionnel. À noter que si l'on diminue le nombre d'employés, l'effet est en sens inverse!

Mentionnons finalement que dans le cas d'un modèle d'optimisation comportant plusieurs contraintes, un tel multiplicateur est associé spécifiquement à chacune d'entre elles.

Retour sur l'exemple 3.1

$\max\ f(x,y)=-x^2-y^2 \quad$ s.c. $\quad h(x,y)=3x+4y=25$.

Pour ce problème, on a trouvé : $\lambda^*=-2$. Si on augmente b de une unité (donc $b=26$), alors la valeur optimale de $f(X)$ devrait diminuer de 2 unités (**environ**).

Si on résolvait le nouveau problème avec la contrainte $h(x,y)=3x+4y=26$ au lieu de, on obtiendrait la solution $x^*=3{,}12 \quad y^*=4{,}16$ et $f(x^*,y^*)=-(3{,}12)^2-(4{,}16)^2=-27{,}04$. Ainsi, la diminution exacte de la fonction-objectif est de 2,04.

3.3 Optimisation avec contraintes d'inégalité (cas général)

Maintenant nous allons voir comment optimiser une fonction non linéaire avec des **contraintes d'inégalité**. Supposer le problème d'optimisation avec des contraintes d'inégalité suivant:

$$\begin{cases} min_x \quad f(X)\text{, Tel que,} \\ \overrightarrow{h_i}(X) = 0\text{ , } i = 1,\ldots,l\text{ ,} \\ \overrightarrow{g_i}(X) \le 0\text{ , } i = 1,\ldots,m. \end{cases}$$

De plus, supposer que f, h_i et g_i sont continument différentiables à un point x^*. Si x^* est un minimum local qui remplit quelques conditions de régularité, alors il existe des constantes $\mu_i \ (i = 1, \ldots, m)$ et $\lambda_j \ (j = 1, \ldots, l)$ qui vérifient les contraintes suivantes :

- **Stationnarité**

$$\nabla f(x^*) + \sum_{i=1}^{m} \mu_i \nabla g_i(x^*) + \sum_{j=1}^{l} \lambda_j \nabla h_j(x^*) = 0,$$

- **Faisabilité principale**

$$h_j(x^*) = 0, \text{ for all } j = 1, \ldots, l$$
$$g_i(x^*) \leq 0, \text{ for all } i = 1, \ldots, m$$

- **Faisabilité Duelle**

$$\mu_i \geq 0 \ (i = 1, \ldots, m)$$

- **Négligence complémentaire**

$$\mu_i g_i(x^*) = 0 \text{ for all } i = 1, \ldots, m.$$

Les conditions précédentes sont les **conditions nécessaires d'optimalité de karush Kuhn et Tucker (KKT)**, qui constitue le résultat fondamental de la programmation non linéaire (PNL) :

Les conditions suffisantes qu'un x* de point est un minimum local strict du problème simple classique de PNL, où f, g_j, et le h_i sont des fonctions deux fois différentiables sont :

- **Les conditions nécessaires de KKT sont vérifiées.**
- **La matrice Hessienne est définie positive :**
  - $\nabla^2 L(x^*) = \nabla^2 f(\mathbf{x^*}) + \Sigma \mu_i \nabla^2 g_i(\mathbf{x^*}) + \Sigma \lambda_i \nabla^2 h_i(\mathbf{x^*})$

Remarque :

1. Pour un problème de maximisation remplacer le signe **« + »** par **« – »**
2. Dans certaines circonstances, les conditions de Kuhn-Tucker peuvent également être prises en tant que conditions suffisantes.

- **si**
  - ***F est une fonction concave (max problème) ou convexe (min problème)***
  - ***hi,gi* sont *des fonctions convexes(concaves).***

Alors toute solution *x qui satisfait les conditions de KKT est la solution optimale.**

Exemple : résoudre le problème d'optimisation suivant :

$$\max \ -x_1^2 - x_2^2$$
$$\text{s.t.} \ -2x_1 - x_2 \leq -1$$
$$-x_1 - 2x_2 \leq -1$$

Problème 1 Au ministère de l'agriculture, on a établi la fonction de profit suivante pour les fermes cultivant des germes de soja et des pistaches :

$$P(x, y) = 600x + 800y - x^2 - 2y^2 - 2xy$$

où $P(x, y)$ sont les profits annuels en \$, x représente le nombre d'acres plantés en germes de soja, et y donne le nombre d'acres plantés en pistaches

a) Déterminez la répartition optimale entre les deux types de récoltes (celle qui maximise les profits). Évaluer le montant de profits ainsi générés.

b) Un fermier possède une terre de 500 acres. Comment devrait-il allouer ses terres à ces deux cultures pour obtenir un profit maximal ? Utiliser la méthode de substitution, puis la méthode du Lagrangien. Montrer qu'il s'agit d'un maximum absolu et donner le montant du profit obtenu. Interpréter le multiplicateur de Lagrange. En vous basant sur cette interprétation, suggéreriez-vous au fermier d'augmenter la surface totale consacrée à ces deux cultures ou, au contraire, de la diminuer ?

Réponse

a)
$$\left.\begin{array}{l} P'_x = -2x - 2y + 600 = 0 \\ P'_y = -2x - 4y + 800 = 0 \end{array}\right\} \Rightarrow y = 100 \text{ et } x = 200$$

$$H(x,y) = \begin{bmatrix} -2 & -2 \\ -2 & -4 \end{bmatrix} \left.\begin{array}{l} \det(H(x,y)) = 4 > 0 \ \ \forall (x,y) \\ P^{(2)}_{xx}(x,y) = -2 < 0 \ \ \forall (x,y) \end{array}\right\}; P(x,y)$$ est strictement concave.

Le point stationnaire (200,100) est un maximum absolu. En plantant 200 acres de germes de soja et 100 acres de pistaches, on obtient des profits de $P(200, 100) = 110\,000$\$.

b) On cherche $\max P(x, y) = 600x + 800y - x^2 - 2y^2 - 2xy$ s.c. $x + y = 500$.

Par substitution : $y = 500\text{-}x$ et la fonction objectif devient alors $P(x) = -x^2 + 800x - 100000$.

Les conditions du premier ordre sont $P'(x) = -2x + 800 = 0 \Rightarrow x = 400$ (et $y = 500 - x = 100$).

Puisque $P^{(2)}(x) = -2 < 0, \forall x$, $(x, y) = (400,100)$ est un maximum absolu. En plantant 400 acres de germes de soja et 100 acres de pistaches, les profits sont $P(400,100) = 60\,000$\$.

Méthode du Lagrangien: $L(x, y, \lambda) = P(x, y) - \lambda(x + y - 500)$ et les conditions du premier ordre sont :

$$\left.\begin{array}{l} L'_x = -2x - 2y + 600 - \lambda = 0 \\ L'_y = -4y - 2x + 800 - \lambda = 0 \end{array}\right\} \Rightarrow -2x - 2y + 600 = -4y - 2x + 800 \Rightarrow y^* = 100$$

$$L'_\lambda = -(x + y - 500) = 0 \ \Leftrightarrow \ x + y - 500 = 0 \ .$$

En remplaçant $y^* = 100$ dans L'_λ: $x + 100 - 500 = 0 \Rightarrow x^* = 400$, on a $\lambda^* = 600 - 2x^* - 2y^* = -400$.

Le point (400,100) est un maximum absolu du problème d'optimisation sous contrainte car on maximise une fonction concave .

L'interprétation économique que l'on peut donner à $\lambda^* = -400$ signifie qu'en augmentant la surface totale consacrée à ces deux cultures, le fermier diminuerait ses profits, à cause du signe négatif de λ^*. Lorsque le fermier satisfait la contrainte, ses profits ne sont que de 60

000$, alors qu'ils sont de 110 000$ lorsque l'on ne tient pas compte de la contrainte (il cultive alors seulement 300 acres de champs).

Problème 2 Soit le problème suivant : $\max\ f(x,y) = xy$ s.c $x^2 + y^2 - 2 = 0$. Sachant qu'il existe deux maxima locaux en (1,1) et (-1, -1) ainsi que deux minima locaux en (1, -1) et (-1,1), calculer la valeur du multiplicateur de Lagrange associé à chacun de ces points.

Réponse

En un point optimal (x^*, y^*), le multiplicateur $\lambda^* = \frac{f_x'(X^*)}{h_x'(X^*)}$ ou $\frac{f_y'(X^*)}{h_y'(X^*)}$.

Les dérivées partielles des fonctions *f* et *h* sont :

$f_x'(x,y) = y,\ \ f_y'(x,y) = x,\ \ h_x'(x,y) = 2x$ et $h_y'(x) = 2y$.

Si *x** = (1,1), alors

$\lambda^* = \frac{f_x'(1,1)}{h_x'(1,1)} = \frac{1}{2}$ [si on utilise $\lambda^* = \frac{f_y'(1,1)}{h_y'(1,1)}$, on obtient la même réponse].

Si *x** = (-1, -1), alors $\lambda^* = \frac{f_x'(-1,-1)}{h_x'(-1,-1)} = \frac{-1}{-2} = \frac{1}{2}$.

Si *x** = (1, -1), alors $\lambda^* = \frac{f_x'(1,-1)}{h_x'(1,-1)} = \frac{-1}{2}$.

Si *x** = (-1,1), alors $\lambda^* = \frac{f_x'(-1,1)}{h_x'(-1,1)} = \frac{1}{-2} = -\frac{1}{2}$.

Problème 3 Pour le problème suivant déterminer les points stationnaires par la méthode du lagrangien:

$$\max\ f(x,y) = x + \ln y \ \text{ s.c } \ h(x,y) = xy + 2x - y - 11 = 0.$$

Réponse

Le Lagrangien est $L(x,y,\lambda) = x + \ln y - \lambda(xy + 2x - y - 11)$. Les conditions du premier ordre sont :

$$L_x' = 1 - \lambda y - 2\lambda = 0 \ \Leftrightarrow\ \lambda(-y-2) + 1 = 0 \ \Leftrightarrow\ \lambda = \frac{1}{y+2} \qquad (1)$$

$$L_y' = \frac{1}{y} - \lambda x + \lambda = 0 \ \Leftrightarrow\ \lambda(1-x) + \frac{1}{y} = 0 \Leftrightarrow\ \lambda = \frac{-1}{y(1-x)} \qquad (2)$$

$$L_\lambda' = -(xy + 2x - y - 11) = 0 \ \Leftrightarrow\ xy + 2x - y - 11 = 0 \qquad (3)$$

On a (1) = (2) $\Leftrightarrow\ \frac{1}{y+2} = \frac{-1}{y(1-x)} \ \Leftrightarrow\ y(1-x) = -(y+2) \ \Leftrightarrow\ 1 - x = \frac{-(y+2)}{y}$

$\Leftrightarrow\ -x = \frac{-(y+2)}{y} - 1 \ \Leftrightarrow\ x = \frac{(y+2)}{y} + 1 = \frac{y+2+y}{y} = \frac{2y+2}{y}$.

On remplace $x = \frac{2y+2}{y}$ dans (3):

$$\left(\frac{2y+2}{y}\right)y + 2\left(\frac{2y+2}{y}\right) - y - 11 = 0 \;\Leftrightarrow\; 2y + 2 + 4 + \frac{4}{y} - y - 11 = 0$$

$$\Leftrightarrow\; y - 5 + \frac{4}{y} = 0 \;\Leftrightarrow\; \frac{y^2 - 5y + 4}{y} = 0 \quad (\text{donc } y \neq 0) \;\Leftrightarrow\; y^2 - 5y + 4 = 0$$

$$\Leftrightarrow\; y = \frac{-(-5) \pm \sqrt{(-5)^2 - 4(1)(4)}}{2(1)} = \frac{5 \pm \sqrt{9}}{2} = \frac{5 \pm 3}{2} = 1 \text{ ou } 4.$$

- Si $y = 1 \;\Rightarrow\; x = \frac{2(1)+2}{1} = 4 \;\Rightarrow$ 1er point stationnaire : $(x^*, y^*) = (4,1)$

  $\Rightarrow\; \lambda^* = \frac{1}{1+2} = \frac{1}{3}.$
- Si $y = 4 \;\Rightarrow\; x = \frac{2(4)+2}{1} = \frac{5}{2} \;\Rightarrow$ 2e point stationnaire : $(x^*, y^*) = (\frac{5}{2},4)$

$\Rightarrow\; \lambda^* = \frac{1}{4+2} = \frac{1}{6}.$

Référence

1. Sylvain Perron, *cours de modélisation et optimisation*, école de gestion canada,2008.
2. Avriel, Mordecai *Nonlinear Programming: Analysis and Methods.* Dover Publishing. 2003.
3. Bertsekas, Dimitri P. *Nonlinear Programming: 2nd Edition.* Athena Scientific. 1999.

Chapitre 4 : Programmation dynamique

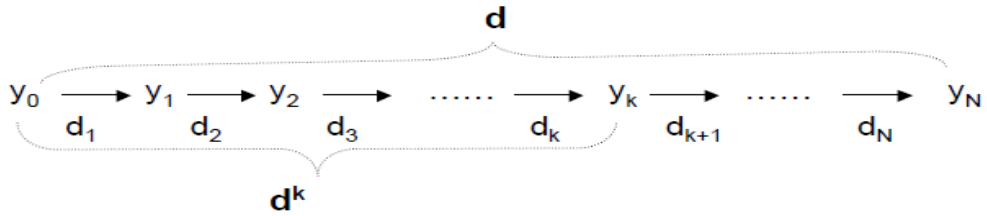
4.1 Introduction

La programmation dynamique est une méthode exacte de résolution de problèmes **d'optimisation séquentielle**, due essentiellement à R. Bellman (1957). Elle est basée sur le principe d'optimalité de Bellman : *" Toute sous politique, d'une politique optimale, est optimale"* :

Définition (sous politique) : étant donné une politique $\mathbf{d}= d_1 d_2 \dots d_N$ générant la séquence des états $y_0 y_1 \dots y_{N-1} y_N$. La séquence de décisions $\mathbf{d^k}=d_1 d_2 \dots d_k$ qui génère la séquence $y_0 y_1 \dots y_k$ sera dite une sous politique de **d**.

Principe de Bellman : « Toute sous politique d'un politique optimale est elle-même optimale ».

Ainsi si la politique $\mathbf{d}= d_1 d_2 \dots d_k \dots d_N$, générant la séquence des états $y_0 y_1 \dots y_k \dots y_{N-1} y_N$ est optimale parmi toutes celles qui permettent d'atteindre y_N à partir de y_0 alors la sous politique $\mathbf{d^k}= d_1 d_2 \dots d_k$ est optimale parmi toutes celles qui permettent d'atteindre y_k à partir de y_0.



Contrairement à la programmation linéaire, il n'y a pas un formalisme mathématique standard. C'est une approche de résolution où les équations doivent être spécifiées selon le problème à résoudre.

La programmation dynamique est similaire à la méthode diviser et régner en ce sens que, une solution d'un problème dépend des solutions précédentes obtenues des sous-problèmes. La différence significative entre ces deux méthodes est que la programmation dynamique permet aux sous-problèmes de se superposer. Autrement dit, un sous-problème peut être utilisé dans la solution de deux sous-problèmes différents. Tandis que l'approche diviser et régner crée des sous-problèmes qui sont complètement séparés et peuvent être résolus indépendamment l'un de l'autre.

La différence fondamentale entre ces deux méthodes devient alors claire: les sous-problèmes dans la programmation dynamique peuvent être en interaction, alors dans la méthode diviser et régner, ils ne le sont pas (figure 2.1).

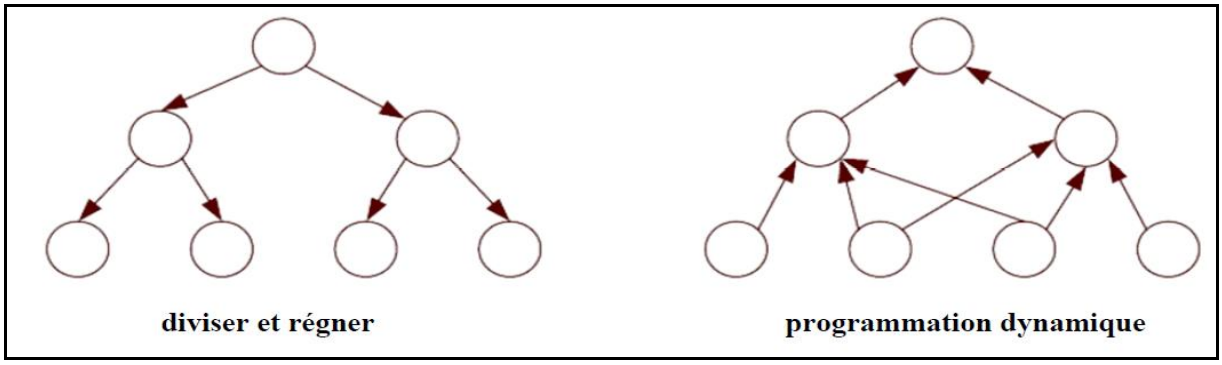


Figure 4.1 – différence entre la programmation dynamique et la méthode « diviser pour régner »

Une seconde différence entre ces deux méthodes est, comme illustré par la figure ci-dessus, est que la méthode diviser et régner est récursive, les calculs se font de haut en bas. Tandis que la programmation dynamique est une méthode dont les calculs se font de bas en haut : on commence par résoudre les plus petits sous-problèmes. En combinant leur solution, on obtient les solutions des sous-problèmes de plus en plus grands.

La programmation dynamique est un "paradigme" simple de conception d'algorithmes qu'on peut appliquer pour résoudre un problème si:

1**.** La solution (optimale) d'un problème de taille n s'exprime en fonction de la solution (optimale) de problèmes de taille inférieure à n -c'est le principe d'optimalité-.

2. Une implémentation récursive "naïve" conduit à calculer de nombreuses fois la solution de mêmes sous-problèmes.

Comment? L'idée est alors simplement d'éviter de calculer plusieurs fois la solution du même sous-problème.

On définit une table pour mémoriser les calculs déjà effectués: à chaque élément correspondra la solution d'un et d'un seul problème intermédiaire, un élément correspondant au problème final. Il faut donc qu'on puisse déterminer les sous-problèmes (ou un sur-ensemble de ceux-ci) qui seront traités au cours du calcul (ou un sur-ensemble de ceux-ci) ... Ensuite il faut remplir cette table; il y a deux approches, l'une itérative, l'autre récursive:

- **Version itérative:** On initialise les "cases" correspondant aux cas de base. On remplit ensuite la table selon un ordre bien précis à déterminer: on commence par les problèmes de "taille" la plus petite possible, on termine par la solution du problème principal: il faut bien sûr qu'à chaque calcul, on n'utilise que les solutions déjà calculées. Le but est bien sûr que chaque élément soit calculé une et une seule fois.
- **Version récursive (tabulation, memoizing):** A chaque appel, on regarde dans la table si la valeur a déjà été calculée (donc une "case" correspond à un booléen et une valeur ou à

une seule valeur si on utilise une valeur "sentinelle"). Si oui, on ne la recalcule pas: on récupère la valeur mémorisée; si non, on la calcule et à la fin du calcul, on stocke la valeur correspondante. Donc si la fonction f est définie par: fonction f(parametres p)

si cas_de_base(p) alors g(p) sinon h(f(p_1),...,f(p_k))

D'une manière générale, la programmation dynamique est nécessaire lorsque le traitement d'un problème de taille n dépend du traitement des problèmes de taille n_i, avec la relation suivante :

$$T_{pb}(n) = f(n) + \sum_i T_{pb}(n_i) \quad \wedge \sum_i n_i > n$$

f(n) : *fonction objectif*

Exemple prototype. Le problème du voyageur

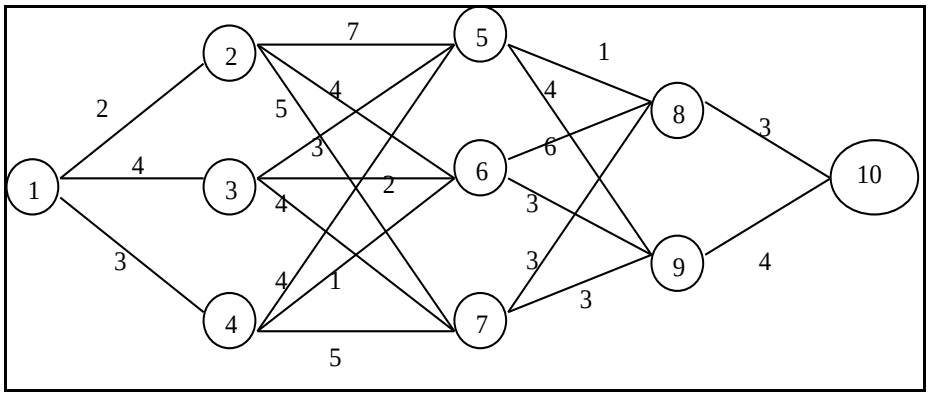


Figure 4.1 – exemple du problème de voyageur.

Un transporteur décide d'effectuer un voyage entre différentes villes qu'il ne connaît pas. Son but est de choisir le chemin le moins couteux. Pour cela, il calcule les différents coûts possibles (péage, carburant, restauration, prix hôtels) entre les différentes routes possibles du voyage.

Le trajet du transporteur est composé de 4 étapes (figure 2.1) et il doit arriver à la ville numérotée 10 en partant de la ville numérotée 1. Les autres numéros représentent les villes susceptibles d'être traversées. Les arcs entre ces nœuds représentent les différents trajets possibles et les valeurs c_{ij} au-dessus des arcs représentent le coût relatif au trajet décrit par l'arc.

Exemple : Si le voyageur empreinte le trajet $1 \rightarrow 2 \rightarrow 6 \rightarrow 9 \rightarrow 10$, le coût total du voyage est 13

Soit x_n $(n=1, ..., 4)$ les variables de décisions relatives à chacune des 4 étapes. Le chemin à suivre par le voyageur est $1 \rightarrow x_1 \rightarrow x_2 \rightarrow x_3 \rightarrow x_4$ avec $x_4 = 10$.

Soit $f_n(S, x_n)$ le coût total pour le reste des étapes sachant que nous somme à l'état S de l'étape n et que la destination choisie est x_n.

Etant donné *S* et *n*, soit x_n^* la valeur de x_n qui minimise $f_n(S, x_n)$ et soit $f_n^*(S)$ la valeur minimale de $f_n(S, x_n)$. ($f_n^*(S) = f_n(S, x_n^*)$).

L'approche de la programmation dynamique repose sur l'idée qu'un chemin ne peut être optimal que si chacune de ses composantes est-elle même optimale. La démarche de la programmation dynamique consiste à étudier d'abord les sous problèmes qui se situent chronologiquement les derniers et sur un principe de retour en arrière.

L'objectif est donc de trouver la valeur de $f_1^*(1)$. Pour l'avoir, la programmation dynamique va essayer d'évaluer avec une procédure de chaînage en arrière $f_4^*(s)$, $f_3^*(s)$, $f_2^*(s)$ et enfin $f_1^*(1)$.

A chaque étape, on va essayer d'évaluer pour chaque état *s*, la valeur de $f(S, x_n)$ pour chaque destination x_n possible, puis de retrouver la meilleure destination x_n^* (celle qui correspond au plus petit coût $f(S, x_n)$) et aussi la valeur de $f_n^*(s)$.

Etape 4

x_4 / S	$f_4(S, x_4)$	$f_4^*(s)$,	x_4^*
8	3	3	10
9	4	4	10

Etape 3

	$f_3(S, x_3) = C_s x_3 + f_4^*(x_3)$			
x_3 / S	8	9	$f_3^*(s)$,	x_3^*
5	4(=1+3)	8(=4+4)	4	8
6	9	7	7	9
7	6	7	6	8

Etape 2

	$f_2(S, x_2)=C_s x_2 + f_3^*(x_2)$				
S \ x_2	5	6	7	$f_2^*(s)$	x_1^*
2	11	11	12	11	5 ou 6
3	7	9	10	7	5
4	8	8	11	8	5 ou 6

Etape 1

	$f_1(S, x_1)=C_s x_1 + f_2^*(x_1)$				
S \ x_1	2	3	4	$f_1^*(1)$	x_1^*
1	13	11	11	11	3 ou 4

La valeur de $f_1^*(1) = 11$ représente le coût total pour le voyage. Le chemin optimal n'est unique puisque dès le départ on peut choisir $x_1^* = 3$ ou 4, donc l'ensemble de ces chemins est:

$1 \to 3 \to 5 \to 8 \to 10$

$1 \to 4 \to 5 \to 8 \to 10$

$1 \to 4 \to 6 \to 9 \to 10$

4.2 Caractéristiques d'un problème de programmation dynamique

Nous allons maintenant sur la base de l'exemple précédant analyser les propriétés communes aux problèmes de programmation dynamique.

(i) Le problème peut être décomposé en étapes et une décision doit être prise à chaque étape.

Le dernier exemple est devisé en 4 étapes où à chaque étape, la décision à prendre est celle de la destination à choisir. Ces décisions sont interdépendantes et séquentielles.

(ii) A chaque étape correspond un certain nombre d'états. Dans l'exemple du voyageur, les états à chaque étape sont représentés par les villes que le voyageur les visiter. Le nombre de ces états dans l'exemple est fini. Le nombre d'états peut être infinie ($x_n \in IN$) ou continue ($x_n \in IR$)

(iii) A chaque étape, la décision prise transforme l'état actuel en un état associé à l'étape suivante (dans certains cas avec une distribution de probabilité).

Dans notre exemple, se trouvant à une ville donnée, le voyageur décide de se rendre à une autre ville qui est un état de l'étape suivante.

(iv) Etant donné un état, une stratégie optimale pour les étapes restantes est indépendante des décisions prises aux étapes précédentes.

Si le voyageur est dans un état quelconque de l'étape i, le chemin optimal entre cet état et l'état 10 sera indépendant de la façon dont il y est arrivé. En d'autres termes, l'état actuel contient toute l'information nécessaire aux décisions futures. Cette propriété est dite principe d'optimalité.

(v) L'algorithme de recherche de la solution optimale commence par trouver la stratégie optimale pour tous les états de la dernière étape.

(vi) Une relation de récurrence identifie la stratégie optimale dans chaque état de l'étape n à partir de la stratégie optimale dans chaque état de l'étape n+1.

Dans notre exemple la relation est $f_n^*(S) = \underset{x_n}{Min}\{C_s x_n + f_{n+1}^*(x_n)\}$

La stratégie optimale étant donné que nous sommes à l'état *S* de l'étape *n*, nécessite de retrouver la valeur de x_n qui minimise l'expression ci-dessus.

La relation de récurrence à toujours cette forme $f_n^*(S) = \underset{x_n}{Max}$ *ou* $\underset{x_n}{Min}\{f_n(S,x_n)\}$,

avec $f_n(S,x_n)$ est une expression en fonction de *S*, x_n et $f_{n+1}^*(-)$

(vii) Utilisant cette relation de récurrence, l'algorithme procède à reculons étape par étape. Il détermine la stratégie optimale pour chaque état de chaque étape.

Dans tout problème de programmation dynamique, on peut construire à chaque étape un tableau analogue au suivant.

Etape n

	$f_n(S, x_1)$		
S \ X_1	*états n+1*	$f_n^*(s)$	x_n^*
états n	*le coût ou la distance*	*La longueur du chemin optimal à partir de S jusqu'à l'état final*	*Le meilleur état de l'étape n+1 le long du chemin optimal final*

D'une manière simple Si on est dans le cadre de la méthode de la programmation dynamique, les étapes suivies peuvent être résumées comme suit :

1) **Obtention de l'équation récursive** liant la solution d'un problème à celle de sous problèmes.
2) **Initialisation de la table**: cette étape est donnée par les conditions initiales de l'équation obtenue à l'étape 1.

3) **Remplissage de la table**: cette étape consiste à résoudre les sous-problèmes de tailles de plus en plus grandes, en se servant bien entendu de l'équation obtenue à l'étape 1.

4) **Lecture de la solution** : l'étape 3 ne conduit qu'à la valeur (optimale) du problème de départ. Elle ne donne pas directement la solution conduisant à cette valeur. En générale, pour avoir cette solution, on fait un travail inverse en lisant dans la table en partant de la solution finale et en faisant le chemin inverse des calculs effectués en à l'étape 3.

4.3 Programmation dynamique déterministe

Dans cette section on s'intéresse au problème dit déterministe, où la connaissance de l'état et de la décision à prendre suffisent pour savoir l'état à l'étape suivante.

Un problème dynamique déterministe est caractérisé par la détermination de la fonction objective. Cette fonction peut être le minimum de la somme de la contribution induite par le passage d'un état à un autre, ou le maximum d'une telle somme, ou le minimum du produit de ces termes… etc.

Il faut aussi déterminer la nature de l'ensemble des états dans chacune des étapes. Ces états S_n peuvent être représentés par des variables discrètes ou par des variables continues ou dans certains cas par un vecteur.

étape n étape n+1

S_n contribution de x_n S_{n+1}

La structure de base d'un problème dynamique déterministe.

Exemple : Répartition optimale des moyens

Un projet du gouvernement est étudié par 3 groupes de chercheurs. La probabilité que chacun de ces groupes 1, 2 et 3, n'arrive pas à terminer le projet est respectivement: 0,4; 0,6 et 0,8.

Si on ajoute à ces groupes deux nouveaux chercheurs, les probabilités d'échec sont données par ce tableau

	Probabilité d'échec		
Nbre de nouveaux chercheurs	*Groupes*		
	1	*2*	*3*
0	*0,4*	*0,6*	*0,8*
1	*0,2*	*0,4*	*0,5*
2	*0,15*	*0,2*	*0,3*

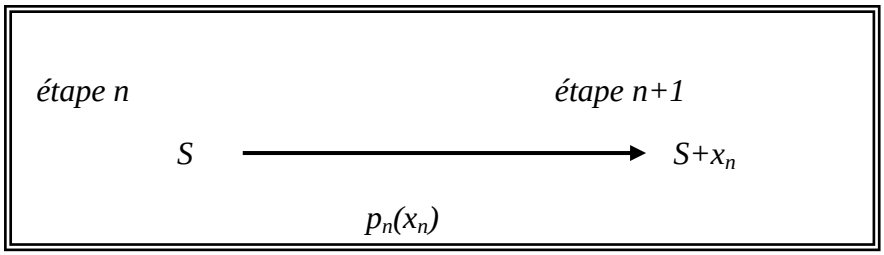
Le problème est de déterminer l'allocation optimale de ces deux chercheurs afin de minimiser la probabilité que les groupes de recherche échouent dans leur travail.

Solution:

- Etapes: 3 étapes ou les états représentent le nombre de chercheurs disponibles
- Variable de décision: x_n représente le nombre de chercheurs à allouer à l'équipe de recherche *n, n = 1, 2, 3.*
- Contribution de la décision x_n est la probabilité que l'équipe n échoue après avoir eu x_n chercheur de plus.
- $f_n^*(S)$ c'est la probabilité minimale que les groupes *n=1,2, 3* échouent dans leurs recherches :

$f_n^*(S) = \underset{x_n \leq S}{Min} f_n(S, x_n)$ $n = 1, 2, 3$

avec $f_n(S, x_n) = p_n(x_n) \times f_{n+1}^*(S, x_n)$, $p_n(x_n)$ est la contribution de la décision x_n et $f_3^*(s) = 1$.



$f_n(S, x_n) = p_n(x_n)\ f_{n+1}^*(S - x_n)$

Etape 3

S \ x_3	$f_3(S, x_3) = p_3(x_3)$ 0	1	2	$f_3^*(S)$	x_3^*
0	0,8	-	-	0,8	0
1	0,8	0,5	-	0,5	1
2	0,8	0,5	0,3	0,3	2

Etape 2

	$f_2(S, x_2) = p_2(x_2)\ f_3^*(x_3)$				
S \ x_2	0	1	2	$f_2^*(S)$	x_2^*
0	0,48	-	-	0,48	0
1	0,3	0,32	-	0,3	0
2	0,18	0,2	0,16	0,16	2

Etape 1

	$f_1(S, x_1) = p_1(x_1)\ f_2^*(x_1)$				
S \ x_1	0	1	2	$f_1^*(S)$	x_1^*
2	0,064	0,06	0,072	0,06	1

La stratégie optimale est $x_1^* = 1$, $x_2^* = 0$ et $x_3^* = 1$.

La probabilité d'échec des trois groupes de recherche est de 0,06.

Exercice à faire : résoudre le *problème de sac à dos* en utilisant la programmation dynamique

Référence :

R.E. Bellman. Dynamic Programming and Modern Control Theory. 1965. New York - Academic Press

E.V. Denardo. Dynamic Programming. Prentice-Hall, Eglewood Cliffs, 2003.

Chapitre 5 : Programmation linéaire en nombres entiers

5. 1. Programmation linéaire

La programmation linéaire (PL) est une branche de l'optimisation permettant de résoudre de nombreux problèmes économiques et industriels. La programmation linéaire désigne la manière de résoudre les problèmes dont la fonction objectif et les contraintes sont toutes linéaires. La programmation linéaire est un domaine central de l'optimisation, car les problèmes de PL sont les problèmes d'optimisation les plus faciles - toutes les contraintes y étant linéaires. Beaucoup de problèmes réels de recherche opérationnelle peuvent être exprimés comme un problème de PL. Pour cette raison un grand nombre d'algorithmes pour la résolution d'autres problèmes d'optimisation sont fondés sur la résolution de problèmes linéaires. Le terme programmation linéaire suppose que les solutions à trouver doivent être représentées en variables réelles. S'il est nécessaire d'utiliser des variables discrètes dans la modélisation du problème (contraintes dites d'intégrité), on parle alors de programmation linéaire en nombres entiers (PLNE). Il est important de savoir que ces derniers sont nettement plus complexes à résoudre que les PL à variables continues.

a)Modélisation :

L'objectif de la méthode de programmation linéaire est minimiser ou (maximiser) une fonction linéaire de n variables réelles non négatives (les coefficients sont notées c_j) sur un ensemble de même inégalités linéaires.

Max (ou min) z = $\sum_{j=1}^{n} c_j x_j$

$\sum_{j=1}^{n} a_{ij} x_{j \leq b_i}$ i = 1,....., r

$\sum_{j=1}^{n} a_{ij} x_j = b_i$ i= r+1,....., s

$\sum_{j=1}^{n} a_{ij} x_j \geq b_i$ i= s+1,....., m

S.c.q x_j =0 j=1, p

$x_j \in \mathbb{R}$ j=p+1, q

x_j =0 j=q+1, n

La forme standard d'un problème linéaire est comme la suivante :

max (ou min) $z= c_1 x_1 + c_2 x_2 +...c_n x_n$

$a_{11} x_1 + a_{12} x_2 +...+a_{1n} x_n =b_1$

$a_{21}x+a_{22}x_2+...+a_{2n}x_n=b_2$ s.c.q $a_{m1} x_1 + a_{m2} x_2+...a_{mn} x_n =b_m$

$x_1, x_2, ...,x_n=0$

5.2. Programmation linéaire en nombre entier

Dans la programmation linéaire en nombre entier PLNE chaque variable est restreinte aux valeurs entières, en général les coefficients a_{ij} sont aussi entiers, et donc on peut se limiter à des constantes b_j entières.

Max (ou min) z = $\sum_{j=1}^{n} c_j x_j$

S .c $\sum_{j=1}^{n} a_{ij} x_j = b_i$ i=1,...,n

$x_j=0$, entier j=1,...,n

Les variables binaires peuvent se voir comme des variables entières soumises à la contrainte d'appartenir à l'intervalle [0 ,1].En effet, la contrainte :

$X_j=0, 1$

est évidemment équivalente à :

$X_j=0$ et $X_j =1$ et X_j entier

Plusieurs problèmes d'ordonnancement, tel que notre problème de « car -sequencing », peuvent être formulées sous la forme de programmation linéaire en nombre entier PLNE.

Un moyen naturel pour résoudre un programme linaire en nombre entier (PLNE) de très petite taille consiste à calculer l'ensemble des solutions entières du problème et à en retenir la meilleure. Or, au-delà de quelques variables, cette énumération explicite devient impossible du fait de la combinatoire du problème. L'idée de la recherche arborescente par séparation et évaluation est d'effectuer une énumération implicite des solutions du PLNE. La méthode par évaluation et séparation progressive (en anglais "Branch and Bound") connue aussi sous le pseudo B&B.

5.3. Méthode Branch and Bound (séparation et évaluation)

Pour plusieurs problèmes, en particulier les problèmes d'optimisation, l'ensemble de leurs solutions est fini (en tous les cas, il est dénombrable). Il est donc possible, en principe, d'énumérer toutes ces solutions, et ensuite de prendre celle qui nous arrange. L'inconvénient majeur de cette approche est le nombre prohibitif du nombre de solutions : il n'est guère évident d'effectuer cette énumération.

L'idée de la recherche arborescente par séparation et évaluation est d'effectuer une énumération implicite des solutions d'un problème. La méthode par évaluation et séparation progressive (en anglais "Branch and Bound") connue aussi sous le pseudo B&B est inspirée du principe latin "Divide ut Regnes" qui préconise de diviser ses ennemis pour régner plus aisément. Cette méthode consiste à dénombrer les solutions d'un problème d'une manière intelligente en ce sens que, en utilisant certaines propriétés du problème en question. Contrairement, à la méthode purement énumérative, Cette

technique utilise des heuristiques pour s'écarter des solutions incomplètes qui ne conduisent pas à la solution que l'on souhaite. De ce fait, on arrive généralement à obtenir la solution recherchée en des temps raisonnables. Bien entendu, dans le pire cas, on retombe toujours sur l'élimination explicite de toutes les solutions du problème. Le principe de l'algorithme B&B est le suivant :

Principe : On décompose S l'ensemble des solutions de P en $S_1, S_2, \ldots, S_m$ tels que
$\begin{cases} S = \bigcup_{i=1,\ldots,m} S_i \\ |S_i| < |S| \text{ pour } i = 1, \ldots, n \end{cases}$
et on itère la décomposition jusqu'à ce qu'on obtienne des ensembles de solutions calculables (typiquement quand $|S| = 1$, mais pas seulement)

Pour appliquer la méthode de branch-and-bound, nous devons être en possession :

- Le calcul d'une borne inférieure (LB): la borne inférieure permet d'éliminer certaines solutions de l'arbre de recherche afin de faciliter l'exploration du reste de l'arbre et de faire converger la recherche vers la solution optimale.
- Le calcul d'une borne supérieure (UB): la borne supérieure peut être une heuristique qui perrnet d'élaguer certaines branches de l'arbre de recherche.
- Application des règles de dominance: dans certains cas et pour certains problèmes, il est possible d'établir des règles qui perrnettent d'élaguer des branches de l'arbre de recherche.
- Stratégie de recherche: Il existe plusieurs techniques pour l'exploration de l'arbre de recherche. On peut utiliser soit une exploration en profondeur d'abord (Depth First Search -DFS), soit une exploration en largeur d'abord (Breadth First Search-BFS), soit encore une recherche qui utilise une file de priorité.
- Choix du critère d'évaluation: Lors d'une application de branch and bound, on a besoin d'avoir une fonction qui permet d'évaluer chaque nœud traité, et éventuellement élaguer ceux qui sont inutiles. De même, un nœud peut être élagué dans trois cas possibles: dans le premier cas, on arrive à un stade où la valeur de la borne inférieure d'un nœud courant est plus grande ou égale à la valeur de la borne supérieure qu'on avait établie auparavant. Dans le deuxième cas, la solution n'est pas réalisable, l'un des critères d'évaluation n'a pas été respecté. Le troisième cas est le cas où on a obtenu une solution réalisable, tous les critères sont valides, mais la solution obtenue est supérieure à la borne inférieure.

5.3 .1 Méthode de résolution générale d'un problème en PLNE par B&B

La méthode générique utilisée pour résoudre un problème linéaire en nombre entier est le la suivante :

1. Initialisation :
   - Calculer un LB par une heuristique qui donne des solutions entières
   - Calculer un UB par la méthode du simplex

2. choisir une variable xi et une constante xi* et ajouter la contrainte suivante xi ≥ xi*+1 (xi ≤ xi*)
3. diviser le problème selon ces deux contraintes (voir exemple)
2. Pour chaque noeud i calculer UBi de ce noeud par la méthode du simplexe

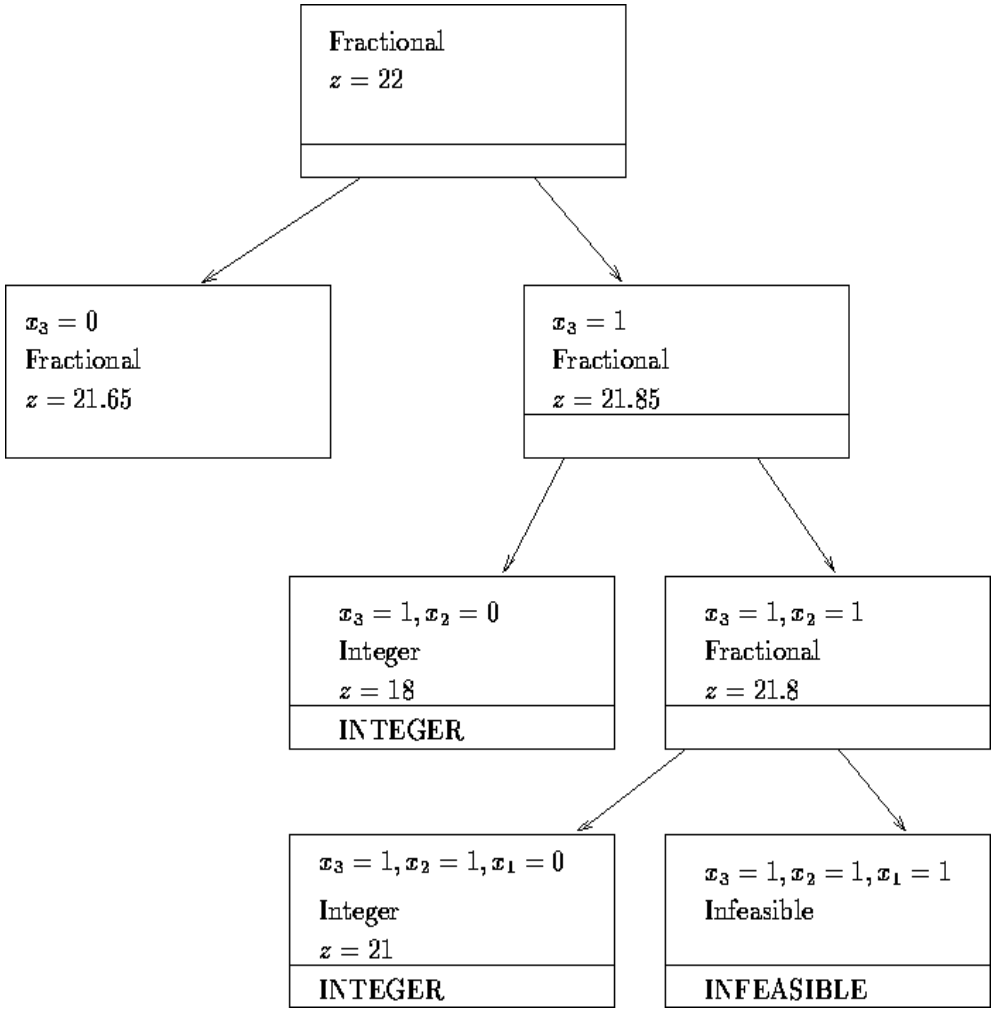
Si (UBi < LB)
Séparer ce noeud

Sinon (si UBi est fractionné)
Aller à 2
Sinon (si UBi est entier)
Mettre LB=UBi
Aller à 2

Exemple : soit le problème PLNE suivant

$$\begin{array}{ll} \text{Maximize} & 8x_1 + 11x_2 + 6x_3 + 4x_4 \\ \text{subject to} & 5x_1 + 7x_2 + 4x_3 + 3x_4 \leq 14 \\ & x_j \in \{0,1\}\ j = 1, \ldots 4. \end{array}$$

L'utilisation de la méthode B&B va générer le graphe suivant. L'ordre de branchement est x3,x2,x1



- ***Discussion***

La procédure B&B est un algorithme et non une heuristique : nous avons démontré qu'elle fournit bien la solution optimale du problème qu'elle résout. Bien que généralement, elle donne de bons résultats en terme de vitesse de calcul, il peut arriver qu'elle doive "backtracker" un grand nombre de fois et qu'elle finisse par construire et explorer un arbre gigantesque.

Lorsque le temps calcul est limité, la procédure B&B offre l'avantage de fournir souvent bien avant son arrêt, une solution réalisable x_R , à défaut d'être optimale. Dans les programmes d'optimisation

B&B, la découverte de x_R puis la meilleure estimation $F(x_R)$ est signalée régulièrement à l'utilisateur qui peut choisir d'arrêter l'algorithme dès qu'il estime avoir atteint une solution satisfaisante ou que le gain espéré, qui est au mieux la différence entre la solution actuelle et la meilleure borne inférieure encore à explorer, lui paraît négligeable. La recherche d'une ou de toutes les solutions optimales est possible par la simple modification du test d'arrêt:

- Si on veut une seule solution on s'arrête dès qu'une solution réalisable a une valeur ≤ toutes estimations des feuilles non encore explorées.
- Si on veut toutes les solutions optimales, on stoppe dès qu'une (ou plusieurs) solution réalisable a une valeur < toutes estimations des feuilles non encore explorées et on stocke toutes les solutions réalisables réalisant le minimum.

Remarque : Il existe plusieurs amélioration de l'algorithme branch-and-bound, comme branch-and-cut, branch-and-price que les note généralement par l'expression « *branch and * »*.

Application : le voyageur de commerce

Ce problème connu sous le nom du TSP est très connu en informatique et possède plusieurs applications dans différents domaines. Rappelons tout de même sa définition.

Définition 1 : Soit un graphe G=(V,E). Un cycle est hamiltonien si et seulement si tous les sommets de G apparaissent une et seule fois dans ce cycle.

Définition 2: Soit un graphe G=(V,E) valué. Le problème du voyageur de commerce consiste à trouver un cycle hamiltonien dont la somme des poids est minimale.

Soit donc le graphe de la Figure 1:

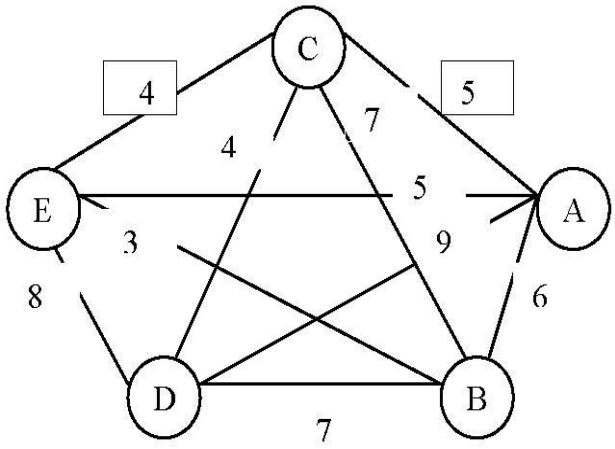


Figure 1. Graphe hamiltonien valué

Cet graphe peut être bien entendu représenté par sa matrice D d'adjacence tel que D[i,j] représente le poids de l'arc (i,j). Résolvons ce problème en commençant par exemple à partir du sommet E.

Soit la borne v pour ce sommet (on verra plus loin comment la trouver !) c'est à dire la valeur de toutes les solutions incluant le sommet E vont avoir un coût ≥ v .

Le prochain sommet du cycle que nous recherchons est soit A, B, C ou D. Pour chacune de ces solutions partielles, nous calculons une nouvelle borne (signifiant que toutes les solutions comprenant cette solution partielle va avoir un coût ≥ à cette nouvelle borne).

Initialement, nous pouvons mettre le coût de la meilleure solution trouvée à un très grand nombre ou bien de le trouver par une quelconque autre méthode (aléatoire ou autre).

L'idée de cette méthode la suivante : toute solution partielle dont le coût est plus grand que celui de la meilleure solution trouvée jusqu'à présent va être exclue de la recherche. Ce processus est continué jusqu'à avoir une solution complète. Si le coût de cette solution complète est inférieur à celui de la meilleure solution que nous avons déjà, alors nous remplaçons ce coût comme étant celui de la meilleure solution courante.

Il existe plusieurs manières de décider quelle solution partielle à explorer en premier. Celle que nous allons décrire ci-dessous est celle qui consiste à choisir toujours la solution ayant la plus petite borne. Mais il existe d'autres manières de procéder. Les unes se valent que les autres.

La partie cruciale de branch and bound est la qualité de la fonction F. Si elle est trop simple, probablement que l'exclusion des solutions partielles se fera à des moments avancés. Cela a pour effet de rallonger encore les temps d'exécution des ces algorithmes.

Une fonction F peut être comme suit :

Lemme : Soit le cycle hamiltonien suivant : $v_1, v_2, .., v_{n+1} = v_1$. Il est facile de voir que, quelque soit la solution, son coût est $\geq 1/2 \sum_{i=1}^{n} (\text{arc} - v_{i1} + \text{arc} - v_{i2})$ où $\text{arc} - v_{i1}$ et $\text{arc} - v_{i2}$ désignent deux arcs adjacents au sommet i ayant le plus petit poids.

Preuve : Quelque soit le cycle hamiltonien, il doit y avoir deux arcs incidents à chaque sommet du graphe : un arc sortant ce sommet et un arc entrant ce sommet. Le poids de l'arc sortant le sommet i est donné par D[i,j] , pour un certain $j \neq i$. Le poids de l'arc entrant est aussi donné par D[k,i], pour un certain $k \neq j$, car les deux arcs devant être distincts. Par conséquent, quelque soit un cycle hamiltonien, la somme des poids de deux arc incidents à un sommet i doit être supérieure à celle des plus petits arcs deux arcs incidents à i. En sommant sur les i, on compte par deux fois chacun des arcs, d'où le résultat du lemme.

Appliquons cette borne sur le graphe ci-dessus. En commençant à partir de E, on aura donc:

$$\text{Le coût est} \geq ½\{(3+4)+(4+4)+(5+5)+(3+6)+(4+7)\} = 22.5$$

Les prochains sommets dans le cycle peuvent être : A, B, C ou D. Pour chacune des ces solutions partielles, une borne est calculée. Par exemple pour D, la nouvelle borne est :

$$½\{(8+3)+(4+4)+(5+5)+(3+6)+(4+8)\} = 25.$$

Pour calculer cette borne, on applique la fonction cidessus en tenant compte que pour le sommet E, l'arc (E,D) doit être pris, et pour le sommet D, l'arc (D,E) doit être pris. La valeur 25 représente donc la plus petite valeur d'un cycle hamiltonien incluant les arcs (E,D) et (D,E) ; les arcs n'étant pas orientés dans ce graphe.

Ceci est fait pour chacun des sommets. On obtient successivement : 22.5 pour le sommet C, 23 pour le sommet A et 22.5 pour le sommet B. Parmi ces quatre valeurs, nous allons explorer le sommet ayant la plus petite valeur, soit le sommet C, avant d'explore le second sommet ayant la plus petite borne, etc. Il est utile de signaler qu'il existe d'autres manières de procéder dans le choix de la prochaine solution

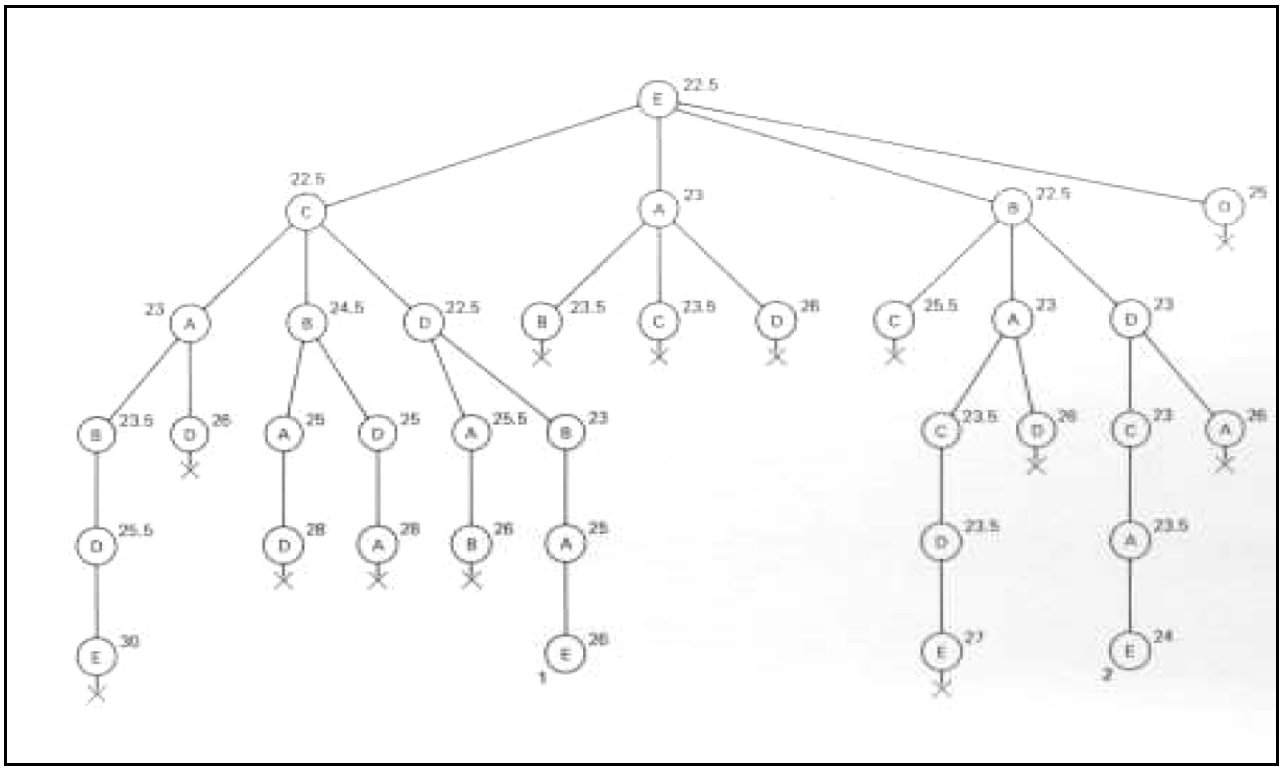
partielle à explorer. Toutefois, chacune des ces méthodes a ses propres avantages et inconvénients. À titre d'exemple, nous pouvons citer l'exploration en profondeur d'abord (DFS) et la méthode en largeur d'abord (BFS).

Ce processus est ensuite répété. Quand une solution complète est trouvée, la meilleure solution courante est modifiée si cela est nécessaire. Par exemple, dans notre exemple, la première solution trouvée a pour coût 26 remplaçant ainsi l'ancienne valeur que nous avons initialisée à un grand nombre.

L'idée de Branch and Bound peut être résumée comme suit :

Quand un sommet a une borne plus grande que cette valeur, il n'y a plus lieu de le considérer étant donné que cette solution partielle ne mènera pas à la solution minimale.

Ce processus de génération de solutions partielles, avec le calcul de leur borne, génère une arborescence. Pour l'exemple cidessus, l'arborescence obtenue est la suivante :



Remarquez qu'à égalité des bornes, on choisit au hasard. On prend en premier le sommet C, on génère les sommets A, B et D. On ensuite, on explore le sommet D. Ensuite B, A et E. à ce stade, nous avons une solution complète (ECDBAE) de valeur 26. Ensuite, on explore le sommet A qui va donner le sommet B, de valeur 26. Il n'y pas lieu d'explorer encore cette solution partielle car sa borne inférieure vaut déjà 26.

Exercice à faire : résoudre le *problème de sac à dos* en utilisant la méthode branch and bound

Chapitre 6 : Introduction aux métaheuristiques

6.1 Les Métaheuristiques

Les métaheuristiques constituent une classe de méthodes qui fournissent des solutions de bonne qualité en temps raisonnable à des problèmes combinatoires réputés difficiles pour lesquels on ne connaît pas de méthode classique plus efficace. On appelle métaheuristiques (du grec, meta = qui englobe) des méthodes conçues pour échapper aux minima locaux. Le terme meta s'explique aussi par le fait que ces méthodes sont des structures générales dont il faut instancier les composants en fonction du problème par exemple, le voisinage, les solutions de départ ou les critères d'arrêt. Les métaheuristiques sont généralement des algorithmes stochastiques itératifs, qui progressent vers un optimum global, c'est-à-dire l'extremum global d'une fonction en évaluant une certaine fonction objectif. Elles se comportent comme des algorithmes de recherche, tentant d'apprendre les caractéristiques d'un problème afin d'en trouver une approximation de la meilleure solution d'une manière proche des algorithmes d'approximation. L'intérêt croissant apporté aux métaheuristiques est tout à fait justifié par le développement des machines avec des capacités calculatoires énorme, ce qui a permis de concevoir des métaheuristiques de plus en plus complexes qui ont fait preuve d'une certaine efficacité lors de la résolution de plusieurs problèmes à caractère NP-difficile.

Il existe de nombreuses métaheuristiques allant de la simple recherche locale à des algorithmes plus complexes de recherche globale. Ces méthodes utilisent cependant un haut niveau d'abstraction, leur permettant d'être adaptées à un large éventail de problèmes d'optimisation combinatoire. Nous pouvons partager les méthodes heuristiques en deux catégories. Celles qui permettent de déterminer un minimum local, et celles qui s'efforcent de déterminer un optimum global.

On appelle méthode (ou algorithme ou recherche) locale celle qui converge vers un minimum local. Les méthodes de recherche locale, appelées aussi méthodes de recherche par voisinages, partent d'une solution initiale et, par raffinements successives, construisent des suites de solutions de coûts décroissants pour un problème de minimisation. Le processus s'arrête lorsqu'on ne peut plus améliorer la solution courante ou parce que le nombre maximal d'itérations (fixé au départ) est atteint. Quoique, ces méthodes ne soient pas complètes (rien n'assure qu'elles pourront trouver toutes les solutions existantes), ni n'assurent la preuve

d'optimalité, de telles méthodes parviennent très souvent à trouver des solutions de bonne qualité dans des temps de calcul raisonnables. En effet, elles sont souvent les premières méthodes testées sur les nouveaux problèmes combinatoires émergeant des applications réelles et académiques. On trouve dans la littérature de nombreuses méthodes locales. Les plus anciennes et les plus utilisées sont : la méthode de la descente, le recuit simulé, la recherche tabou, etc.

Contrairement aux méthodes de recherche locale, les méthodes globales ont pour objectif d'atteindre un ou plusieurs optima globaux. Ces méthodes sont appelées également des méthodes à population. Celles-ci sont d'une grande diversité : parmi elles on retrouve notamment les algorithmes génétiques, les algorithmes à évolution différentielle, la recherche dispersée, etc.

D'autre part, on peut partager les métaheuristiques en deux grandes classes: les métaheuristiques à solution unique (i.e.; évoluant avec une seule solution) et celles à solution multiple ou population de solution (Figure 6.1). Les méthodes d'optimisation à population de solutions améliorent, au fur et à mesure des itérations, une population de solutions. L'intérêt de ces méthodes est d'utiliser la population comme facteur de diversité.

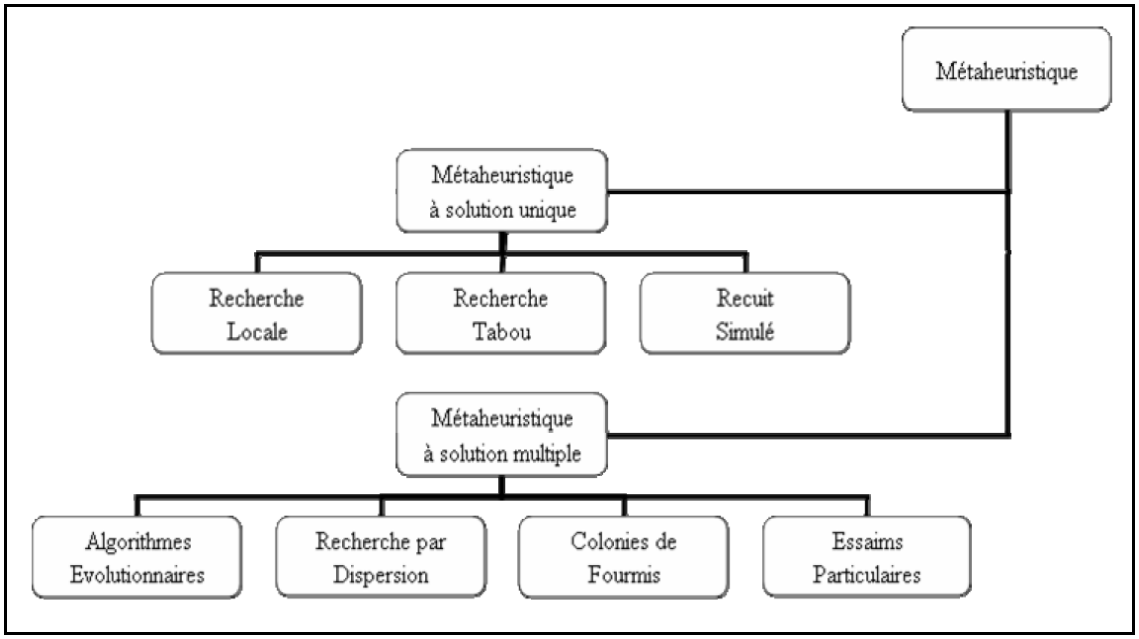


Figure 6.1 – Classification des métaheuristiques.

Finalement une « pseudo-classe » d'algorithmes a émergé ces dernières années pour résoudre des problèmes d'optimisation, qui contient des méthodes hybrides. Le principe consiste à combiner des algorithmes exacts et/ou des algorithmes approchés pour essayer de tirer profit des points forts de chaque approche et améliorer le comportement global de l'algorithme. Néanmoins, ces méthodes appartiennent forcément à l'une des classes de

méthodes de résolution vues précédemment. En fait, une méthode hybride est soit exacte (i.e. donne une solution optimale) ou bien approchée (i.e. donne une solution approchée).

6.2 Les Algorithmes Evolutionnaires

Parmi l'ensemble de techniques de recherche et d'optimisation, le développement des algorithmes évolutionnaires (AE) a été très important dans la dernière décennie. Les algorithmes évolutionnaires font partie du champ de l'Intelligence Artificielle (IA). Il s'agit d'IA de bas niveau, inspirée par " l'intelligence " de la nature. Les algorithmes évolutionnaires sont basés sur le concept de la sélection naturelle élaborée par Charles Darwin [Koza, 1999]. En effet, ces algorithmes adoptent une sorte d'évolution artificielle analogue à l'évolution naturelle. Le vocabulaire utilisé dans les AE est directement inspiré de celui de la théorie de l'évolution et de la génétique. En effet, nous trouvons des mots d'individus (solutions potentielles), de population, de gènes (variables), de chromosomes, de parents, de croisement, de mutations, etc., et nous nous appuierons constamment sur des analogies avec les phénomènes biologiques. Il s'agit de simuler l'évolution d'une population d'individus diverse à laquelle on applique différents opérateurs d'évolution comme les recombinaisons, les mutations, la sélection, etc (figure 6.2). Si la sélection s'opère en se basant sur une fonction d'adaptation, alors la population tend à s'améliorer. Un tel algorithme ne nécessite aucune connaissance du problème. En effet, on peut représenter celui-ci par une boîte noire comportant des entrées (les variables) et des sorties (les fonctions objectifs). L'algorithme ne fait que manipuler les entrées, lire les sorties, manipuler à nouveau les entrées de façon à améliorer les sorties, etc.

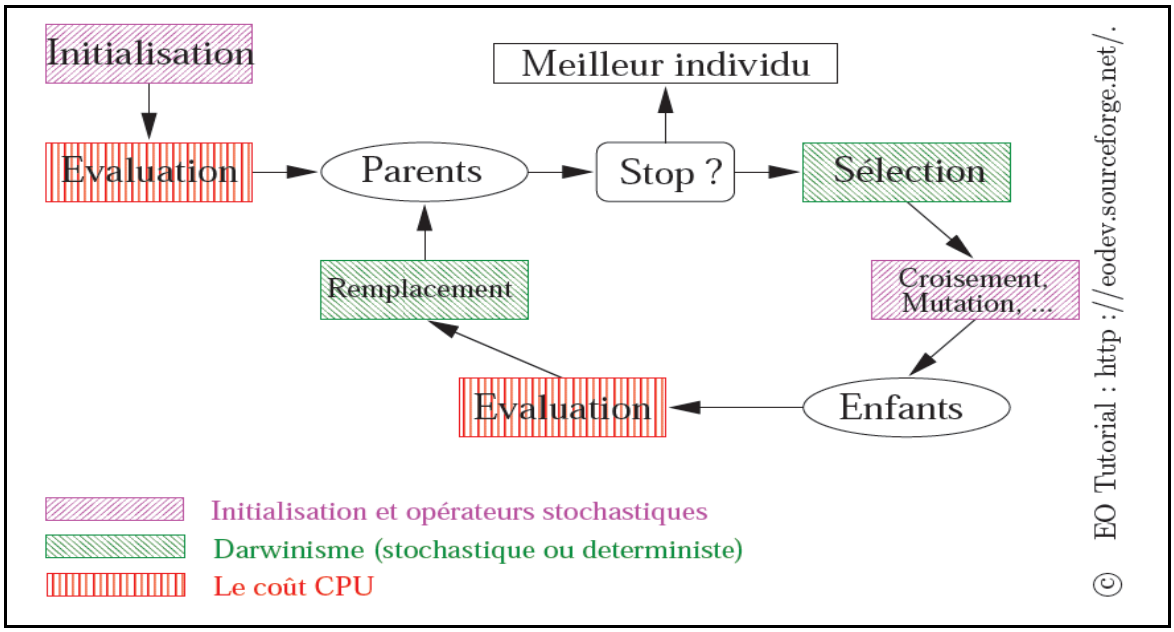


Figure 6.2 – Squelette d'un algorithme évolutionnaire.

On distingue quatre types d'AE (figure 5.3): les Algorithmes Génétiques (AG), les Stratégies d'Évolution (SE), la Programmation Évolutionnaire (PE), et la Programmation Génétique (PG). Dans les années 90, ces quatre champs ont commencé à sortir de leur isolement et ont été regroupés sous le terme anglo-saxon *d'Evolutionnary Computation*.

Le rapport entre la qualité de la solution finale et le temps d'exécution des métaheuristiques se diffère d'un algorithme à un autre, et dépend étroitement du problème à optimiser. Généralement, la fonction objectif a un grand impact sur la complexité de l'algorithme. En effet, le coût machine d'une optimisation est conditionné par le nombre d'évaluation de la fonction objectif. D'un autre coté, Les applications des métheuristques sont multiples : optimisation de fonctions numériques difficiles (discontinues, multimodales, bruitées…), traitement d'image (alignement de photos satellites, reconnaissance de suspects…), optimisation d'emplois du temps, optimisation de design, contrôle de systèmes industriels, apprentissage des réseaux de neurones, etc.

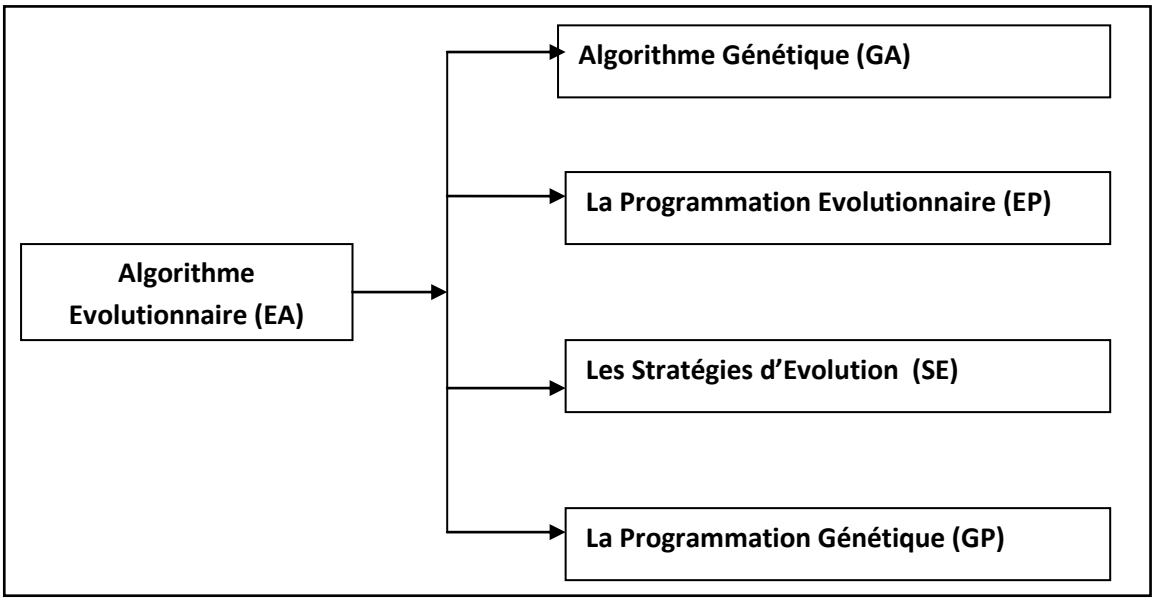


Figure 6.3 – Présentation de types des algorithmes évolutionnaire (EA).

6.3 Les algorithmes génétiques

Les algorithmes génétiques sont à la base des algorithmes d'optimisation stochastiques, mais peuvent également servir pour l'apprentissage automatique, par exemple. Les premiers travaux dans ce domaine ont commencé dans les années cinquante, lorsque plusieurs biologistes américains ont simulé des structures biologiques sur ordinateur. Puis, entre 1960 et 1970, John Holland sur la base des travaux précédents, développe les principes fondamentaux des algorithmes génétiques dans le cadre de l'optimisation mathématique. Malheureusement, les ordinateurs de l'époque n'étaient pas assez puissants pour envisager l'utilisation des algorithmes génétiques sur des problèmes réels de grande taille. La parution en 1989 de

l'ouvrage de référence écrit par D.E. Goldberg qui décrit l'utilisation de ces algorithmes dans le cadre de résolution de problèmes concrets a permis de mieux faire connaître ces derniers dans la communauté scientifique et a marqué le début d'un nouvel intérêt pour cette technique d'optimisation, qui reste néanmoins très récente. Parallèlement, des techniques proches ont été élaborées, dont on peut notamment citer la programmation génétique ou les stratégies évolutionnistes. Dans le même temps, la théorie mathématique associée aux algorithmes génétiques s'est développée mais reste encore bien limitée face à la complexité théorique induite par ces algorithmes.

6.3.1. Les éléments des algorithmes génétiques

Les algorithmes génétiques sont inspirés de la génétique classique et utilise le même vocabulaire. De manière générale, un algorithme génétique est constitué d'une population P de solutions appelées individus, dont l'adaptation à leur entourage est mesurée grâce à une fonction d'aptitude g qui retourne une valeur réelle, appelée fitness. Le principe général d'un AG consiste à simuler l'évolution d'une population d'individus jusqu'à atteindre un critère d'arrêt. Avant d'expliquer en détail le fonctionnement d'un algorithme génétique, nous allons présenter quelques mots de vocabulaire relatifs à la génétique. Ces mots sont souvent utilisés pour décrire un algorithme génétique :

Définition 1.8. (Gène) Un gène est une suite de bases azotées (adénine (A), cytosine (C), guanine (G) et la thymine (T)) qui contient le code d'une protéine donnée. On appellera gène la suite de symboles qui codent la valeur d'une variable. Dans le cas général, un gène correspond à un seul symbole (0 ou 1 dans le cas binaire). Une mutation changera donc systématiquement l'expression du gène muté.

Définition 1.9. (Chromosome) Un chromosome est constitué d'une séquence finie de gènes qui peuvent prendre des valeurs appelées allèles qui sont prises dans un alphabet qui doit être judicieusement choisi pour convenir du problème étudié.

Définition 1.10. (Individu) On appellera individu une des solutions potentielles. Dans la plupart des cas un individu sera représenté par un seul chromosome, dans ce cas, par abus de langage, on utilisera indifféremment individu et chromosome.

Définition 1.11. (Population) On appellera population l'ensemble des solutions potentielles qu'utilise l'AG.

Définition 1.12. (Génération) On appellera génération l'ensemble des opérations qui permettent de passer d'une population Pi à une population Pj. Ces opérations seront généralement : sélection des individus de la population courante, application des opérateurs génétiques, évaluation des individus de la nouvelle population.

Définition 1.13. (La fitness ou fonction d'évaluation) : La fonction de fitness est la pièce maitresse dans le processus d'optimisation. C'est l'élément qui permet aux algorithmes

génétiques de prendre en compte un problème donné. Pour que le processus d'optimisation puisse donner de bons résultats, il faut concevoir une fonction de fitness permettant une évaluation pertinente des solutions d'un problème sous forme chiffrée. Cette fonction est déterminée en fonction du problème à résoudre et du codage choisi pour les chromosomes. Pour chaque chromosome, elle attribue une valeur numérique, qui est supposée proportionnelle à la qualité de l'individu en tant que solution. Le résultat renvoyé par la fonction d'évaluation va permettre de sélectionner ou de refuser un individu selon une stratégie de sélection.

6.3.2. Principes généraux des algorithmes génétiques

Nous allons présenter d'une manière abstraite le principe et le fonctionnement des algorithmes génétiques. Les opérations successives utilisées dans les algorithmes génétiques sont illustrées sur la figure ci-après (figure 6.4).

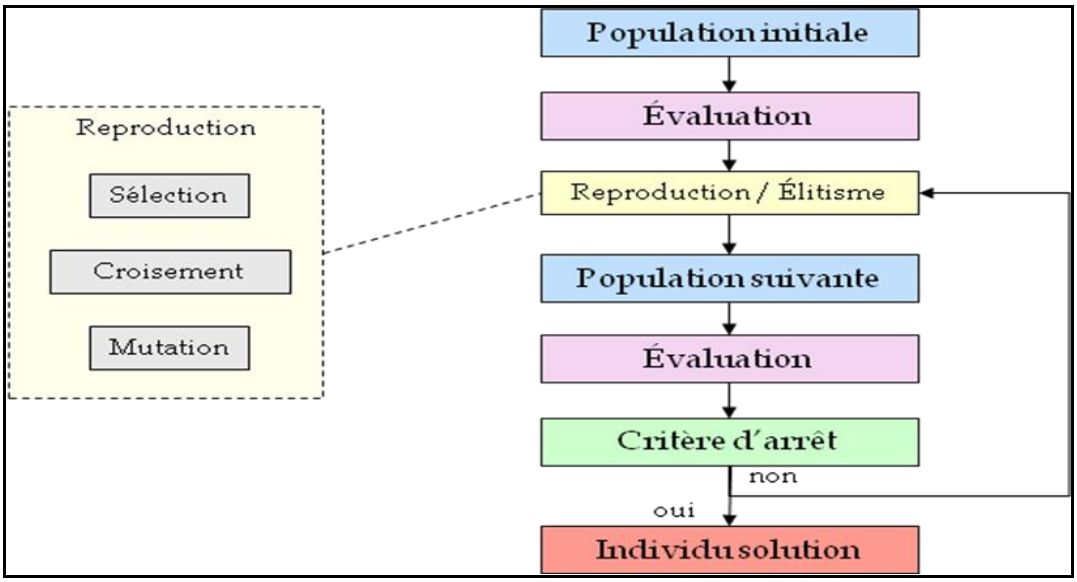


Figure 6.4– Les opérations successives utilisées dans les algorithmes génétiques.

L'algorithme génétique est constitué des étapes suivantes :

- Tout d'abord une population d'individus est générée de façon aléatoire.
- On procède à l'évaluation de l'ensemble des individus de la population initiale.
- On sélectionne un certain nombre d'individus dans la population, afin de produire une population intermédiaire, appelée aussi « mating pool ».
- On sélectionne deux chromosomes parents P1 et P2 en fonction de leurs adaptations. On applique aléatoirement l'opérateur de croisement avec une probabilité Pc pour générer deux chromosomes enfants C1 et C2. L'opération de croisement est suivie par l'opération de mutation avec une probabilité Pm, ce qui produit deux nouveaux individus C'1 et C'2

pour lesquels on évalue leurs fitness avant de les insérer dans la nouvelle population. Contrairement à la reproduction et au croisement qui favorisent l'intensification, cet opérateur favorise la diversification des individus. On réitère les opérations de sélection, de croisement et de mutation afin de compléter la nouvelle population, ceci termine le processus d'élaboration d'une génération.

- On réitère les opérations précédentes à partir de la seconde étape jusqu'à la satisfaction du critère d'arrêt.

De manière plus formelle, voici un algorithme génétique de base :

Algorithme 6.1 : algorithme génétique de base

Début

1: Générer une population aléatoire de n chromosomes.

2: Evaluer la fitness des chromosomes avec la fonction f (x)

3: **Répéter**

4: Calculer la fonction fitness f (x), pour tout chromosome x

5: Appliquer l'opération de sélection

6: Appliquer l'opération de croisement avec une probabilité PC

7: Appliquer l'opération de mutation avec une probabilité PM

8: Ajouter les nouveaux chromosomes à la nouvelle population

9: Calculer la fonction fitness f (x), pour tout chromosome x

10: Appliquer l'opération de remplacement

11 : **Jusqu'à** la satisfaction des conditions de terminaison

Fin

6.3.3. Codage

C'est une modélisation d'une solution d'un problème quelconque en un chromosome. Le choix du codage est important est souvent délicat. En effet, le choix du type de codage ne peut pas être effectué de manière évidente. La méthode actuelle à appliquer dans le choix du codage consiste à choisir celui qui semble le plus naturel en fonction du problème à traiter. L'objectif est bien sûr d'abord de pouvoir coder n'importe quelle solution. Mais il est souhaitable, au-delà de cette exigence, d'imaginer soit un codage tel que toute chaîne de caractères représente bien une solution réalisable du problème, soit un codage qui facilite ensuite la conception du croisement de telle sorte que les « enfants » obtenus à partir de la

recombinaison de leurs « parents » puissent être associés à des solutions réalisables, au moins pour un grand nombre d'entre eux. Parmi les codages les plus utilisés on peut citer.

6.3.3.1 Codage binaire

Ce type de codage est certainement le plus utilisé car il présente plusieurs avantages. Son principe est de coder la solution selon une chaîne de bits (les valeurs 0 ou 1). Le codage binaire a permis certes de résoudre beaucoup de problèmes, mais il s'est avéré obsolète pour certains problèmes d'optimisation numérique. Il est plus pratique d'utiliser un codage réel des chromosomes.

1	**0**	**0**	**1**	**1**	**0**	**1**

Figure 6.5 – codage binaire d'un chromosome (problème Max Sat).

6.3.3.2 Codage réel

Une autre approche semblable est de coder les solutions en tant que des suites de nombres entiers ou de nombres réels, avec chaque position représentant encore un certain aspect particulier de la solution. Cette approche tient compte d'une plus grande précision et de complexité que le codage basé sur les nombres binaires seulement. Ce type de codage est intuitivement plus proche à l'environnement des problèmes à résoudre.

3	**5**	**2**	**1**	**4**	**6**

Figure 6.6 – Codage binaire d'un chromosome
(Problème TSP).

6.3.3.3 Codage à l'aide de suite alphabétique

Une troisième approche est de représenter des individus dans AG comme une suite de caractères, où chaque caractère représente encore un aspect spécifique de la solution. Ce type de codage est utilisé dans de nombreux cas poussés d'algorithmes génétiques comme en bioinformatique.

A	**-**	**G**	**T**	**C**

Figure 6.7 – Codage alphabétique d'un chromosome
(Problème bioinformatique).

6.3.3.4 Codage sous forme d'arbre

Ce codage utilise une structure arborescente avec une racine de laquelle peuvent être issus un ou plusieurs fils. Un de leurs avantages est qu'ils peuvent être utilisés dans le cas de problèmes où les solutions n'ont pas une taille finie.

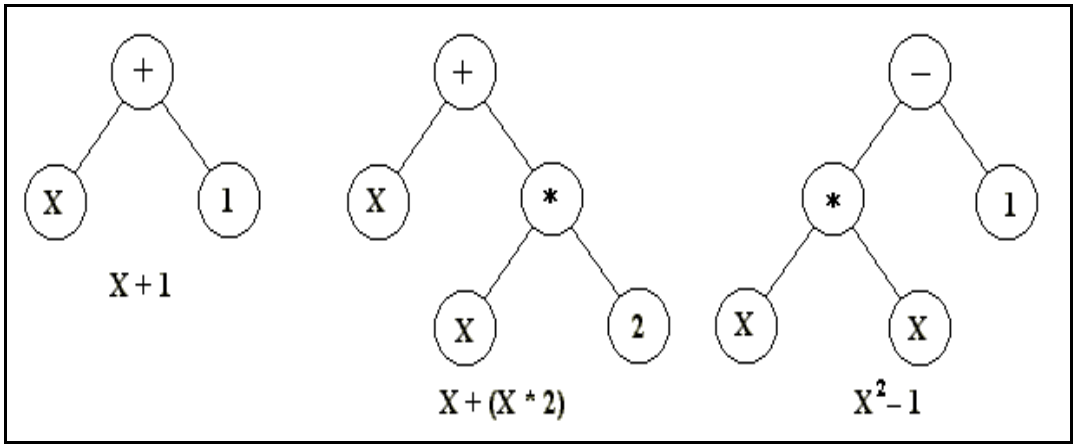


Figure 6.8 – Codage en arbre Trois arbres simples de programme de la sorte normalement utilisée dans la programmation génétique.

6.3.4. Les opérateurs d'un algorithme génétique

Quatre opérateurs caractérisent les algorithmes génétiques et rappellent l'origine de ces méthodes. Ils vont permettre à la population d'évoluer, par la création d'individus nouveaux construits à l'aide des individus anciens. Ces opérations sont : la sélection et le croisement, la mutation et le remplacement.

6.3.4.1 La sélection

La sélection permet d'identifier statistiquement les meilleurs individus de la population courante qui seront autorisés à se reproduire. Cette opération est fondée sur la performance des individus, estimée à l'aide de la fonction d'adaptation. Il existe différents principes de sélection :

- *Sélection par roulette (Wheel)*

Elle consiste à associer à chaque individu un segment dont la longueur est proportionnelle à sa fitness. Ces segments sont ensuite concaténés sur un axe gradué que l'on normalise entre 0 et 1 (figure 6.9). On tire alors un nombre aléatoire de distribution uniforme entre 0 et 1, puis on regarde quel est le segment sélectionné, et on reproduit l'individu correspondant. Avec cette technique, les bons individus seront plus souvent sélectionnés que les mauvais, et un même individu pourra avec cette méthode être sélectionné plusieurs fois. Néanmoins, sur des populations de petite taille, il est difficile d'obtenir exactement l'espérance mathématique de sélection à cause du faible nombre de tirages. Le cas idéal d'application de cette méthode est bien évidemment celui où la population est de taille infinie. On aura donc un biais de sélection

plus ou moins fort suivant la dimension de la population. Un autre problème rencontré lors de l'utilisation de la sélection par roulette est lorsque la valeur d'adaptation des chromosomes varie énormément. Si la meilleure fonction d'évaluation d'un chromosome représente 90% de la roulette alors les autres chromosomes auront très peu de chance d'être sélectionnés et on arriverait à une stagnation de l'évolution.

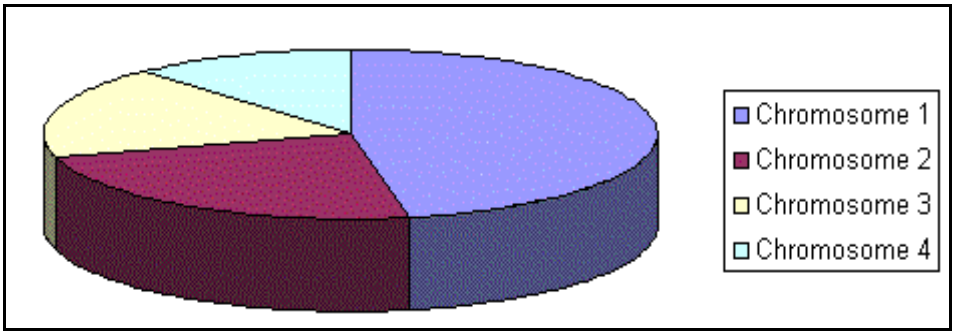


Figure 6.9 – Sélection par roulette.

- *Sélection par rang*

La sélection par rang est basée sur le trie de la population selon la fitness de chaque individu. Elle consiste à attribuer un entier de 1 à N pour une population de N chromosomes, du mauvais au meilleur. La sélection par rang d'un chromosome est similaire à la sélection par roulette, mais les proportions sont en relation avec le rang plutôt qu'avec la valeur de l'évaluation. Avec cette méthode de sélection, tous les chromosomes ont une chance d'être sélectionnés. Cependant, son grand inconvénient est la convergence lente vers la bonne solution. Ceci est dû au fait que les meilleurs chromosomes ne diffèrent pas énormément des plus mauvais. Vous pouvez voir dans la figure ci-après, comment la situation change après avoir transformé la fitness en rang.

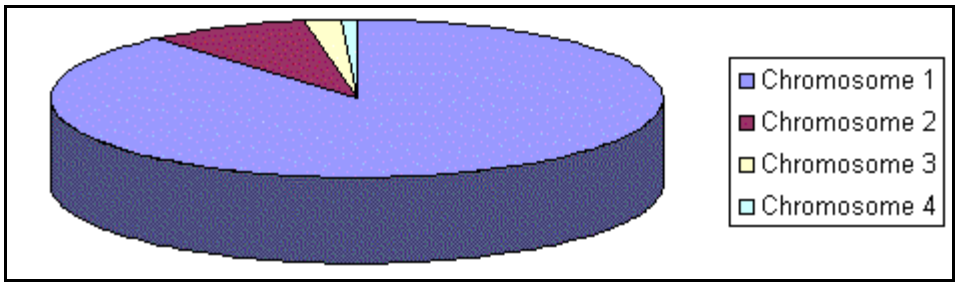
- *Sélection par tournoi*

Cette méthode est celle avec laquelle on obtient les résultats les plus satisfaisants. Elle consiste à choisir aléatoirement k individus (le nombre de participants à un tournoi) et à les confronter entre eux par le biais de la fonction d'adaptation, et de sélectionner ensuite le meilleur parmi eux. On répète ce processus autant de fois de manière à obtenir les n individus de la population qui serviront de parents. La variance de cette méthode est élevée et le fait d'augmenter ou de diminuer la valeur de *k* permet respectivement de diminuer ou d'augmenter la pression de la sélection.

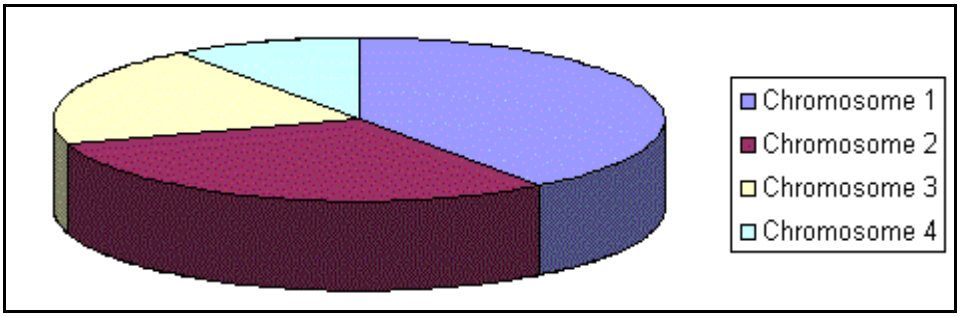
- *Sélection Steady-State*

Ce n'est pas une méthode particulière de sélection des chromosomes parents. L'idée principale est qu'une grande partie de la population puisse survivre à la prochaine génération.

L'algorithme génétique marche alors de la manière suivante. A chaque génération quelques chromosomes sont sélectionnés parmi ceux qui ont le meilleur coût, afin de créer des chromosomes fils. Ensuite les chromosomes les plus mauvais sont retirés et remplacés par les nouveaux. Le reste de la population survie à la nouvelle génération.



a. Situation avant le tri (graphe de fitnesses)



b. Situation après le tri (graph d'ordre)

Figure 6.10 – Sélection par rang.

- *Elitisme*

L'élitisme consiste à conserver à chaque génération un certain nombre des meilleurs chromosomes de la population qui pourraient disparaître par les opérations de mutation, croisement ou sélection. Elle consiste à copier un ou plusieurs des meilleurs chromosomes dans la nouvelle génération. Ensuite, on génère le reste de la population selon l'algorithme de reproduction usuel. Cette méthode améliore considérablement les algorithmes génétiques, car elle permet de ne pas perdre les meilleures solutions.

6.3.4.2 Le Croisement

Le croisement a pour but de produire une nouvelle génération d'individus en recombinant les individus sélectionnés par l'opérateur de sélection. C'est l'opérateur de l'algorithme génétique qui permet le plus souvent de se rapprocher de l'optimum d'une fonction en combinant les gènes. Ainsi, dans un premier temps, les individus sélectionnés sont répartis aléatoirement en couples de parents. Puis, chaque couple de parents subit une opération de

recombinaison afin de générer un ou deux enfants. Bien qu'il soit aléatoire, cet échange d'informations offre aux algorithmes génétiques une part de leur puissance : quelque fois, de "bons" gènes d'un parent viennent remplacer les "mauvais" gènes d'un autre et créent des fils mieux adaptés aux parents.

Le type de croisement le plus ancien est le croisement à découpages de chromosomes, ou *croisement 1-point* (figure 5.11 à gauche). Pour effectuer ce type de croisement sur des chromosomes constitués de L gènes, on tire aléatoirement une position inter-gènes dans chacun des parents. On échange ensuite les deux sous-chaînes de chacun des chromosomes ce qui produit deux enfants C1 et C2. Ce mécanisme présente l'inconvénient de privilégier les extrémités des individus. Et selon le codage choisi, il peut générer des fils plus ou moins proches de leurs parents. Pour éviter ce problème, on peut étendre ce principe en découpant le chromosome non pas en 2 sous-chaînes mais en 3, 4, etc. On parle alors de croisement *k-point ou multi-point* (figures 6.11 à droite).

L'autre type de croisement est le *croisement uniforme*. Le croisement uniforme consiste à générer un enfant en échangeant chaque gène des deux individus parents avec une probabilité uniforme égale à 0.5 (voir Figure 6.12). Cet opérateur peut être vu comme le cas extrême du croisement multi-point dans lequel le nombre de points de coupure est déterminé aléatoirement au cours de l'opération.

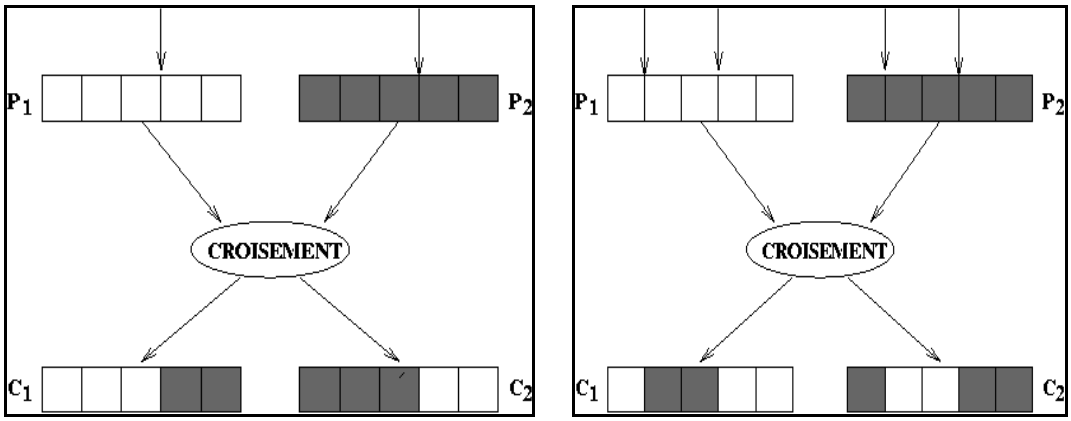


Figure 6.11 – L'opération de croisement k-point. À gauche Croisement 1-point.

Figure 6.12 – L'opération de croisement uniforme.

6.3.4.3 La mutation

Contrairement au croisement qui est un operateur d'exploitation, la mutation est principalement un opérateur d'exploration de l'espace de recherche. Le rôle de cet opérateur consiste à modifier aléatoirement, avec une certaine probabilité, la valeur d'un gène d'un chromosome (figure 6.13). Dans le cas du codage binaire, chaque bit $a_i \in \{0,1\}$, est remplacé selon une probabilité p_m par son complémentaire : $\tilde{a}_i = 1 - a_i$. Malgré, l'aspect marginal de la mutation dans la mesure où sa probabilité est en général fixée assez faible (de l'ordre de 1%), elle confère aux algorithmes génétiques une grande capacité d'exploration de tous les points de l'espace de recherche. Cet opérateur est donc d'une grande importance et il est loin d'être marginal. Comme on le voit dans la figure 6.14, sans mutation, la solution pourrait converger vers une solution locale optimale (les points jaunes). Une mutation réussie peut créer des solutions complètement aléatoires, conduisant à des solutions "inexplorées" qui ne peuvent être atteint via l'opération de croisement une fois que la population a convergée. L'opération de mutation a de fait un double rôle : celui d'effectuer une recherche locale et/ou éviter une stagnation autour d'optima locaux (figure 6.14). On distingue dans la littérature plusieurs types de mutations selon la nature de codage utilisé : mutation 1-bit, mutation réelle, etc.

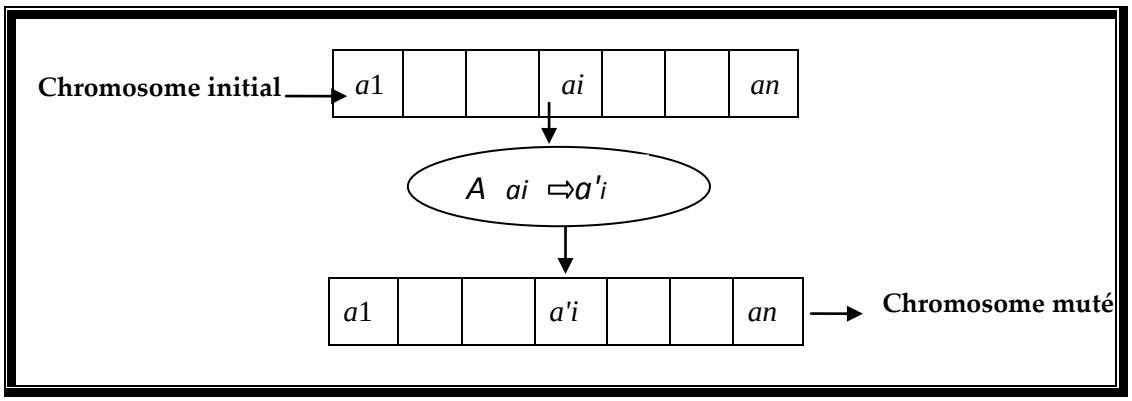


Figure 6.13 – Principe de l'opérateur de mutation.

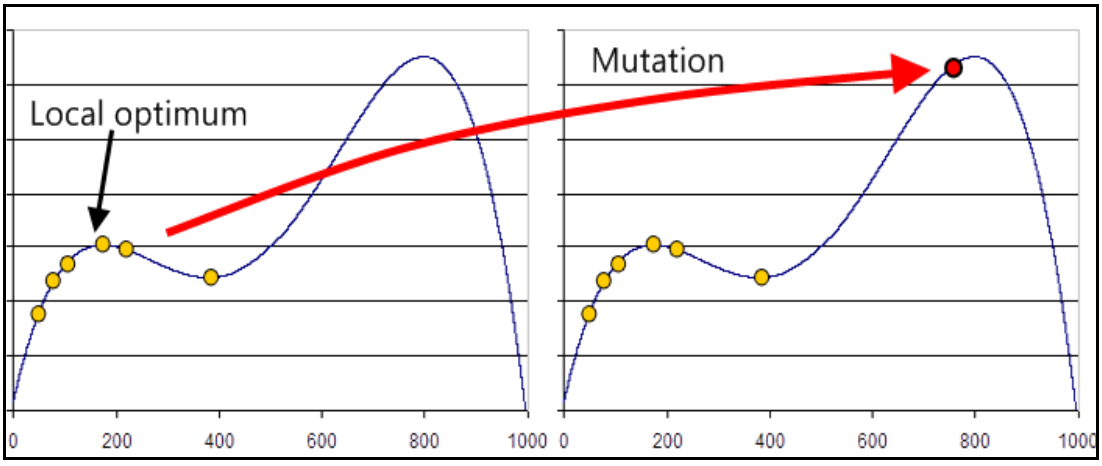


Figure 6.14– La diversification de l'opérateur de mutation.

6.3.4.3 Operateur de remplacement

Cet operateur est basé, comme l'operateur de sélection, sur la fitness des individus. Son travail consiste à déterminer les chromosomes parmi la population courante qui constituant la population de la génération suivante. Cette opération est appliquée après l'application successive des opérateurs de sélection, de croisement et de mutation. On trouve essentiellement 2 méthodes de remplacement différentes :

• *Le remplacement stationnaire :* dans ce cas, les enfants remplacent automatiquement les parents sans tenir compte de leurs performances respectives. Le nombre d'individus de la population est constant tout au long du cycle d'évolution, ce qui implique donc d'initialiser la population initiale avec un nombre suffisant d'individus.

• *Le remplacement élitiste :* dans ce cas, on garde au moins les individus possédant les meilleures performances d'une génération à la suivante. En général, on peut partir du principe qu'un nouvel individu (enfant) prend place au sein de la population que s'il remplit le critère d'être plus performant que le moins performant des individus de la population précédente. Donc les enfants d'une génération ne remplaceront pas nécessairement leurs parents comme dans le remplacement stationnaire et par conséquent la taille de la population n'est pas figée au cours du temps. Ce type de stratégie améliore les performances des algorithmes évolutionnaires dans certains cas. Mais présente aussi un désavantage en augmentant le taux de convergence prématuré.

6.3.5. Critères d'arrêt

Le critère d'arrêt indique que la solution est suffisamment approchée de l'optimum. Plusieurs critères d'arrêt de l'algorithme génétique sont possibles. On peut arrêter l'algorithme après un nombre de générations suffisant pour que l'espace de recherche soit convenablement exploré. Ce critère peut s'avérer coûteux en temps de calcul si le nombre d'individus à traiter dans chaque population est important. L'algorithme peut aussi être arrêté lorsque la population n'évolue plus suffisamment rapidement. On peut aussi envisager d'arrêter l'algorithme lorsque la fonction d'adaptation d'un individu dépasse un seuil fixé au départ. Nous pouvons également faire des combinaisons des critères d'arrêt précédents.

6.3.6. Les avantages et les limites des algorithmes génétiques

Un des grands avantages des algorithmes génétiques est qu'ils autorisent la prise en compte de plusieurs critères simultanément, et qu'ils parviennent à trouver de bonnes solutions sur des problèmes très complexes. Le principal avantage des AG par rapport aux autres techniques d'optimisation combinatoire consiste en une combinaison de :

- l'exploration de l'espace de recherche, basée sur des paramètres aléatoires, grâce à une recherche parallèle,

- l'exploitation des meilleures solutions disponibles à un moment donné.

Ils doivent simplement déterminer entre deux solutions quelle est la meilleure, afin d'opérer leurs sélections. Leur utilisation se développe dans des domaines aussi divers que l'économie, la bioinformatique ou la programmation des robots, etc.

L'inconvénient majeur des algorithmes génétiques est le coût d'exécution important par rapport à d'autres métaheuristiques. Les AG nécessitent de nombreux calculs, en particulier au niveau de la fonction d'évaluation. Mais avec les capacités calculatoire des ordinateurs récents, ce problème n'est pas grand. D'un autre côté, l'ajustement d'un algorithme génétique est délicat : des paramètres comme la taille de la population ou le taux de mutation sont parfois difficiles à déterminer. Or le succès de l'évolution en dépend et plusieurs essais sont donc nécessaires, ce qui limite encore l'efficacité de l'algorithme. Un autre problème important est celui des optima locaux. En effet, lorsqu'une population évolue, il se peut que certains individus qui, à un instant occupent une place importante au sein de cette population deviennent majoritaires. À ce moment, il se peut que la population converge vers cet individu et s'écarte ainsi d'individus plus intéressants mais trop éloignés de l'individu vers lequel on converge. Il faut mentionner également le caractère indéterministe des AGs. Comme les opérateurs génétiques utilisent des facteurs aléatoires, un AG peut se comporter différemment pour des paramètres et population identiques. Afin d'évaluer correctement l'algorithme, il faut l'exécuter plusieurs fois et analyser statistiquement les résultats.

6.4 Méthodes de recherche locale

En informatique, la recherche locale est une métaheuristique utilisée pour résoudre des problèmes dans plusieurs domaines comme l'intelligence artificielle, mathématiques, la recherche opérationnelle, l'ingénierie et la bioinformatique. Elle peut être utilisée sur des problèmes de recherche d'une solution maximisant (minimisant) un critère parmi un ensemble de solutions candidates. Ceci est dû à sa rapidité et à son principe relativement simple. Le principe de la recherche locale est le suivant : l'algorithme débute avec une solution initiale réalisable. Sur cette solution initiale, on applique une série de modifications locales (définissant un voisinage de la solution courante), tant que celles-ci améliorent la qualité de la fonction objectif. La figure suivante illustre bien le fonctionnement de l'algorithme général des méthodes de recherche locale.

Le passage d'une solution vers une autre se fait grâce à la définition de structure de voisinage qui est un élément très important dans la définition de ce type de méthode. Le voisinage d'une solution est défini en fonction du problème à résoudre. On définit l'espace de recherche comme l'espace dans lequel la recherche locale s'effectue. Cet espace peut correspondre à l'espace des solutions possibles du problème étudié. Habituellement, chaque solution candidate a plus d'une solution voisine, le choix de celle vers laquelle se déplacer est

pris en utilisant seulement l'information sur les solutions voisines de la solution courante, d'où le terme de recherche locale .

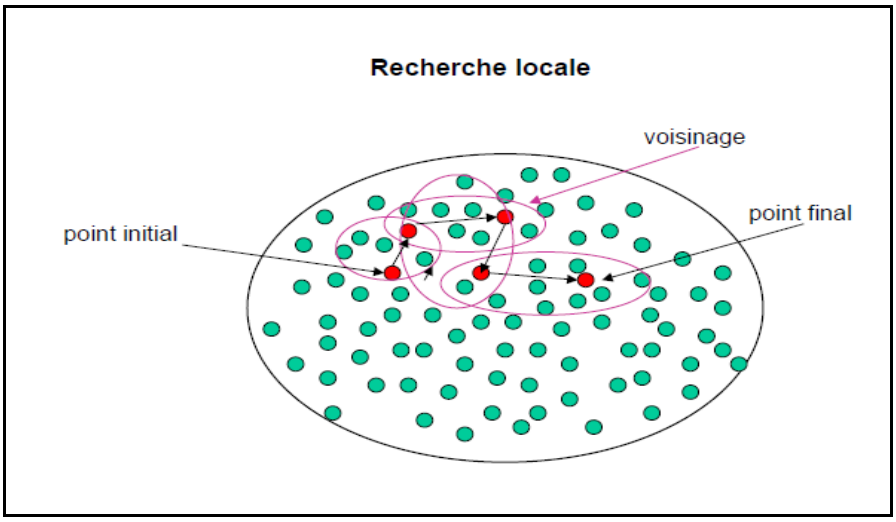


Figure 6.17 – Représentation de la procédure générale de la recherche locale.

D'une manière abstraite, une recherche locale peut se résumer de la façon suivante :

(i) Démarrer avec une solution.

(ii) Améliorer la solution.

(iii) Répéter le processus jusqu'à la satisfaction des critères d'arrêt.

Les méthodes de recherche locales utilisent principalement trois fonctions: une fonction d'évaluation, une fonction de voisinage et une fonction d'amélioration basée sur la fonction d'évaluation. D'un autre côté, le critère d'arrêt de la recherche locale peut être une limite en durée, une autre possibilité est de s'arrêter quand la meilleure solution trouvée par l'algorithme n'a pas été améliorée depuis un nombre donné d'itérations. Par ailleurs, les algorithmes de recherche locale sont des algorithmes sous-optimaux puisque la recherche peut s'arrêter alors que la meilleure solution trouvée par l'algorithme n'est pas la meilleure de l'espace de recherche. Cette situation peut se produire même si l'arrêt est provoqué par l'impossibilité d'améliorer la solution courante car la solution optimale peut se trouver loin du voisinage des solutions parcourues par l'algorithme. En effet, la plupart des méthodes de recherche locale utilisent une seule structure de voisinage ce qui rend très difficile l'optimisation de problèmes contenant plusieurs minima locaux. On trouve dans la littérature un grand nombre de méthodes de recherche locale, nous évoquerons dans la suite trois méthodes générales, souvent classées dans la catégorie des métaheuristiques qui sont : méthode de la descente, le recuit simulé et la recherche tabou. La procédure générale de la recherche locale est décrite par l'algorithme ci-dessous. Les bases de la recherche locale peuvent être définies ainsi. Soient :

– f () la fonction qu'on l'on souhaite maximiser,
– S la solution courante,
– S* la meilleure solution connue,
– N(S) le voisinage de S

Algorithme 6.2 : Algorithme de recherche locale simple

Étape 1 (initialisation)
a) choisir une solution initial s ∈S.
b) s* ←s (i.e. s* mémorise la meilleure solution trouvée)
Étape 2 *(choix et terminaison)*
a) *choisir* s'∈ N(s)
b) s ←s' (i.e. remplacer s par s')
c) fin si la condition d'arrêt est vérifiée
Étape 3 *(mise à jour)*
a) s* ←s si f(s) < f(s*)
b) aller à l'Étape 2

6.4.1. La Méthode de la Descente (Hill-Climbing)

Cette méthode appelée aussi Hill-Climbing est assez ancien, mais d'une grande simplicité. Elle consiste à se déplacer dans l'espace de recherche en choisissant toujours la meilleure solution parmi le voisinage de la solution courante. On part d'une solution si possible bonne 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 (L'algorithme 6.2). La convergence vers un optimum local pouvant être très lente, on peut éventuellement fixer le nombre d'itérations maximum, si on veut limiter le temps d'exécution. Cette méthode a l'inconvénient de rester bloquée dans un optimum local. En effet, une fois un optimum local trouvé, on s'arrête, même si ce n'est pas l'optimum global. Selon le paysage des solutions, l'optimum local peut être très bon ou très mauvais par rapport à l'optimum global. Si la solution de départ est donnée par une heuristique déterministe, l'algorithme sera déterministe. Si elle est tirée au hasard, on a un algorithme non déterministe et donc plusieurs exécutions différentes sur la même instance pourront donner des solutions différentes et de qualités différentes.

Il est tout à fait clair que dans ce type de méthodes, la notion de voisinage est primordiale. Si les voisins sont très nombreux, on a de fortes chances de trouver l'optimum global. Malgré cela, visiter un voisinage peut être très long par ce qu'on visitera une grande partie de l'espace des solutions. D'un autre côté, si le voisinage est très restreint, on risque fort de rester bloqué

dans un optimum local de mauvaise qualité. Par conséquent, le choix de la notion de voisinage est un compromis entre efficacité et qualité.

Algorithme 6.3 : Schéma général du Hill-Climbing

```
Début
Générer et évaluer une solution initiale s
 Tant que La condition d'arrêt n'est pas vérifiée faire
  Modifier s pour obtenir s' et évaluer s'
  si (s' est meilleure que s) alors
   Remplacer s par s'
  finsi
 Fin tant que
 retourner s
Fin
```

On distingue différents types de descente en fonction de la stratégie de génération de la solution de départ et du parcours du voisinage : la descente déterministe, la descente stochastique et la descente vers le premier meilleur. On peut par exemple remarquer que dans la formulation précédente, il n'est pas mentionné que l'aléatoire est mis en jeu pour la génération de la solution initiale ou sa modification. L'algorithme 6.4 décrit la version stochastique *Random Hill-Climbing* (RHC). Il faut alors définir un pas de recherche qui détermine la taille du voisinage pouvant être exploré.

Algorithme 6.4 : Schéma général du Random Hill-Climbing

```
Début
 Générer aléatoirement une solution initiale s et l'évaluer
 tant que La condition d'arrêt n'est pas vérifiée faire
  Modifier aléatoirement s pour obtenir s' et évaluer s'
  si (s' est meilleure que s) alors
   Remplacer s par s'
  fin si
 fin tantque
 retourner s
Fin
```

La méthode hill-climbing possède plusieurs avantages. C'est une méthode intuitive, simple à implémenter par rapport à d'autres méthodes de recherche locale. Typiquement, Il n'existe pas de notion d'arbre de recherche ni de retour en arrière. La méthode hill-climbing donne souvent de bonnes solutions. Cependant, cette méthode présente quelques limitations comme le problème du minimum local et la lenteur de la méthode.

6.4.2 Le recuit simulé

La méthode du Recuit Simulé (RS) est une technique de recherche locale inspirée d'un processus utilisé en métallurgie. Ce processus alterne des cycles de refroidissement lent et de réchauffage ou de recuit qui tendent à minimiser l'énergie du matériau. Le recuit simulé s'appuie sur l'algorithme de Metropolis-Hastings, qui permet de décrire l'évolution d'un système thermodynamique. Par analogie avec le processus physique, la fonction à minimiser deviendra l'énergie E du système. On introduit également un paramètre fictif, la température T du système.

L'algorithme du recuit simulé part d'une solution donnée, et la modifie itérativement jusqu'à le refroidissement du système. Les solutions trouvées peuvent améliorer le critère que l'on cherche à optimiser, on dit alors qu'on a fait baisser l'énergie du système, comme elles peuvent les dégrader. Si on accepte une solution améliorant le critère, on tend ainsi à chercher l'optimum dans le voisinage de la solution de départ. Contrairement autres méthodes de recherche locale, le recuit simulé peut accepter des solutions de qualité moindre en fonction de la dégradation de la solution considérée. Le schéma général de l'algorithme RS est décrit par l'algorithme suivant :

Algorithme 6.5 : Schéma général du Recuit simulé

```
Engendrer une configuration initiale S0 de S ; S := S0
Initialiser T en fonction du schéma de refroidissement
Répéter
    -Engendrer un voisin aléatoire S' de S
    -Calculer Δ = f(S') − f(S)
    -Si CritMetropolis (Δ, T), alors S : = S'
    -Mettre T à jour en fonction du schéma de refroidissement
Jusqu'à <condition fin>
Retourner la meilleure configuration trouvée
```

L'acceptation d'une mauvaise solution permet alors d'explorer une plus grande partie de l'espace de solution et tend à éviter de s'enfermer trop vite dans la recherche d'un optimum

local. Afin de fixer le niveau de dégradation acceptable à une itération donnée le paramètre de température T est utilisé. Ce paramètre décroît au cours du temps. Néanmoins, Une difficulté spécifique à l'algorithme du recuit simulé est l'ajustement du paramètre de la température T. En effet, Un refroidissement trop rapide mènerait vers un optimum local pouvant être de très mauvaise qualité. Un refroidissement trop lent serait très coûteux en temps de calcul. Le nouvel état est accepté si l'énergie du système diminue, sinon il est accepté avec une probabilité. Le recuit simulé possède plusieurs applications dans des domaines différents : Traitement d'images, Problème d'ordonnancement, Bioinformatique, etc. Le recuit simulé donne généralement des solutions de bonnes qualités. C'est une méthode générale et facile à programmer, Souple d'emploi (de nouvelles contraintes peuvent être facilement incorporées). Cependant, les inconvénients principaux sont le nombre important de paramètres à ajuster et le temps de calcul excessif dans certaines applications.

6.4.3 La recherche Tabou (tabu search)

La recherche tabou est une méthode de recherche locale avancée qui fait appel à un ensemble de règles et de mécanismes généraux pour guider la recherche de manière intelligente. Le principe de la recherche tabou est d'explorer le voisinage et choisir la position dans ce voisinage qui minimise la fonction objectif à partir d'une position donnée. Lorsque tous les points du voisinage ont une valeur plus élevée, cette opération peut conduire à augmenter la valeur de la fonction. On peut ainsi repérer les minima locaux. Pour éviter à l'étape suivante de retomber dans un minima local, il faut stocker en mémoire les dernières positions explorées d'où le nom tabou. Les positions sont enregistrées dans une pile FIFO de taille paramétrable. Toutefois, la mémorisation des solutions complètes rend la gestion de la liste Tabou trop coûteuse en temps de calcul et en place mémoire; particulièrement si le nombre de variables est très élevé. On peut réussir à pallier ce problème en ne stockant que les mouvements et la valeur de la fonction à minimiser. Il a été démontré pour la recherche tabou que la convergence existe dans des conditions strictes rarement mis en pratique. On notera qu'il existe de nombreuses variantes aussi bien au niveau de la définition du voisinage que de la façon de gérer la mémoire. La recherche tabou possède plusieurs avantages. En effet, elle est une méthode rapide, facile à implémenter, donne souvent de bonnes solutions, et possède un fonctionnement intuitif. Cependant, la recherche tabou possède certaines limitations comme le manque de modélisation mathématiques. En effet, chaque cas est différent et il faut donc adapter la recherche à chaque problème particulier. D'un autre le grand nombre de paramètre à régler rend l'utilisation de la méthode tabou difficile. Finalement, la recherche tabou a été appliquée avec succès pour résoudre de nombreux problèmes difficiles d'optimisation combinatoire : problèmes de routage de véhicules, problèmes d'affectation quadratique, problèmes d'ordonnancement, problèmes de coloration de graphes, etc. L'algorithme général de la recherche tabou est le suivant :

Algorithme 6.6 : Schéma général de la recherche tabou

```
Début
  S = solution initiale ;
  Best = S ;
  Répéter
     S_best = meilleur voisin de S ; //le meilleur voisin est choisi
     si (fitness(S_best) < fitness (Best)) Alors
        Best = Sbest ;
     Mise à jour de la liste Tabou
     S = Sbest ;
  Jusqu'à(critère de fin est vérifié)
  Retourner Best ;
Fin
```

6.4.4. Discussion des méthodes de recherche locale

Bien que les méthodes de recherche locale ont démontré leurs efficacités dans plusieurs problèmes d'optimisation, la difficulté essentielle liée à l'utilisation de ces méthodes consiste à réussir une exploration de l'espace de recherche sans stagner dans des minima locaux. La recherche se trouve dans un minimum local lorsque plus aucun des voisins de l'affectation courante ne l'améliore, par exemple, dans la figure 6.18, on observe plusieurs minima locaux. Pour sortir de ces minima, les algorithmes doivent avoir la capacité de détériorer l'affectation courante afin de l'améliorer ultérieurement en atteignant l'optimum global. Cette technique est connue sous le nom du *chemin aléatoire* (Random Walk). Elle consiste à effectuer, de temps en temps, un mouvement aléatoire pour diversifier la recherche et ainsi espérer se rapprocher de la bonne solution. Le fait d'accepter la détérioration de l'affectation courante permet de sortir de certains minima locaux mais peut induire un allongement du temps d'exécution. On trouve une technique similaire à celle-ci dans l'algorithme de recuit simulé. Une autre technique couramment utilisée est celle de la *relance* (Multiple Start). Elle consiste à répéter une nouvelle recherche à partir d'une autre solution initiale appartenant à une autre région de l'espace de recherche. Cette technique peut être réitérée plusieurs fois dans l'espérance d'atteindre différentes zones prometteuses de l'espace de recherche.

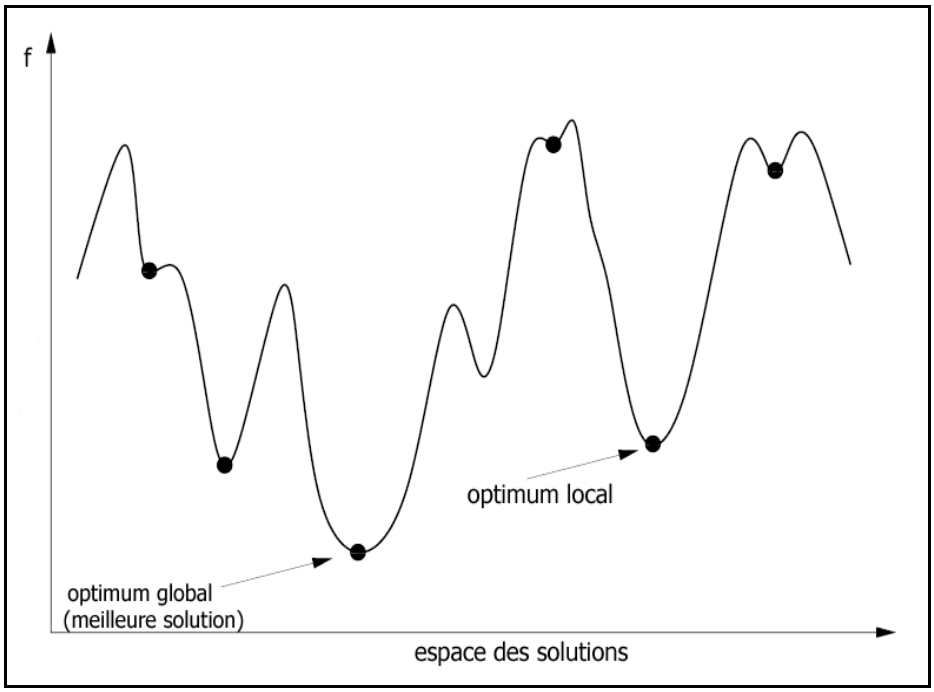


Figure 6.18– Le problème de minimum local dans la recherche locale.

6.5 Méthodes hybrides

L'hybridation est une tendance observée dans de nombreux travaux réalisés sur les métaheuristiques ces dix dernières années. Elle exploite la puissance de plusieurs algorithmes, et les combine en un seul méta-algorithme. Il existe plusieurs manières pour faire cette hybridation.

Une des techniques les plus populaires d'hybridation concerne l'utilisation de métaheuristiques à base d'une solution unique avec des métaheuristiques à population. La plupart des applications réussies des métaheuristiques à population sont complétées par une phase de recherche locale, afin d'augmenter leurs capacités d'intensification. Les individus d'une population tentent d'explorer l'espace de solutions afin de trouver les zones prometteuses, lesquelles sont ensuite explorées plus en détail par des méthodes de recherche locale comme la recherche tabou ou le hill-climbing, ou par un algorithme de type 2-opt. Il est à noter également que les deux types de recherche, globale et locale, peuvent être lancées de façon asynchrone, sur des processeurs différents afin d'augmenter la vitesse d'exécution de l'algorithme hybride.

Une deuxième manière d'hybrider consiste à lancer en parallèle l'exécution de la même métaheuristique, voire même plusieurs fois la même métaheuristique mais avec des paramètres différents. Ces processus parallèles communiquent entre eux régulièrement pour échanger de l'information sur leurs résultats partiels. Presque toutes les métaheuristiques classiques ont donné naissance à des versions parallèles plus ou moins fidèles à leur origine.

Enfin une troisième forme d'hybridation combine les métaheuristiques avec des méthodes exactes. Une méthode exacte peut être utilisée pour la génération du meilleur voisin d'une solution. On peut par exemple utiliser la méthode branch-and-bound sur une partie du problème de petite taille pour générer des solutions de bonnes qualités. Par ailleurs, une métaheuristique peut être utilisée pour fournir des bornes à une méthode exacte de type branch-and-bound.

Les algorithmes hybrides sont sans aucun doute parmi les méthodes les plus puissantes. Malheureusement, les temps de calcul nécessaires peuvent devenir prohibitifs à cause du nombre d'individus manipulés dans la population. Une voie pour résoudre ce problème est la parallélisation de ces algorithmes sur des machines parallèles ou sur des systèmes distribués. En effet, le coût de calcul des méthodes hybrides peut être supporté avec le recours aux grilles de calcul.

Bibliographie

[Baeck et al., 1997] Baeck T., Fogel D.B., and Michalewicz Z. Handbook of Evolutionary Computation. *Institute of Physics Publishing and Oxford University Press*, 1997.

[Baeck et al., 2000] Baeck T., Fogel D.B., and Michalewicz Z., editors. Evolutionary Computation: Advanced Algorithms and Operators. *Institute of Physics Publishing*, Bristol, 2000.

[Goldberg, 1989] Goldberg, D. E. Genetic algorithms in search, optimization, and machine learning. Addison-Wesley, 1989.

[Goldberg, 1994] Goldberg G.E., Algorithmes génétiques, Éditions Addison-Wesley, Paris, 1994.

[Moscato et al, 2005] Moscato, P., and Cotta, C. A Gentle Introduction to Memetic Algorithms. Operations Research & Management Science 57(2), 105–144, 2005.

[Moscato,1989] Moscato, P.: On Evolution, Search, Optimization, Genetic Algorithms and Martial Arts: Towards Memetic Algorithms. Caltech Concurrent Computation Program (report 826),1989.

[Palpant, 2005] Palpant M. : Recherche exacte et approchée en optimisation combinatoire : schémas d'intégration et applications. Thèse de Doctorat, Université d'Avignon, 2005.

[Heudin, 1994] Heudin J-C, La Vie artificielle, *éditions Hermès Science*, Paris, 1994.

[Layeb, 2009] Layeb,A. working Thesis, 2009.

Chapitre 6 : Programmation par contraintes

La totalité de ce chapitre est prise du cours de christine solnon

1 - Qu'est-ce qu'une contrainte ?

Une contrainte est une relation logique (une propriété qui doit être vérifiée) entre différentes inconnues, appelées variables, chacune prenant ses valeurs dans un ensemble donné, appelé domaine. Ainsi, une contrainte restreint les valeurs que peuvent prendre simultanément les variables. Par exemple, la contrainte "*x + 3*y = 12*" restreint les valeurs que l'on peut affecter simultanément aux variables *x* et *y*.

1.1 - Quelques caractéristiques des contraintes

Une contrainte est ***relationnelle*** : elle n'est pas "dirigée" comme une fonction qui définit la valeur d'une variable en fonction des valeurs des autres variables.

Ainsi, la contrainte "*x - 2*y = z*" permet de déterminer *z* dès lors que *x* et *y* sont connues, mais aussi *x* dès lors que *y* et *z* sont connues et *y* dès lors que *x* et *z* sont connues.

Notons également qu'une contrainte est ***déclarative*** : elle spécifie quelle relation on doit retrouver entre les variables, sans donner de procédure opérationnelle pour effectivement assurer/vérifier cette relation.

Ainsi, lorsqu'on pose la contrainte "*x - 2*y = z*", on ne s'occupe pas de donner un algorithme permettant de résoudre cette équation.

Notons enfin que ***l'ordre dans lequel sont posées les contraintes n'est pas significatif*** : la seule chose importante à la fin est que toutes les contraintes soient satisfaites (...cependant, dans certains langages de programmation par contraintes l'ordre dans lequel les contraintes sont ajoutées peut avoir une influence sur l'efficacité de la résolution.

1.2 - Définition d'une contrainte

Une contrainte est une relation entre différentes variables. Cette relation peut être définie en *extension* ou en *intension* :

- Pour définir une contrainte en extension, on énumére les tuples de valeurs appartenant à la relation.

Par exemple, si les domaines des variables *x* et *y* contiennent les valeurs *0, 1* et *2*, alors on peut définir la contrainte *"x est plus petit que y"* en extension par *"(x=0 et y=1) ou (x=0 et y=2) ou (x=1 et y=2)"* , ou encore par *"(x,y) élément-de {(0,1),(0,2),(1,2)}"*

- Pour définir une contrainte en intention, on utilise des propriétés mathématiques connues.

Par exemple : *"x < y"* ou encore *"A et B => non(C)"*

1.3 - Arité d'une contrainte

L'arité d'une contrainte est le nombre de variables sur lesquelles elle porte. On dira que la contrainte est

- ***unaire*** si son arité est égale à 1 (elle ne porte que sur une variable),

par exemple "*x*x = 4*" ou encore "*est-un-triangle(y)*"

- ***binaire*** si son arité est égale à 2 (elle met en relation 2 variables),

par exemple "$x \neq y$" ou encore "*A U B = A*"

- ***ternaire*** si son arité est égale à 3 (elle met en relation 3 variables),

par exemple "*x+y < 3*z-4*" ou encore "*(non x) ou y ou z = vrai*"

- ***n-aire*** si son arité est égale à *n* (elle met en relation un ensemble de *n* variables). On dira également dans ce cas que la contrainte est globale.

Par exemple, une contrainte globale courante (et très pratique) est la contrainte "*toutesDifférentes(E)*", où *E* est un ensemble de variables, qui contraint toutes les variables appartenant à *E* à prendre des valeurs différentes.

1.4 - Différents types de contraintes

On distingue différents types de contraintes en fonction des domaines de valeurs des variables:

- Les ***contraintes numériques***, portant sur des variables à valeurs numériques : une contrainte numérique est une égalité (=) , une différence (≠) ou une inégalité (<, ≤, >, ≥) entre 2 expressions arithmétiques.

  On distingue :

  - les ***contraintes numériques sur les réels***, quand les variables de la contrainte peuvent prendre des valeurs réelles

par exemple une contrainte physique comme "*U = R*I*"

- et les ***contraintes numériques sur les entiers***, quand les variables de la contrainte ne peuvent prendre que des valeurs entières

par exemple une contrainte sur le nombre de personnes pouvant embarquer dans un avion.

- On distingue également :
  - les ***contraintes numériques linéaires***, quand les expressions arithmétiques sont linéaires

par exemple "*4*x - 3*y + 8*z < 10*"

  - et les ***contraintes numériques non linéaires*** , quand les expressions arithmétiques contiennent des produits de variables, ou des fonctions logarithmiques, exponentielles...

par exemple "*x*x = 2*" ou "*sinus(x) + z*log(y) = 4*".

- Les ***contraintes booléennes***, portant sur des variables à valeur booléenne (vrai ou faux) : une contrainte booléenne est une implication (=>), une équivalence (<=>) ou une non équivalence (<≠>) entre 2 expressions logiques.

Par exemple "*(non a) ou b => c*" ou encore "*non (a ou b) <=> (c et d)*".

- Les ***contraintes de Herbrand***, portant sur des variables à valeur dans l'univers de Herbrand : une contrainte de Herbrand est une égalité (=) ou diségalité (≠) entre 2 termes (appelés aussi arbres) de l'univers de Herbrand. En particulier, unifier deux termes Prolog revient à poser une contrainte d'égalité entre eux.

Par exemple l'unification de *"f(X,Y)"*avec *"f(g(a),Z)"* s'exprime par la contrainte *"f(X,Y) = f(g(a),Z)"*.

- ... et bien d'autres encore, comme les contraintes sur les ensembles, ... que nous ne détaillerons pas ici.

---

2 - Qu'est ce qu'un Problème de Satisfaction de Contraintes (CSP) ?

2.1 - Définition d'un CSP

Un CSP (Problème de Satisfaction de Contraintes) est un problème modélisé sous la forme d'un ensemble de contraintes posées sur des variables, chacune de ces variables prenant ses valeurs dans un domaine. De façon plus formelle, on définira un CSP par un triplet *(X,D,C)* tel que

- *X = { X1, X2, ..., Xn}* est l'ensemble des variables (les inconnues) du problème ;
- *D* est la fonction qui associe à chaque variable *Xi* son domaine *D(Xi),* c'est-à-dire l'ensemble des valeurs que peut prendre *Xi ;*

- *C = {C1, C2, ..., Ck}* est l'ensemble des contraintes. Chaque contrainte *Cj* est une relation entre certaines variables de *X*, restreignant les valeurs que peuvent prendre simultanément ces variables.

Par exemple, on peut définir le CSP *(X,D,C)* suivant :

- $X = \{a,b,c,d\}$
- $D(a) = D(b) = D(c) = D(d) = \{0,1\}$
- $C = \{ a \neq b, c \neq d, a+c < b \}$

Ce CSP comporte 4 variables *a*, *b*, *c* et *d*, chacune pouvant prendre 2 valeurs (*0* ou *1*). Ces variables doivent respecter les contraintes suivantes :*a* doit être différente de *b* ; *c* doit être différente de *d* et la somme de *a* et *c* doit être inférieure à *b*.

2.2 - Solution d'un CSP

Etant donné un CSP *(X, D, C)*, sa résolution consiste à affecter des valeurs aux variables, de telle sorte que toutes les contraintes soient satisfaites. On introduit pour cela les notations et définitions suivantes :

- On appelle ***affectation*** le fait d'instancier certaines variables par des valeurs (évidemment prises dans les domaines des variables). On notera *A = { (X1,V1), (X2,V2), ..., (Xr,Vr) }* l'affectation qui instancie la variable *X1* par la valeur *V1*, la variable *X2* par la valeur *V2*, ..., et la variable *Xr* par la valeur *Vr*.

Par exemple, sur le CSP précédent, *A={(b,0),(c,1)}* est l'affectation qui instancie *b* à *0* et *c* à *1*.

- Une affectation est dite ***totale*** si elle instancie toutes les variables du problème ; elle est dite ***partielle*** si elle n'en instancie qu'une partie.

Sur notre exemple, *A1 = {(a,1),(b,0),(c,0),(d,0)}* est une affectation totale ; *A2 = {(a,0),(b,0)}* est une affectation partielle.

- Une affectation *A* ***viole*** une contrainte *Ck* si toutes les variables de *Ck* sont instanciées dans *A*, et si la relation définie par *Ck* n'est pas vérifiée pour les valeurs des variables de *Ck* définies dans *A*.

Sur notre exemple, l'affectation partielle *A2 = {(a,0),(b,0)}* viole la contrainte $a \neq b$; en revanche, elle ne viole pas les deux autres contraintes dans la mesure où certaines de leurs variables ne sont pas instanciées dans *A2*.

- Une affectation (totale ou partielle) est ***consistante*** si elle ne viole aucune contrainte, et ***inconsistante*** si elle viole une ou plusieurs contraintes.

Sur notre exemple, l'affectation partielle *{(c,0),(d,1)}* est consistante, tandis que l'affectation partielle *{(a,0),(b,0)}* est inconsistante.

- Une ***solution*** est une affectation totale consistante, c'est-à-dire une valuation de toutes les variables du problème qui ne viole aucune contrainte.

Sur notre exemple, *A = {(a,0),(b,1),(c,0),(d,1)}* est une affectation totale consistante : c'est une solution.

2.3 - Notion de CSP surcontraint ou souscontraint

Lorsqu'un CSP n'a pas de solution, on dit qu'il est *surcontraint* : il y trop de contraintes et on ne peut pas toutes les satisfaire. Dans ce cas, on peut souhaiter trouver l'affectation totale qui viole le moins de contraintes possibles.

Un tel CSP est appelé *max-CSP* (max car on cherche à maximiser le nombre de contraintes satisfaites).

Une autre possibilité est d'affecter un poids à chaque contrainte (une valeur proportionnelle à l'importance de cette contrainte, et de chercher l'affectation totale qui minimise la somme des poids des contraintes violées.

Un tel CSP est appelé *CSP valué* (*VCSP*).

Il existe encore d'autre types de CSPs, appelés *CSPs basés sur les semi-anneaux* (semiring based CSPs), permettant de définir plus finement des préférences entre les contraintes.

Inversement, lorsqu'un CSP admet beaucoup de solutions différentes, on dit qu'il est *sous-contraint*. Si les différentes solutions ne sont pas toutes équivalentes, dans le sens où certaines sont mieux que d'autres, on peut exprimer des préférences entre les différentes solutions. Pour cela, on ajoute une fonction qui associe une valeur numérique à chaque solution, valeur dépendante de la qualité de cette solution. L'objectif est alors de trouver la solution du CSP qui maximise cette fonction.

Un tel CSP est appelé *CSOP* (Constraint Satisfaction Optimisation Problem).

---

3 - Un premier exemple : le problème des reines

3.1 - Description du problème

Il s'agit de placer 4 reines sur un échiquier comportant 4 lignes et 4 colonnes, de manière à ce qu'aucune reine ne soit en prise. On rappelle que 2 reines sont en prise si elles se trouvent sur une même diagonale, une même ligne ou une même colonne de l'échiquier.

3.2 - Modélisation sous la forme d'un CSP

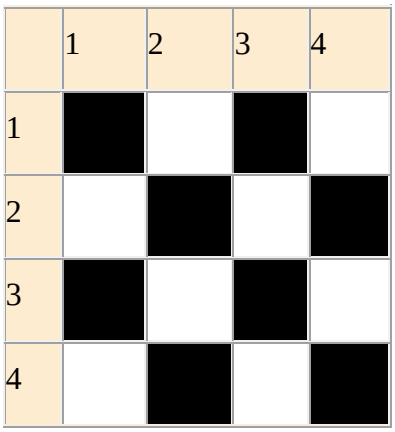
Pour modéliser un problème sous la forme d'un CSP, il s'agit tout d'abord d'identifier l'ensemble des variables *X* (les inconnues du problème), ainsi que la fonction *D* qui associe à chaque variable de *X* son domaine (les valeurs que la variable peut prendre). Il faut ensuite identifier les contraintes *C* entre les variables. Notons qu'à ce niveau, on ne se soucie pas de

savoir comment résoudre le problème : on cherche simplement à le spécifier formellement. Cette phase de spécification est indispensable à tout processus de résolution de problème ; les CSPs fournissent un cadre structurant à cette formalisation.

Un même problème peut généralement être modélisé par différents CSPs. Pour ce problème, on peut par exemple en proposer 3. On verra plus tard que la modélisation choisie peut avoir une influence sur l'efficacité de la résolution.

Les reines / Première modélisation

Les "inconnues" du problème sont les positions des reines sur l'échiquier. En numérotant les lignes et les colonnes de l'échiquier de la façon suivante



on peut déterminer la position d'une reine par un numéro de ligne et un numéro de colonne. Ainsi, une première modélisation consiste à associer à chaque reine *i* deux variables *Li* et *Ci* correspondant respectivement à la ligne et la colonne sur laquelle placer la reine. Les contraintes spécifient alors que les reines doivent être sur des lignes différentes, des colonnes différentes et des diagonales différentes. Notons pour cela que lorsque 2 reines sont sur une même diagonale montante, la somme de leurs numéros de ligne et de colonne est égale, tandis que lorsqu'elles sont sur une même diagonale descendante, la différence de leurs numéros de ligne et de colonne est égale. On en déduit le CSP suivant :

- Variables :

X = {L1, L2, L3, L4, C1, C2, C3, C4}

- Domaines :

D(L1) = D(L2) = D(L3) = D(L4) = D(C1) = D(C2) = D(C3) = D(C4) = {1,2,3,4}

- Contraintes : on identifie 4 types de contraintes
    - Les reines doivent être sur des lignes différentes. *Clig = {L1≠L2, L1≠L3, L1≠L4, L2≠L3, L2≠L4, L3≠L4 }*
    - Les reines doivent être sur des colonnes différentes. *Ccol = {C1≠C2, C1≠C3, C1≠C4, C2≠C3, C2≠C4, C3≠C4 }*

- Les reines doivent être sur des diagonales montantes différentes. *Cdm = {C1+L1≠C2+L2, C1+L1≠C3+L3, C1+L1≠C4+L4, C2+L2≠C3+L3, C2+L2≠C4+L4, C3+L3≠C4+L4}*
- Les reines doivent être sur des diagonales descendantes différentes. *Cdd = {C1-L1≠C2-L2, C1-L1≠C3-L3, C1-L1≠C4-L4, C2-L2≠C3-L3, C2-L2≠C4-L4, C3-L3≠C4-L4}*

L'ensemble des contraintes est défini par l'union de ces 4 ensembles :

C = Clig U Ccol U Cdm U Cdd

Les contraintes *Clig* et *Ccol* sont des contraintes binaires ; les contraintes *Cdm* et *Cdd* sont des contraintes quaternaires. L'énumération de toutes ces contraintes est ici un peu fastidieuse. On peut tout aussi bien les définir de la façon suivante :

- Contraintes :
  - les reines doivent être sur des lignes différentes *Clig = {Li ≠ Lj / i élément_de {1,2,3,4}, j élément_de {1,2,3,4} et i ≠ j}*
  - les reines doivent être sur des colonnes différentes *Ccol = {Ci ≠ Cj / i élément_de {1,2,3,4}, j élément_de {1,2,3,4} et i ≠ j}*
  - les reines doivent être sur des diagonales montantes différentes *Cdm = {Ci+Li≠Cj+Lj / i élément_de {1,2,3,4}, j élément_de {1,2,3,4} et i ≠ j}*
  - les reines doivent être sur des diagonales descendantes différentes *Cdd = {Ci-Li≠Cj-Lj / i élément_de {1,2,3,4}, j élément_de {1,2,3,4} et i ≠ j}*

On aurait également pu utiliser une contrainte globale pour exprimer le fait que toutes les variables d'un ensemble doivent avoir des valeurs différentes : *Clig = toutesDiff({L1,L2,L3,L4})* et *Ccol = toutesDiff({C1,C2,C3,C4}).*

Une solution du problème des 4 reines, pour cette première modélisation, est

A = {(C1,1), (L1,2), (C2,2), (L2,4), (C3,3), (L3,1), (C4,4), (L4,3)}

ou autrement dit, la première reine est placée colonne 1 ligne 2, la deuxième, colonne 2 ligne 4, la troisième, colonne 3 ligne 1 et la quatrième, colonne 4 ligne 3.

Les reines / Deuxième modélisation

Dans la mesure où l'on sait dès le départ qu'il y aura une reine et une seule sur chaque colonne de l'échiquier, le problème peut se résumer à déterminer sur quelle ligne se trouve la reine placée sur la colonne *i*. Par conséquent, une deuxième modélisation consiste à associer une variable*Xi* à chaque colonne *i* de telle sorte que *Xi* désigne le numéro de ligne sur laquelle placer la reine de la colonne *i*. Notons que pour cette deuxième modélisation, on a été obligé de "réfléchir" un peu pour introduire dans la modélisation une déduction (il y a une seule reine

par colonne) qui, on l'espère, va faciliter le travail de la machine. Le CSP correspondant à cette deuxième modélisation est le suivant :

- Variables :

X = {X1,X2,X3,X4}

- Domaines :

D(X1) = D(X2) = D(X3) = D(X4) = {1,2,3,4}

- Contraintes :
  - les reines doivent être sur des lignes différentes *Clig = {Xi ≠ Xj / i élément_de {1,2,3,4}, j élément_de {1,2,3,4} et i ≠ j}*
  - les reines doivent être sur des diagonales montantes différentes *Cdm = {Xi+i ≠ Xj+j / i élément_de {1,2,3,4}, j élément_de {1,2,3,4} et i ≠ j}*
  - les reines doivent être sur des diagonales descendantes différentes *Cdd = {Xi-i ≠ Xj-j / i élément_de {1,2,3,4}, j élément_de {1,2,3,4} et i ≠ j}*

L'ensemble des contraintes est défini par l'union de ces 3 ensembles

C = Clig U Cdm U Cdd

Une solution du problème des 4 reines, pour cette deuxième modélisation, est

A = {(X1,2), (X2,4), (X3,1), (X4,3)}

ou autrement dit, la reine de la colonne 1 est placée sur la ligne 2, celle de la colonne 2, ligne 4, celle de la colonne 3, ligne 1 et celle de la colonne 4, ligne 3.

Les reines / Troisième modélisation

Une autre façon, radicalement opposée, de modéliser le problème consiste à choisir comme variables non pas les positions des reines, mais les états des cases de l'échiquier : on associe une variable à chacune des 16 cases de l'échiquier (on notera *Xij* la variable associée à la case située ligne *i* et colonne *j*) ; chaque variable peut prendre pour valeur *0* (s'il n'y a pas de reine sur la case) ou *1* (s'il y a une reine sur la case) ; les contraintes spécifient qu'il ne peut y avoir plusieurs reines sur une même ligne, une même colonne ou une même diagonale. Le CSP correspondant à cette troisième modélisation est le suivant :

- Variables :

X = { X11, X12, X13, X14, X21, X22, X23, X24, X31, X32, X33, X34, X41, X42, X43, X44}

- Domaines :

D(Xij) = {0,1} pour tout i et tout j compris entre 1 et 4

- Contraintes :

- il y a une reine par ligne
  Clig = {Xi1 + Xi2 + Xi3 + Xi4 = 1 / i élément_de {1,2,3,4}}
- il y a une reine par colonne
  Ccol = {X1i + X2i + X3i + X4i = 1 / i élément_de {1,2,3,4}}
- les reines doivent être sur des diagonales montantes diférentes
  Cdm = pour tout couple de variables différentes Xij et Xkl, i+j=k+l => Xij + Kkl ≤ 1
- les reines doivent être sur des diagonales descendantes différentes
  Cdd = pour tout couple de variables différentes Xij et Xkl, i-j=k-l => Xij + Kkl ≤ 1

L'ensemble des contraintes est défini par l'union de ces 4 ensembles

C = Clig U Ccol U Cdm U Cdd

Une solution du problème des 4 reines, pour cette troisième modélisation, est

A = {(X11,0), (X12,1), (X13,0), (X14,0), (X21,0), (X22,0), (X23,0), (X24,1), (X31,1), (X32,0), (X33,0), (X34,0), (X41,0), (X42,0), (X43,1), (X44,0)}

ou, autrement dit, la case ligne 1 colonne 1 (*X11*) est vide, la case ligne 1 colonne 2 (*X12*) est occupée, ...

Les reines / Discussion

La question (légitime) que l'on peut maintenant se poser est : "Quelle est la meilleure modélisation ?"

Il n'y a pas une seule réponse à cette question... mais au moins trois :

1. Celle qui modélise le mieux la réalité du problème. De ce point de vue, les 3 modélisations sont équivalentes.
2. Celle qui est la plus facile à trouver. De ce point de vue, la première modélisation est probablement plus "simple"... même si cela est subjectif !
3. Celle qui permettra de résoudre le problème le plus efficacement. On ne peut vraiment répondre à cette question qu'à partir du moment où l'on sait comment un CSP est résolu (ce que vous ne tarderez pas à savoir ...). Intuitivement, on se doute que la deuxième modélisation devrait être meilleure que la première dans la mesure où elle prend en compte le fait que les reines sont sur des colonnes différentes par la définition même des variables, sans avoir à poser de contrainte. On verra que "l'espace de recherche" de cette deuxième modélisation est plus petit que celui de la première.

Généralisation à n reines

On peut généraliser le problème au placement de *n* reines sur un échiquier comportant *n* colonnes et *n* lignes. Par exemple, la deuxième modélisation devient :

- Variables :

X = {Xi / i est un entier compris entre 1 et n}

- Domaines :

quelquesoit Xi élément_de X, D(Xi) = {j / j est un entier compris entre 1 et n}

- Contraintes :
    - les reines doivent être sur des lignes différentes *Clig = {Xi ≠ Xj / i et j sont 2 entiers différents compris entre 1 et n}*
    - les reines doivent être sur des diagonales montantes différentes *Cdm = {Xi+i ≠ Xj+j / i et j sont 2 entiers différents compris entre 1 et n}*
    - les reines doivent être sur des diagonales descendantes différentes *Cdd = {Xi-i ≠ Xj-j / i et j sont 2 entiers différents compris entre 1 et n}*

L'ensemble des contraintes est défini par l'union de ces 3 ensembles

C = Clig U Cdm U Cdd

4. La résolution de problèmes d'optimisation sous contraintes

Les algorithmes que nous allons étudier permettent de rechercher une solution à un CSP (n'importe laquelle, c'est-à-dire la première que l'on trouve). Suivant les applications, "résoudre un CSP" peut signifier autre chose que chercher simplement une solution. En particulier, il peut s'agir de chercher la "meilleure" solution selon un critère donné.

Par exemple, pour le problème du coloriage d'une carte on peut chercher la solution qui utilise le moins de couleurs possibles ; pour le problème du retour de monnaie, on peut chercher la solution qui minimise le nombre total de pièces rendues, ...

Ces problèmes d'optimisation sous contraintes, où l'on cherche à optimiser une fonction objectif donnée tout en satisfaisant toutes les contraintes, peuvent être résolus en explorant l'ensemble des affectations possibles selon la stratégie de "Séparation & Evaluation " ("Branch & Bound") bien connue en recherche opérationnelle. On n'étudiera pas cet algorithme dans ce cours.

4.1. Les algorithmes de résolution de CSPs autres que sur les domaines finis

Les algorithmes que nous allons étudier permettent de résoudre de façon générique n'importe quel CSP sur les domaines finis. Il existe d'autres algorithmes plus spécifiques qui tirent parti de connaissances sur les domaines et les types de contraintes pour résoudre des CSPs. Par exemple, les CSPs numériques linéaires sur les réels peuvent être résolus par l'algorithme du

Simplex; les CSPs numériques linéaires sur les entiers peuvent être résolus en combinant l'algorithme du Simplex avec une stratégie de "Séparation et Evaluation" ; les CSPs numériques non linéaires sur les réels peuvent être résolus en utilisant des techniques de propagation d'intervalles ; etc...

4.2. Les algorithmes pour la résolution de CSPs

Les algorithmes que nous allons étudier sont dits complets, dans le sens où l'on est certain de trouver une solution si le CSP est consistant. Cette propriété de complétude est fort intéressante dans la mesure où elle offre des garanties sur la qualité du résultat. En revanche, elle impose de parcourir exhaustivement l'ensemble des combinaisons possibles, et même si l'on utilise différentes techniques et heuristiques pour réduire la combinatoire, il existe certains problèmes pour lesquels ce genre d'algorithme ne termine pas "en un temps raisonnable". Par opposition, les algorithmes incomplets ne cherchent pas à envisager toutes les combinaisons, mais cherchent à trouver le plus vite possible une affectation "acceptable" : ces algorithmes permettent de trouver rapidement de bonnes affectations (qui violent peu ou pas de contraintes) ; en revanche, on n'est pas certain que la meilleure affectation trouvée par ces algorithmes soit effectivement optimale ; on est par ailleurs certain que l'on ne pourra pas prouver l'optimalité de l'affectation trouvée... Il existe différents algorithmes incomplets pour résoudre de façon générique des CSPs sur les domaines finis, généralement basés sur des techniques de recherche locale (éventuellement combinées avec des méthodes tabou, du recuit simulé, des algorithmes à base de fourmis...).

4.3. L'algorithme "génère et teste"

4.3.1. Principe de l'algorithme "génère et teste"

La façon la plus simple (et très naïve !) de résoudre un CSP sur les domaines finis consiste à énumérer toutes les affectations totales possibles jusqu'à en trouver une qui satisfasse toutes les contraintes. Ce principe est repris dans la fonction récursive "*genereEtTeste(A,(X,D,C))*" décrite ci-dessous. Dans cette fonction, *A* contient une affectation partielle et *(X,D,C)* décrit le CSP à résoudre (au premier appel de cette fonction, l'affectation partielle *A* sera vide). La fonction retourne *vrai* si on peut étendre l'affectation partielle *A* en une affectation totale consistante (une solution), et faux sinon.

fonction genereEtTeste(A,(X,D,C)) retourne un booléen
Précondition :

(X,D,C) = un CSP sur les domaines finis
A = une affectation partielle pour (X,D,C)

Postrelation :

retourne vrai si l'affectation partielle A peut être étendue en une solution pour (X,D,C), faux sinon

début

si toutes les variables de X sont affectées à une valeur dans A alors

/ A est une affectation totale */*
si A est consistante alors

/ A est une solution */*
retourner vrai

sinon

retourner faux

finsi

sinon */* A est une affectation partielle */*

choisir une variable Xi de X qui n'est pas encore affectée à une valeur dans A
pour toute valeur Vi appartenant à D(Xi) faire

si genereEtTeste(A U {(Xi,Vi)}, (X,D,C)) = vrai alors retourner vrai

finpour
retourner faux

finsi

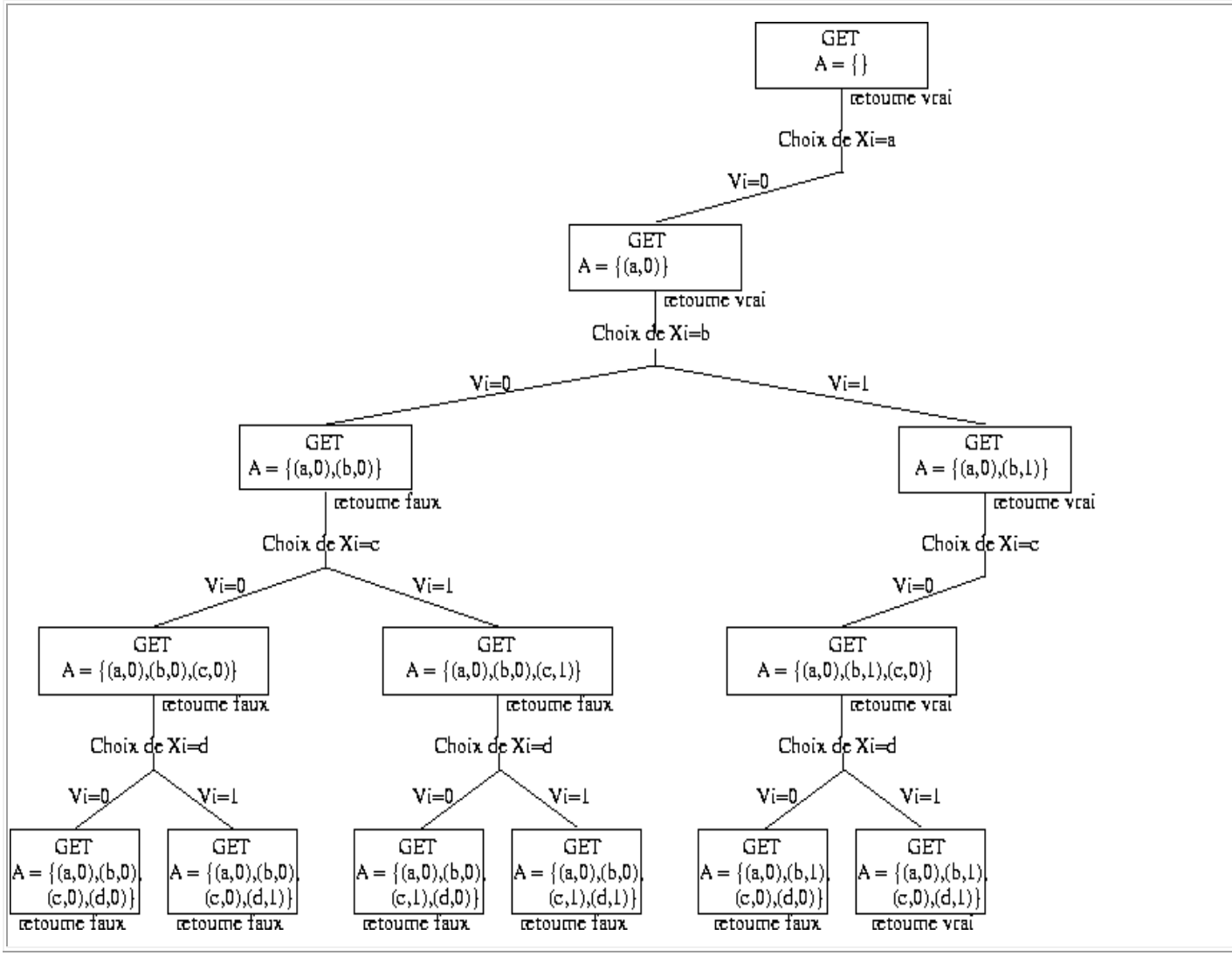
fin

4.3.2. Exemple de trace d'exécution de "génère et teste"

Considérons par exemple le CSP *(X,D,C)* suivant :

- $X = \{a,b,c,d\}$
- $D(a) = D(b) = D(c) = D(d) = \{0,1\}$
- $C = \{ a \neq b, c \neq d, a+c < b \}$

L'enchainement des appels successifs à la fonction genereEtTeste (abrégée par GET) est représenté ci-dessous (chaque rectangle correspond à un appel de la fonction, et précise la valeur de l'affectation partielle en cours de construction *A*).



4.3.3. Critique de "génère et teste" et notion d'espace de recherche d'un CSP

L'algorithme "génère et teste" que nous venons de voir énumère l'ensemble des affectations complètes possibles, jusqu'à en trouver une qui soit consistante. L'ensemble des affectations complètes est appelé l'*espace de recherche du CSP*. Si le domaine de certaines variables contient une infinité de valeurs, alors cet espace de recherche est infini et on ne pourra pas énumérer ses éléments en un temps fini. Néanmoins, même en se limitant à des domaines comportant un nombre fini de valeurs, l'espace de recherche est souvent de taille tellement importante que l'algorithme "génère et teste" ne pourra se terminer en un temps "raisonnable". En effet, l'espace de recherche d'un CSP *(X,D,C)* comportant *n* variables (*X = {X1, X2, ..., Xn}*) est défini par

E = { {(X1,v1), (X2,v2), ..., (Xn,vn)} / quelquesoit i compris entre 1 et n, vi est un élément de D(Xi) }

et le nombre d'éléments de cet espace de recherche est défini par

$| E | = | D(X1) | * | D(X2) | * ... * | D(Xn) |$

de sorte que, si tous les domaines des variables sont de la même taille k (autrement dit, $| D(Xi) |=k$), alors la taille de l'espace de recherche est

$| E | = k^n$

Ainsi, le nombre d'affectations à générer croit de façon exponentielle en fonction du nombre de variables du problème. Considérons plus précisément un CSP ayant n variables, chaque variable pouvant prendre 2 valeurs ($k=2$). Dans ce cas, le nombre d'affectations totales possibles pour ce CSP est 2^n. En supposant qu'un ordinateur puisse générer et tester un milliard d'affectations par seconde (ce qui est une estimation vraiment très optimiste !), le tableau suivant donne une idée du temps qu'il faudrait pour énumérer et tester toutes les affectations en fonction du nombre de variables n.

Nombre de variables n	**Nombre d'affectations totales 2^n**	**Temps (si on traitait 10^9 affectations par seconde)**
10	environ 10^3	environ 1 millionième de seconde
20	environ 10^6	environ 1 millième de seconde
30	environ 10^9	environ 1 seconde
40	environ 10^12	environ 16 minutes
50	environ 10^15	environ 11 jours
60	environ 10^18	environ 32 ans
70	environ 10^21	environ 317 siècles

- **Les améliorations de "génère et teste" que nous allons étudier dans la suite...**

La conclusion de ce petit exercice de dénombrement est que, dès lors que le CSP a plus d'une trentaine de variables, on ne peut pas appliquer "bêtement" l'algorithme "génère et teste". Il faut donc chercher à réduire l'espace de recherche. Pour cela, on peut notamment :

- ne développer que les affectations partielles consistantes: dès lors qu'une affectation partielle est inconsistante, il est inutile de chercher à l'étendre en une affectation totale puisque celle-ci sera nécessairement inconsistante ;

- réduire les tailles des domaines des variables en leur enlevant les valeurs "incompatibles" : pendant la génération d'affectations, on filtre le domaine des variables pour ne garder que les valeurs "localement consistantes" avec l'affectation en cours de construction, et dès lors que le domaine d'une variable devient vide, on arrête l'énumération pour cette affectation partielle ;
- introduire des "heuristiques" pour "guider" la recherche : lorsqu'on énumère les affectations possibles, on peut essayer d'énumérer en premier celles qui sont les plus "prometteuses", en espérant ainsi tomber rapidement sur une solution.

Il existe encore bien d'autres façons de (tenter de) réduire la combinatoire, afin de rendre l'exploration exhaustive de l'espace de recherche possible, que nous ne verrons pas dans ce cours. Par exemple, lors d'un échec, on peut essayer d'identifier la cause de l'échec (quelle est la variable qui viole une contrainte) pour ensuite "retourner en arrière" directement là où cette variable a été instanciée afin de remettre en cause plus rapidement la variable à l'origine de l'échec. C'est ce que l'on appelle le "retour arrière intelligent" ("intelligent backtracking").

Une autre approche particulièrement séduisante consiste à exploiter des connaissances sur les types de contraintes utilisées pour réduire l'espace de recherche.

considérons par exemple le CSP (X,D,C) suivant :

- X={a,b,c},
- D(a)=D(b)=D(c)={0,1,2,3,4, ..., 1000},
*- C={4*a - 2*b = 6*c + 3}*

*L'espace de recherche de ce CSP comporte 1 milliard d'affectations ; pour résoudre ce CSP, on peut énumérer toutes ces combinaisons, en espérant en trouver une qui satisfasse la contrainte 4*a - 2*b = 6*c + 3... c'est long et si on remplace la borne supérieure 1000 par l'infini, ça devient carrément impossible. En revanche un simple raisonnement permet de conclure très rapidement que ce CSP n'a pas de solution : la partie gauche de l'équation "4*a - 2*b" donnera toujours un nombre pair, quelles que soient les valeurs affectées à a et b, tandis que la partie droite "6*c + 3" donnera toujours un nombre impair, quelle que soit la valeur affectée à c ; par conséquent, on ne peut trouver d'affectation qui satisfasse la contrainte, et il est inutile d'énumérer toutes les affectations possibles pour s'en convaincre !*

Ce genre de raisonnement demande de l'intelligence, ou pour le moins des connaissances. De fait, l'homme est capable de résoudre des problèmes très combinatoires en raisonnant (en utilisant son "expérience" et des connaissances plus ou moins explicitées). Un exemple typique est le jeu d'échec : les grands joueurs d'échecs n'envisagent pour chaque coup à jouer que très peu de combinaisons (les meilleures évidemment !), éliminant par des raisonnements souvent difficiles à expliciter un très grand nombre de combinaisons moins intéressantes. Cela

explique le fait que, malgré leur capacité à envisager un très grand nombre de combinaisons, les ordinateurs sont encore (bien souvent) moins forts que ces grands joueurs.

4.4. L'algorithme "simple retour-arrière"

4.4.1. Principe de l'algorithme "simple retour-arrière"

Une première façon d'améliorer l'algorithme "génère et teste" consiste à tester au fur et à mesure de la construction de l'affectation partielle sa consistance : dès lors qu'une affectation partielle est inconsistante, il est inutile de chercher à la compléter. Dans ce cas, on "retourne en arrière" ("backtrack" en anglais) jusqu'à la plus récente instanciation partielle consistante que l'on peut étendre en affectant une autre valeur à la dernière variable affectée.

Par exemple, sur la trace d'exécution décrite en figure précédente, *on remarque que l'algorithme génère tous les prolongements de l'affectation partielle A={(a,0),(b,0)}, en énumérant toutes les possibilités d'affectation pour les variables c et d, alors qu'elle viole la contrainte a ≠ b. L'algorithme "simple retour-arrière" ne va donc pas chercher à étendre cette affectation, mais va "retourner en arrière" à l'affectation partielle précédente A={(a,0)}, et va l'étendre en affectant 1 à b , ...*

Ce principe est repris dans la fonction récursive "simpleRetourArrière(A,(X,D,C))" décrite ci-dessous. Dans cette fonction, *A* contient une affectation partielle et *(X,D,C)* décrit le CSP à résoudre (au premier appel de cette fonction, l'affectation partielle *A* sera vide). La fonction retourne *vrai* si on peut étendre l'affectation partielle *A* en une affectation totale consistante (une solution), et faux sinon.

```
fonction simpleRetourArrière(A,(X,D,C)) retourne un booléen
Précondition :

A = affectation partielle
(X,D,C) = un CSP sur les domaines finis

Postrelation :

retourne vrai si A peut être étendue en une solution pour (X,D,C), faux sinon

début

si A n'est pas consistante alors retourner faux finsi
si toutes les variables de X sont affectées à une valeur dans A alors

/* A est une affectation totale et consistante = une solution */
retourner vrai

sinon /* A est une affectation partielle consistante */
```

choisir une variable Xi de X qui n'est pas encore affectée à une valeur dans A
pour toute valeur Vi appartenant à D(Xi) faire

si simpleRetourArrière(A U {(Xi,Vi)}, (X,D,C)) = vrai alors retourner vrai

finpour
retourner faux

finsi

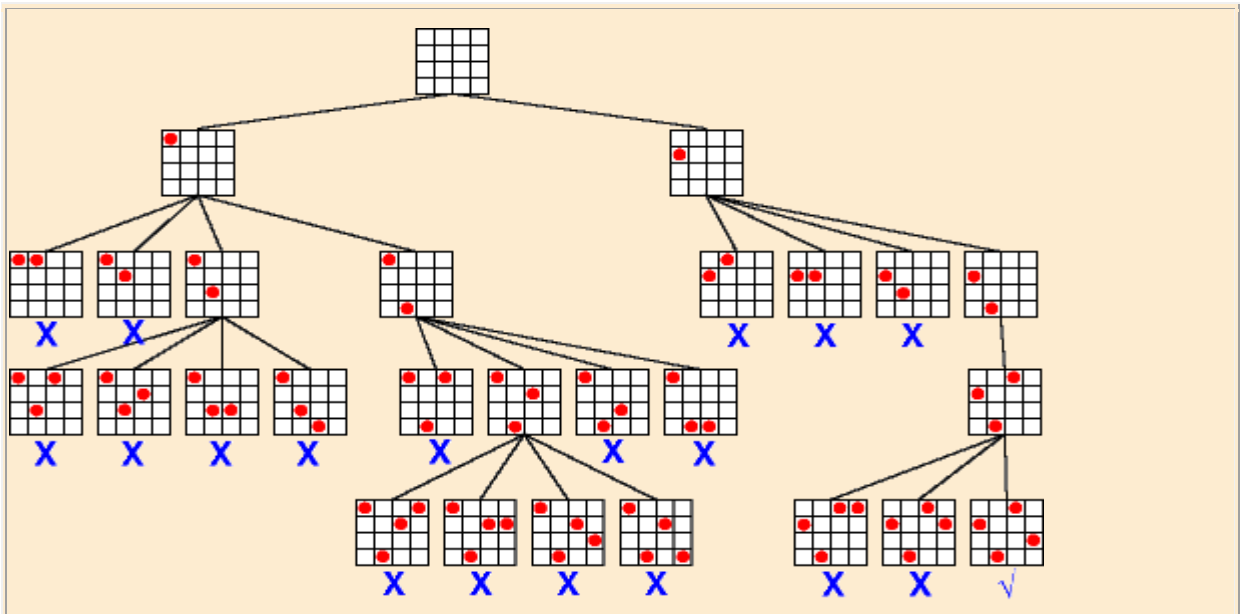
fin

3.2 - Exemple de "trace d'exécution" de SimpleRetourArrière

Considérons le problème du placement de 4 reines sur un échiquier 4x4, et sa deuxième modélisation proposée au début de ce chapitre

- Variables : $X = \{X1,X2,X3,X4\}$
- Domaines : $D(X1) = D(X2) = D(X3) = D(X4) = \{1,2,3,4\}$
- Contraintes : $C = \{Xi \neq Xj \ / \ i \text{ élément\_de } \{1,2,3,4\}, j \text{ élément\_de } \{1,2,3,4\} \text{ et } i \neq j\}$
  $U \{Xi+i \neq Xj+j \ / \ i \text{ élément\_de } \{1,2,3,4\}, j \text{ élément\_de } \{1,2,3,4\} \text{ et } i \neq j\}$
  $U \{Xi-i \neq Xj-j \ / \ i \text{ élément\_de } \{1,2,3,4\}, j \text{ élément\_de } \{1,2,3,4\} \text{ et } i \neq j\}$

L'enchainement des appels successifs à la fonction SimpleRetourArrière peut être représenté par l'arbre suivant (chaque noeud correspond à un appel de la fonction ; l'échiquier dessiné à chaque noeud décrit l'affectation partielle en cours) :



Cette image est empruntée au "Guide to Constraint Programming" de Roman Bartak.

4.5. L'algorithme "anticipation"

4.6. Notions de filtrage et de consistance locale

Pour améliorer l'algorithme "simple retour-arrière", on peut tenter d'anticiper ("look ahead" en anglais) les conséquences de l'affectation partielle en cours de construction sur les domaines des variables qui ne sont pas encore affectées : si on se rend compte qu'une variable non affectée *Xi* n'a plus de valeur dans son domaine *D(Xi)* qui soit "localement consistante" avec l'affectation partielle en cours de construction, alors il n'est pas nécessaire de continuer à développer cette branche, et on peut tout de suite retourner en arrière pour explorer d'autres possibilités.

Pour mettre ce principe en œuvre, on va, à chaque étape de la recherche, filtrer les domaines des variables non affectées en enlevant les valeurs "localement inconsistantes", c'est-à-dire celles dont on peut inférer qu'elles n'appartiendront à aucune solution. On peut effectuer différents filtrages, correspondant à différents niveaux de consistances locales, qui vont réduire plus ou moins les domaines des variables, mais qui prendront aussi plus ou moins de temps à s'exécuter : considérons un CSP *(X,D,C),* et une affectation partielle consistante *A,*

- le filtrage le plus simple consiste à anticiper d'une étape l'énumération : pour chaque variable *Xi* non affectée dans *A,* on enlève de *D(Xi)*toute valeur *v* telle que l'affectation *A U {(Xi,v)}* soit inconsistante.

Par exemple pour le problème des 4 reines, après avoir instancié X1 à 1, on peut enlever du domaine de X2 la valeur 1 (qui viole la contrainte X1 ≠ X2) et la valeur 2 (qui viole la contrainte 1-X1 ≠ 2-X2).

Un tel filtrage permet d'établir ce qu'on appelle la ***consistance de noeud***, aussi appelée ***1-consistance***.

Définition de la consistance de noeud : *Un CSP (X,D,C) est consistant de noeud si pour toute variable Xi de X, et pour toute valeur v de D(Xi), l'affectation partielle {(Xi,v)} satisfait toutes les contraintes unaires de C.*

Par exemple, si C contient la contrainte "X1 > 2", et si le domaine de X1 contient les valeurs {1,2,3,4,5}, alors le CSP n'est pas consistant de noeud. Pour qu'il soit consistant de noeud, il faut enlever du domaine de X1 les valeurs 1 et 2 qui violent la contrainte "X1 > 2".

- un filtrage plus fort, mais aussi plus long à effectuer, consiste à anticiper de deux étapes l'énumération : pour chaque variable *Xi* non affectée dans *A,* on enlève de *D(Xi)* toute valeur *v* telle qu'il existe une variable *Xj* non affectée pour laquelle, pour toute valeur *w* de*D(Xj),* l'affectation *A U {(Xi,v),(Xj,w)}* soit inconsistante.

Par exemple pour le problème des 4 reines, après avoir instancié X1 à 1, on peut enlever la valeur 3 du domaine de X2 car si X1=1 et X2=3, alors la variable X3 ne peut plus prendre de valeurs : si X3=1, on viole la contrainte X3 ≠ X1 ; si X3=2, on viole la contrainte X3+3 ≠ X2+2 ; si X3=3, on viole la contrainte X3 ≠ X2 ; et si X3=4, on viole la contrainte X3-3 ≠ X2-2.

Notons que ce filtrage doit être répété jusqu'à ce plus aucun domaine ne puisse être réduit. Ce filtrage permet d'établir ce qu'on appelle la ***consistance d'arc***, aussi appelée ***2-consistance***. ***Définition de la consistance d'arc :*** *Un CSP (X,D,C) est consistant d'arc si pour tout couple de variables (Xi,Xj) de X, et pour toute valeur vi appartenant à D(Xi), il existe une valeur vj appartenant à D(Xj) telle que l'affectation partielle {(Xi,vi),(Xj,vj)} satisfasse toutes les contraintes binaires de C.*

un filtrage encore plus fort, mais aussi encore plus long à effectuer, consiste à anticiper de trois étapes l'énumération. Ce filtrage permet d'établir ce qu'on appelle la ***consistance de chemin***, aussi appelée ***3-consistance***.

- ... et ainsi de suite... notons que s'il reste *k* variables à affecter, et si l'on anticipe de *k* étapes l'énumération pour établir la k-consistance, l'opération de filtrage revient à résoudre le CSP, c'est-à-dire que toutes les valeurs restant dans les domaines des variables après un tel filtrage appartiennent à une solution.

4.7. Principe de l'algorithme "anticipation"

Le principe général de l'algorithme "anticipation" reprend celui de l'algorithme "simple retour-arrière", en ajoutant simplement une étape de filtrage à chaque fois qu'une valeur est affectée à une variable. Comme on vient de le voir au point 4.1, on peut effectuer différents filtrages plus ou moins forts, permettant d'établir différents niveaux de consistance locale (noeud, arc, chemin, ...). Par exemple, la fonction récursive "anticipation/noeud(A,(X,D,C))" décrite ci-dessous effectue un filtrage simple qui établit à chaque étape la consistance de noeud. Dans cette fonction, *A* contient une affectation partielle consistante et *(X,D,C)* décrit le CSP à résoudre (au premier appel de cette fonction, l'affectation partielle *A* sera vide). La fonction retourne *vrai* si on peut étendre l'affectation partielle *A* en une affectation totale consistante (une solution), et faux sinon.

fonction anticipation/noeud(A,(X,D,C)) retourne un booléen
Précondition :

A = affectation partielle consistante
(X,D,C) = un CSP sur les domaines finis

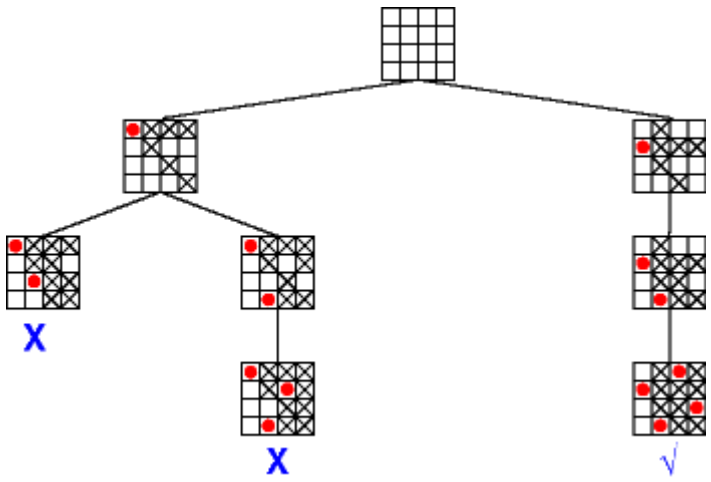
Postrelation :

```
retourne vrai si A peut être étendue en une solution pour (X,D,C), faux sinon
début
si toutes les variables de X sont affectées à une valeur dans A alors
/* A est une affectation totale et consistante = une solution */
retourner vrai
sinon /* A est une affectation partielle consistante */
choisir une variable Xi de X qui n'est pas encore affectée à une valeur dans A
pour toute valeur Vi appartenant à D(Xi) faire
/* filtrage des domaines par rapport à A U {(Xi,Vi)} */
pour toute variable Xj de X qui n'est pas encore affectée faire
Dfiltré(Xj) <- { Vj élément de D(Xj) / A U {(Xi,Vi),(Xj,Vj)} est consistante }
si Dfiltré(Xj) est vide alors retourner faux
finpour
si anticipation(A U {(Xi,Vi)}, (X,Dfiltré,C))=vrai alors retourner vrai
finpour
retourner faux
finsi
fin
```

Nous n'étudierons pas dans le cadre de ce cours l'algorithme de filtrage qui établit la consistance d'arc.

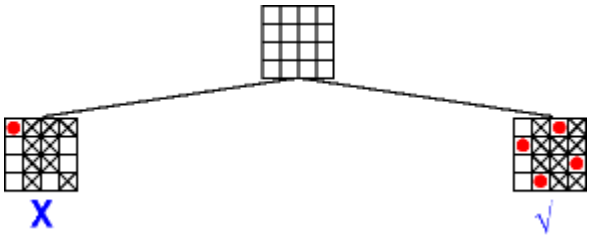
4.3 - Exemple de "trace d'exécution" de "anticipation"

Considérons de nouveau le problème du placement de 4 reines sur un échiquier 4x4. L'enchainement des appels successifs à la fonction "Anticipation/nœud" peut être représenté par l'arbre suivant (les valeurs supprimées par le filtrage sont marquées d'une croix) :



Cette image est empruntée au "Guide to Constraint Programming" de Roman Bartak.

Si on appliquait un filtrage plus fort, qui rétablirait à chaque étape la consistance d'arc, l'enchainement des appels successifs à la fonction "Anticipation/arc" correspondante serait le suivant (les valeurs supprimées par le filtrage sont marquées d'une croix) :



Cette image est empruntée au "Guide to Constraint Programming" de Roman Bartak.

Ainsi, on constate sur le problème des 4 reines que le filtrage des domaines permet de réduire le nombre d'appels récursifs : on passe de 27 appels pour "simple retour-arrière" à 8 appels pour l'algorithme d'anticipation avec filtrage simple établissant une consistance de noeud. En utilisant des filtrages plus forts, comme celui qui établit la consistance d'arc, on peut encore réduire la combinatoire de 8 à 3 appels récursifs. Cependant, il faut noter que plus le filtrage utilisé est fort, plus cela prendra de temps pour l'exécuter...

De façon générale, on constate que, quel que soit le problème considéré, l'algorithme "anticipation/noeud" est généralement plus rapide que l'algorithme "simple retour-arrière" car le filtrage utilisé est vraiment peu coûteux. En revanche, si l'algorithme "anticipation/arc" envisage toujours moins de combinaisons que l'algorithme "anticipation/noeud", il peut parfois prendre plus de temps à l'exécution car le filtrage pour établir une consistance d'arc est plus long que celui pour établir la consistance de noeud.

4.8. Intégration d'heuristiques

Les algorithmes que nous venons d'étudier choisissent, à chaque étape, la prochaine variable à instancier parmi l'ensemble des variables qui ne sont pas encore instanciées ; ensuite, une fois la variable choisie, ils essayent de l'instancier avec les différentes valeurs de son domaine. Ces algorithmes ne disent rien sur l'ordre dans lequel on doit instancier les variables, ni sur l'ordre dans lequel on doit affecter les valeurs aux variables. Ces deux ordres peuvent changer considérablement l'efficacité de ces algorithmes : imaginons qu'à chaque étape on dispose des conseils d'un "oracle-qui-sait-tout" qui nous dise quelle valeur choisir sans jamais se tromper ; dans ce cas, la solution serait trouvée sans jamais retourner en arrière... Malheureusement, le problème général de la satisfaction d'un CSP sur les domaines finis étant NP-complet, il est plus qu'improbable que cet oracle fiable à 100% puisse jamais être "programmé". En revanche, on peut intégrer des heuristiques pour déterminer l'ordre dans lequel les variables et les valeurs doivent être considérées : une heuristique est une règle non systématique (dans le sens où elle n'est pas fiable à 100%) qui nous donne des indications sur la direction à prendre dans l'arbre de recherche.

Les heuristiques concernant l'ordre d'instanciation des valeurs sont généralement dépendantes de l'application considérée et difficilement généralisables. En revanche, il existe de nombreuses heuristiques d'ordre d'instanciation des variables qui permettent bien souvent d'accélérer considérablement la recherche. L'idée générale consiste à instancier en premier les variables les plus "critiques", c'est-à-dire celles qui interviennent dans beaucoup de contraintes et/ou qui ne peuvent prendre que très peu de valeurs. L'ordre d'instanciation des variables peut être :

- ***statique***, quand il est fixé avant de commencer la recherche ;

Par exemple, on peut ordonner les variables en fonction du nombre de contraintes portant sur elles : l'idée est d'instancier en premier les variables les plus contraintes, c'est-à-dire celles qui participent au plus grand nombre de contraintes.

- ou ***dynamique***, quand la prochaine variable à instancier est choisie dynamiquement à chaque étape de la recherche.

Par exemple, l'heuristique "échec d'abord" ("first-fail" en anglais) consiste à choisir, à chaque étape, la variable dont le domaine a le plus petit nombre de valeurs localement consistantes avec l'affectation partielle en cours. Cette heuristique est généralement couplée avec l'algorithme "anticipation", qui filtre les domaines des variables à chaque étape de la recherche pour ne garder que les valeurs qui satisfont un certain niveau de consistance locale.

References

http://liris.cnrs.fr/christine.solnon/Site-PPC/e-miage-ppc-som.htm