جامعة محمد بوضياف - المسيلة
Université Mohamed Boudiaf - M'sila

République Algérienne Démocratique et Populaire
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique
Université Mohamed Boudiaf de M'sila
Faculté de Mathématiques et d'Informatique
Département d'Informatique

Support de cours d'optimisation combinatoire

Focus sur les méthodes de résolution approchée

Par :

HEMMAK Allaoua

- Juin 2017 -

Table des matières

CHAPITRE IV
HYBRIDATION DE METAHEURISTIQUES

Préface

Dans toutes les références académiques et professionnelles, l'action d'optimiser une grandeur dans un espace d'instances candidates consiste à explorer ce dernier afin d'en dégager la meilleure dite optimale réalisant un cout minimal ou un profil maximal ou encore un bon compromis de ces deux critères. Ce processus s'impose fréquemment aux entreprises suite aux conjectures de recherche opérationnelle auxquelles elles font face dans leur survie caractérisée essentiellement par une concurrence acharnée. Cette action est toujours couronnée par une prise de décision qui affecte profondément le devenir de l'entreprise en question. En effet, le trio recherche opérationnelle, optimisation et prise de décisions est devenu l'élément clé dans la gestion de l'entreprise qui conditionne son allure. En effet, des techniques de modélisation, des outils de formulation et des méthodes de résolution ont été développés au fil des années afin de mener à terme les problèmes confrontés. Cependant, en dépit de l'évolution spectaculaire de l'outil informatique et les technologies de l'information et de la communication en particulier, la difficulté de l'exploration de l'espace des instances réside dans sa structure de plus en plus complexe et sa taille souvent importante et de l'ordre de l'exponentielle, ce qui a imposé la mise au point de nouvelles méthodes dites approchées qui fournissent des solutions pas nécessairement optimales mais prouvées empiriquement proches de l'optimal et dans un temps admissible. C'est ainsi que les heuristiques et métaheuristiques, à exploration partielle stochastique mais guidée de cet espace, ont pu voir le jour et se sont imposées dans le domaine de l'optimisation au détriment des méthodes exactes, à exploration exhaustive, qui se contentent de jauger les méthodes approchées avec les instances de petites tailles.

C'est en partant de ce principe, que nous nous proposons de mettre à la disposition de nos étudiants, enseignants, chercheurs et ingénieurs le présent document qui exhibe les notions fondamentales de l'optimisation combinatoire dotées d'exemples et exercices adéquats. Nous y mettrons l'accent sur la taxonomie des différentes méthodes de résolution tout en focalisant sur les métaheuristiques, leur flexibilité et leur efficacité.

Dans ce contexte, et afin de mieux aborder et gérer cette tâche, nous nous sommes définis les objectifs suivants :

- Fournir un support théorique de cours d'optimisation combinatoire qui servira d'appui pour nos étudiants pour bien assimiler les notions de ce domaine.
- Mette l'optique sur l'importance des métaheuristiques, leur emploi, leur paramétrage et leur apport en matière d'optimisation combinatoire.
- Justifier l'ere de l'hybridation de metaheuristiques qui vise à atténuer leurs handicaps et profiter de leurs forces.

Enfin, pour mieux exposer le sujet abordé et mettre en relief les éléments clés revus ainsi que les points développés, nous nous sommes optés pour une organisation de ce document en cinq chapitres :

- Dans le premier chapitre, nous présenterons la terminologie de l'optimisation combinatoire, les définitions de notions fondamentales en mettant l'accent sur la taxonomie des différentes méthodes de résolution.
- Dans le deuxième chapitre, nous étalerons quelques méthodes exactes, leurs principes, leurs complexités en les dotant par des exemples explicatifs.
- Le troisième chapitre sera dédié aux methodes approchées. Nous y evoquerons les heuristiques, les métaheuristiques à solution unique, en l'occurrence la famille de métaheuristiques de recherche locale ou à trajectoire et enfin les métaheuristiques à population de solutions en focalisant sur les algorithmes évolutionnaires et ceux dits de la vie artificielle, leurs modalités d'applications et des exemples pratiques.
- Dans le dernier chapitre, nous présenterons les algorithmes hybrides, leurs aspects, leur apport et leurs différentes classifications.

CHAPITRE PREMIER

OPTIMISATION COMBINATOIRE
Terminologie et définitions

1. Introduction

Avant d'entamer les méthodes de résolution, il nous est avéré indispensable d'aborder dans le présent chapitre les concepts fondamentaux ayant rapport avec notre thème tels que les problèmes d'optimisation et les différentes méthodes de résolution pour mettre en évidence le cadre général où se situe notre thème et à quelles notions se réfère-t-il. Ainsi, afin de mieux cerner le contexte et les objectifs de notre thème, nous y évoquerons respectivement les notions d'optimisation et en particulier l'optimisation combinatoire puis les différentes méthodes de résolution: les méthodes exactes, les méthodes approchées, les métaheuristiques et les méthodes hybrides, en se focalisant sur leur classification pour qu'on puisse en construire une idée plus claire et systématique en sachant où situer chacune d'elles et quelles en sont les spécificités.

2. Optimisation combinatoire

D'abord laissons à part le qualificatif "combinatoire" et focalisons nous, pour l'instant, sur le terme "optimisation".

Quand une entreprise désire par exemple minimiser le cout de stock de son produit tout en tenant compte des délais de livraison à ses clients et des capacités de stock de ses hangars, elle doit rechercher le meilleur compromis répondant à cette conjecture.

Il s'agit, en fait, de trouver une meilleure politique de production dite "solution optimale" (quantités à produire pour chaque période et pour chaque hangar) minimisant une grandeur dite "fonction objectif" (le cout de stock) et satisfaisant des clauses dites "contraintes" (délais de livraison et capacités de stock des hangars).

2.1. Définitions

En termes mathématiques, un problème d'optimisation peut être formulé par le système suivant (Figure 1.1.) :

$$\begin{cases} \text{Opt}_{x \in \Omega} \ f(x) \\ \text{Sujet à } C_i(x) \ / \ i = 1,2,\ldots \end{cases}$$

Où :

- Ω est l'espace de recherche à explorer pour trouver les solutions. C'est l'ensemble des instances de la variable x. On peut avoir $\Omega = R^n$, ainsi x serait un vecteur de réels (Figure 1.2.), Ω est dit espace d'états ou encore espaces de configurations.
- f: $\Omega \rightarrow R$ est une fonction (généralement réelle), dite critère ou fonction objectif; c'est la grandeur à optimiser, c'est-à-dire rechercher son optimum. En général, il s'agit d'un cout à

minimiser ou d'un gain à maximiser, ainsi l'opérateur opt désigne max ou min selon les cas. L'argument de cet opérateur étant *f*.

- $C_i(x)$ est un système d'assertions booléennes représentant les clauses que devrait satisfaire la variable x. Ce sont les contraintes du problème.

Optimiser une fonction c'est donc trouver l'instance $x^* \in \Omega$ de sa variable x réalisant la meilleure valeur pour *f* et satisfaisant les contraintes $C_i(x)$: $x^* = \text{ArgOpt } f(x)$.

Dans la plupart des cas, meilleure valeur de *f* désigne son extremum, minimum ou maximum. Ainsi, optimiser une fonction veut dire trouver son minimum ou son maximum satisfaisant les contraintes imposées par la spécification du problème. Si la variable x est continue (i.e. Ω est isomorphe à R ou à une partie de R), on parle d'optimisation continue. En revanche, si la variable x est discrète (i.e. Ω est isomorphe à N ou à une partie de N), on parle d'optimisation combinatoire ou discrète. Néanmoins, en général, le qualificatif combinatoire est utilisé quand le nombre de combinaisons à explorer (|Ω|) est fini mais explose (potentiellement très grand), c'est-à-dire sa grandeur est de d'ordre exponentiel.

Dans ce chapitre, nous nous intéresserons qu'à ce type de problèmes, et plus particulièrement, nous mettons l'accent sur les méthodes de leur résolution.

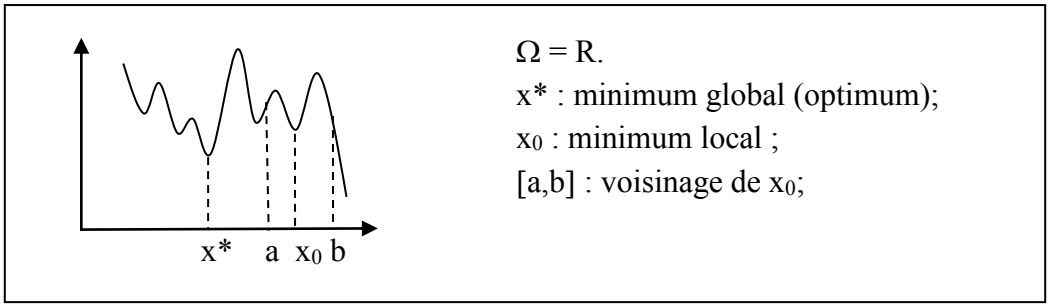


Figure 1.1. Représentation graphique d'un critère à optimiser.

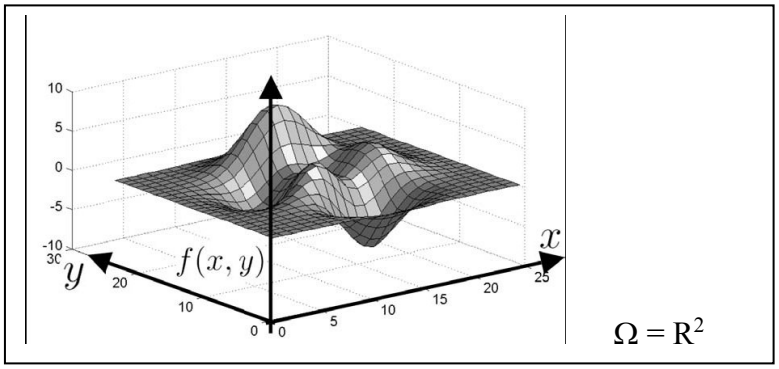
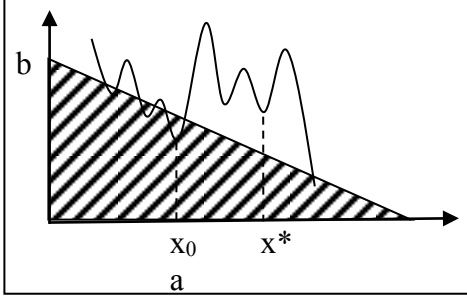


Figure 1.2. Représentation graphique d'un objectif en 2D.

2.2. Terminologie

Dans le but de bien éclaircir le domaine de notre thème, nous en présentons ici quelques termes que nous jugeons utiles pour la compréhension du travail qui sera abordé plus loin :
Solution réalisable : une instance de $x \in \Omega$ satisfaisant les contraintes du problème.
Région réalisable : ensemble S des solutions réalisables ; $S \subseteq \Omega$. (Figure 1.3.)



x_0 : solution non réalisable ;
Contrainte : $f(x) \geq -bx/a + b$;
x* : minimum global ;
Région réalisable : abscisses des points de la zone non hachurée.

Figure 1.3. Représentation graphique de la région réalisable.

Solution optimale : c'est la solution réalisant la meilleure valeur de la fonction objectif parmi celles explorées.

Méthode exacte : une méthode à exploration totale (exhaustive mais souvent intelligente) de l'espace de solutions Ω qui fournit ainsi une solution exacte prouvée formellement optimale.

Méthode approchée : une méthode à exploration partielle de Ω (stochastique mais souvent guidée par l'inspiration d'un phénomène naturel) et qui fournit alors une solution approchée qui devrait être prouvée empiriquement.

Méthode déterministe : à chaque exécution, on effectue les mêmes opérations et fournit les mêmes résultats pour les mêmes données.

Méthode stochastique : une méthode dont certaines (ou toutes) opérations sont soumises au hasard (guidées par des tirages aléatoires) et ne fournit pas forcément les mêmes résultats pour les mêmes données.

Problème multi-objectifs : la fonction objectif n'associe pas une valeur numérique, mais un point d'un ensemble de valeurs qui sera le plus souvent associé à un vecteur. L'objectif est alors d'optimiser simultanément l'ensemble des composantes de ce vecteur. On peut aussi voir l'optimisation multi-objectifs comme un ensemble de problèmes d'optimisation portant sur les

mêmes paramètres, ayant des objectifs éventuellement contradictoires, et que l'on cherche à résoudre au mieux.
Voisinage de x_0 : une partie V_0 de S contenant x_0. Le voisinage d'une solution est l'ensemble des solutions voisines, c'est l'ensemble des solutions accessibles par un mouvement (et un seul) c.-à-d. un changement élémentaire de la solution.

Recherche gloutonne : une recherche où l'on construit une solution incrémentalement en rajoutant à chaque pas un élément selon un critère glouton, i.e. celui qui nous paraît "localement" le meilleur. On fait donc des choix à "court terme" sans jamais les remettre en cause.

Recherche Perturbatrice
- Espace de recherche = ensemble complet de solutions candidates.
- Une étape de recherche = modification d'une ou plusieurs composante d'une solution.

Recherche systématique
- 'Traverser/parcourir' l'espace de recherche de façon systématique.
- Evalue toute solution possible jusqu'à la solution optimale globale trouvée.
- Complète : garantie de trouver une solution (optimale), où de déterminer qu'aucune solution n'existe.

Recherche constructive
- Espace de recherche = solutions candidates partielles ;
- Etape de recherche = extension avec un ou plusieurs nouveaux composants.
- Utilisée pour construire une bonne solution qui est ensuite améliorée en utilisant d'autres paradigmes de recherche.

Recherche locale
- Commencer avec une solution initiale de l'espace de recherche ;
- Itérativement visiter une solution 'voisine' ;
- Incomplète : pas de garantie de trouver une solution (optimale) et/ou ne peut pas garantir qu'il n'y a pas de solutions avec certitude.

Minimum global : une solution réalisable x* telle que : $\forall x \in S : f(x^*) \leq f(x)$ (cas d'un problème de minimisation) : $f(x^*) = \min_{x \in S} f(x)$.

Minimum local : une solution réalisable x_0 telle que :
$\exists\, V_0 \subseteq S : x_0 \in V_0$ et $\forall x \in V_0 : f(x_0) \leq f(x) : f(x_0) = \min_{x \in V0} f(x)$.

2.3. Domaines d'application

Les problèmes d'optimisation combinatoire représentent un outil très puissant d'aide à la décision dans les institutions économiques, sociales, industrielles, … et trouvent leur application dans pas mal de domaines entre autres :

- Services publics : hôpitaux, transport public, informatique ;
- Industrie : automobile, aviation, énergie, télécom, production ;
- Finances : gestion de portefeuille ;
- Militaire : gestion des ressources, logistique ;
- L'économie : planification de projets, organisation d'activités, affectation de tâches, les problèmes de transport.
- L'informatique : conception de programmes. Implémentation de systèmes informatiques.
- La sociologie : modélisation et étude de phénomènes sociaux.

2.4. Exemples de problèmes d'optimisation combinatoire

Les problèmes d'optimisation combinatoire en général se caractérisent par le fait qu'ils peuvent être modélisés par des formalismes mathématiques qui représentent en quelques sortes des méta problèmes dont la résolution implique la résolution de toute une classe de problèmes, parmi ces modèles on trouve:

- Problème du voyageur de commerce ;
- Problème du sac-à-dos ;
- Problème du plus court chemin ;
- Problème d'arbre minimal ;
- Problème du plus court chemin ;
- Problème du chemin hamiltonien ;
- Problème du chemin eulérien ;
- Problème de coloration de graphes;…

Ainsi, pour résoudre un problème d'optimisation combinatoire, il suffit de le projeter dans cet ensemble de problèmes pour trouver son modèle mathématique et faciliter alors sa formulation et sa résolution. Pour mieux expliquer cette démarche nous présentons ci-après quelques exemples de problèmes :

2.4.1. Recherche d'un itinéraire de moindre cout dans un réseau routier

Etant donné un réseau routier constitué d'un ensemble de villes et d'un ensemble de routes permettant d'aller d'une ville à une autre, avec au plus une route entre deux villes. Un coût est associé à chaque route, représentant sa longueur, ou le temps, ou encore la consommation

d'énergie nécessaire à son parcours, supposé indépendant du sens de ce parcours (de A vers B ou de B vers A) et supposé additif (le coût d'un itinéraire est égal à la somme des coûts des routes qu'il utilise). (Figure 1.4).

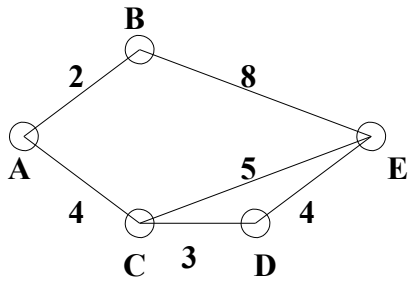


Figure 1.4. Exemple de réseau routier.

Problème : étant donné deux villes, déterminer un itinéraire de moindre coût de la première à la seconde. Un exemple d'un tel itinéraire entre les villes A et E est de coût 9 (Figure 1.5.) dans le réseau de la figure 1.4.

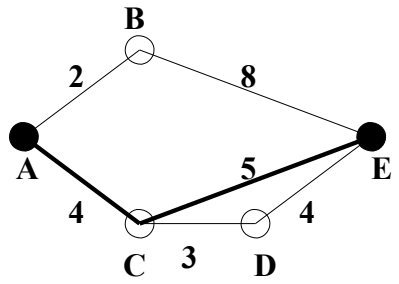


Figure 1.5. Exemple d'itinéraire de moindre coût.

Ce problème se modélise naturellement comme un problème de recherche d'un plus court chemin dans un graphe non orienté pondéré (Shortest Path Problem) [CAV 06b]. Ce dernier est certainement l'un des problèmes d'optimisation combinatoire les plus connus, essentiellement du fait de ses nombreuses applications et des algorithmes efficaces permettant de le résoudre [CAV 06b].

2.4.2. Organisation de la tournée d'un cadre commercial

Considérons un réseau routier similaire au précédent (Figure 1.6.).
Déterminer un itinéraire de moindre coût permettant à un cadre commercial de visiter chaque ville une fois et une seule et de revenir à son point de départ (Figure 1.7.).
Ce problème, connu sous le nom de Problème du Voyageur de Commerce (PVC) (en anglais TSP pour : Traveling Salesman Problem), se modélise naturellement comme un problème de recherche d'un plus court-circuit hamiltonien dans un graphe non orienté pondéré [CAV 06b].

Malgré sa proximité avec le problème précédent (Shortest Path Problem), ce dernier constitue l'un des problèmes classiques les plus difficiles en optimisation combinatoire. Il n'existe pas d'algorithme permettant de le résoudre efficacement de façon optimale (avec garantie d'optimalité) et même un problème apparemment plus simple (déterminer s'il existe un circuit hamiltonien dans un graphe non orienté) est aussi difficile.

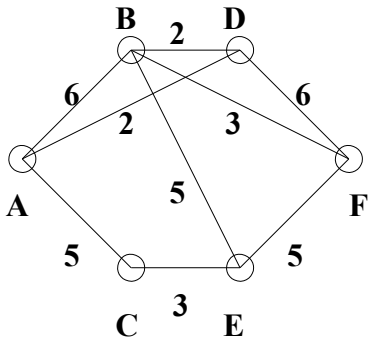


Figure 1.6. Un autre exemple de réseau routier.

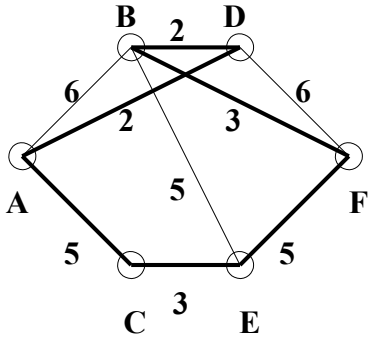


Figure 1.7. Exemple de tournée de moindre coût.

2.4.3. Organisation de la tournée d'une flotte de véhicules de livraison

Il s'agit d'une variante du problème précédent où les villes sont des magasins et une ville particulière est le dépôt en charge de l'approvisionnement des magasins. On suppose disposer d'un ensemble de véhicules de livraison.

Le problème est de déterminer l'itinéraire de chaque véhicule (partant du dépôt et y revenant) de telle façon que l'ensemble des magasins soient approvisionnés par un véhicule et un seul et qu'un critère qui peut être la somme ou le maximum sur l'ensemble des véhicules des coûts des itinéraires soit minimum. Voir la figure 1.9 pour un exemple d'un tel itinéraire à deux véhicules de coût 19 (somme des coûts des deux itinéraires) dans le réseau de la figure 1.8. Des variantes

de ce problème peuvent prendre en compte le volume des commandes des magasins, leurs horaires d'ouverture et les capacités des véhicules.

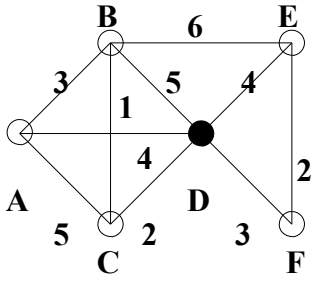


Figure 1.8. Un exemple de réseau routier avec un dépôt.

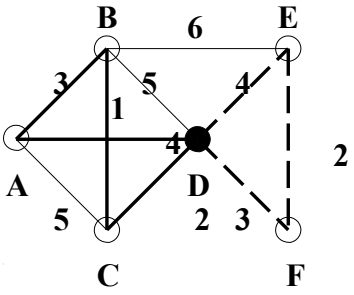


Figure 1.9. Exemple de tournée de moindre cout à deux véhicules.

Connu sous le nom de Vehicle Routing Problem et au moins aussi difficile que le problème précédent, ce problème a fait et fait encore l'objet de nombreuses études du fait de son intérêt pratique dans le domaine de la logistique.

2.5. Complexité et classes de problèmes

Résoudre automatiquement un problème d'optimisation combinatoire nécessite sa définition, sa modélisation et sa formulation mathématique puis le choix d'une méthode appropriée qui peut se traduire à un algorithme informatique. L'exécution de cet algorithme exige la réquisition de deux ressources majeures de l'ordinateur, en l'occurrence : l'espace mémoire nécessaire pour stocker les données du problème et le temps processeur nécessaire pour effectuer les opérations de la méthode en question. L'estimation de la première ressource est dite complexité spatiale de la méthode, celle de la seconde est dite sa complexité temporelle. En vertu du développement spectaculaire du matériel informatique au fil des années qui a induit la dominance du coté soft du cout de l'ordinateur au détriment du coté hard, on ne s'intéresse plus à la complexité spatiale de la méthode. Ainsi, on se contente de caractériser un problème uniquement par sa complexité temporelle qui est désignée, par abus de langage, par le terme complexité tout court.

On entend ici par «complexité d'un problème» une estimation du nombre d'opérations élémentaires à exécuter pour résoudre les instances de ce problème, cette estimation étant un ordre de grandeur par rapport à la taille de l'instance. Il s'agit là d'une estimation dans le pire des cas dans le sens où la complexité d'un problème est définie en considérant son instance la plus délicate. Les travaux théoriques dans ce domaine ont permis d'identifier différentes classes de problèmes en fonction de la complexité de leur résolution [PAP 94].

Problèmes de décision.

Les classes de complexité ont été introduites pour les problèmes de décision, c'est-à-dire les problèmes posant une question dont la réponse est «oui» ou «non». Pour ces problèmes, on définit notamment les deux classes P et NP :

– La classe P contient l'ensemble des problèmes polynomiaux, i.e., pouvant être résolus par un algorithme de complexité polynomiale (majorée par un polynôme). Cette classe caractérise l'ensemble des problèmes que l'on peut résoudre « efficacement ».

La classe NP contient l'ensemble des problèmes polynomiaux non déterministes, i.e., pouvant être résolus par un algorithme de complexité polynomiale pour une machine non déterministe (que l'on peut voir comme une machine capable d'exécuter en parallèle un nombre fini d'alternatives). Intuitivement, cela signifie que la résolution des problèmes de NP peut nécessiter l'examen d'un grand nombre (éventuellement exponentiel) de cas, mais que l'examen de chaque cas doit pouvoir être fait en temps polynomial.

Les problèmes NP-complets sont des problèmes combinatoires dans le sens où leur résolution implique l'examen d'un nombre exponentiel de cas. Notons que cette classe des problèmes NP-complets contient un très grand nombre de problèmes. Pour autant, tous les problèmes combinatoires n'appartiennent pas à cette classe. En effet, pour qu'un problème soit NP-complet, il faut qu'il soit dans la classe NP, i.e., que l'examen de chaque cas puisse être réalisé efficacement, par une procédure polynomiale. Si on enlève cette contrainte d'appartenance à la classe NP, on obtient la classe plus générale des problèmes NP-difficiles, contenant l'ensemble des problèmes qui sont « au moins aussi difficiles » que n'importe quel problème de NP, sans nécessairement appartenir à NP.

La théorie de la complexité montre que si un problème est NP-difficile, alors il est illusoire de chercher un algorithme polynomial pour ce problème (à moins que P = NP). Toutefois, ces travaux sont basés sur une évaluation de la complexité dans le pire des cas, car la difficulté d'un problème est définie par rapport à son instance la plus difficile. En pratique, si on sait qu'on ne pourra pas résoudre en temps polynomial toutes les instances d'un problème NP-difficile, il apparait que certaines instances sont beaucoup plus faciles que d'autres et peuvent être résolues très rapidement.

Une caractéristique fort intéressante de ce phénomène est que les instances les plus difficiles à résoudre se trouvent dans la zone de transition [CKT 91], [HOG 96], [MPSW 98], là où les instances ne sont ni trivialement solubles ni trivialement insolubles. Il s'agit là d'un pic de

difficulté dans le sens où une toute petite variation du paramètre de contrôle permet de passer d'instances vraiment très faciles à résoudre à des instances vraiment très difficiles. Notons également que ce pic de difficulté est indépendant du type d'approche de résolution considérée [DAV 95], [CFG 96].

De nombreux travaux, à la fois théoriques et expérimentaux, se sont intéressés à la localisation de ce pic par rapport au paramètre de contrôle. En particulier, [GMP 96] introduit la notion de « degré de contrainte » (« constrainedness ») d'une classe de problèmes, noté, et montre que lorsque est proche de 1, les instances sont critiques et appartiennent à la région autour de la transition de phase. Ces travaux sont particulièrement utiles pour la communauté de chercheurs s'intéressant à la résolution pratique de problèmes combinatoires. Ils permettent en particulier de se concentrer sur les instances vraiment difficiles, qui sont utilisées en pratique pour valider et comparer les approches de résolution. Ces connaissances peuvent également être utilisées comme des heuristiques pour résoudre plus efficacement les problèmes NP-complets [HOG 98].

A chaque algorithme de résolution est ainsi associée une complexité qui peut être généralement corrélée au problème lui-même. En termes mathématiques, le mathématicien Landau définit une notation qui exprime une grandeur globale de la complexité. C'est une mesure formelle indépendante de l'ordinateur, du système d'exploitation et du langage de programmation utilisés pour implémenter une méthode de résolution du problème en question. Il s'agit des notations O, Ω, Θ, qui permettent respectivement de majorer, minorer ou cadrer la valeur de la complexité mais juste asymptotiquement, puisque, ce qui importe c'est le comportement de l'algorithme quand la taille du problème est assez grand.

Définitions

Soit f et g deux fonctions réelles.

On dit que f est en Ω (g) si et seulement si: $\exists c \in R+, \exists n_0 \in N, \forall n \geq n_0$: $f(n) \geq c.g(n)$.

On dit que f est en O(g) si et seulement si: $\exists c \in R+, \exists n_0 \in N, \forall n \geq n_0$: $f(n) \leq c.g(n)$.

On dit que f est en Θ (g) si et seulement si: $\exists c,d \in R+, \exists n_0 \in N, \forall n \geq n_0$: $c.g(n) \leq f(n) \leq d.h(n)$.

Si la complexité d'un algorithme est en O(g) est que g(n) est un polynôme, alors l'algorithme est polynomial et le problème pouvant être résolu par cet algorithme est ainsi facile.

3. Taxonomie des méthodes de résolution

Les méthodes de résolutions des problèmes d'optimisation sont généralement classifiées en deux grandes catégories : celles d'optimisation combinatoire et celles d'optimisation continue. Les méthodes d'optimisation continue sont classifiées en linéaires et non-linéaires, celles de l'optimisation combinatoire sont classifiées en exactes et approchées. (Figure 1.10)

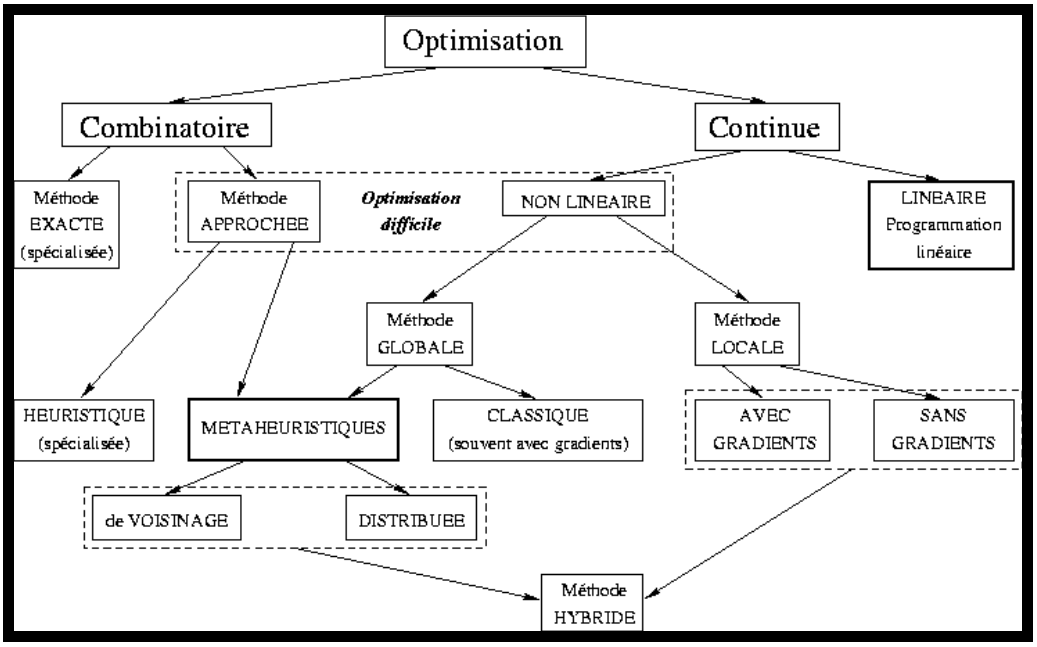


Figure 1.10. Classification générale des méthodes de résolution.

Selon la définition donnée ci-dessus, la résolution d'un problème d'optimisation combinatoire nécessite d'abord sa formulation mathématique qui consiste à bien définir :

- L'espace de solutions ou la variable balayant cet espace ;
- La fonction objectif à optimiser ;
- Les contraintes que devraient satisfaire les solutions à explorer ;
- Les paramètres éventuels du modèle à utiliser pour la résolution du problème.

Trouver une solution optimale dans un ensemble discret et fini est un problème théoriquement facile : il suffit d'examiner toutes les instances, et de comparer leurs qualités pour en dégager la meilleure. Cependant, en pratique, l'énumération de toutes les instances peut prendre trop de temps ; or, le temps de recherche de la solution optimale est un facteur très important et c'est à cause de ce temps que les problèmes d'optimisation combinatoire sont réputés si difficiles. La théorie de la complexité fournit des outils pour mesurer ce temps de recherche. De plus, comme l'ensemble des solutions réalisables est défini de manière implicite, il est aussi parfois très difficile de trouver, ne serait-ce, qu'une solution réalisable.

Quelques problèmes d'optimisation combinatoire peuvent être résolus (de manière exacte) en temps polynomial par exemple par un algorithme glouton, un algorithme de programmation dynamique ou en montrant que le problème peut être formulé comme un problème d'optimisation linéaire en variables réelles.

En pratique, la complexité physiquement acceptable n'est souvent que polynômiale. C'est pourquoi, parfois, on se contente alors d'avoir une solution approchée au mieux, obtenue par une heuristique ou une métaheuristique.

Pour le problème du VRP envisagé à large échelle, Karp zone les villes, résout le problème zone par zone, puis assemble au mieux les parcours locaux.

Pour un problème d'optimisation d'affectation d'équipages, l'optimisation linéaire en variables réelles dégrossit le problème : on considère comme définitives les valeurs des variables ainsi collées aux extrêmes de leur domaine, et on n'applique les méthodes d'optimisation combinatoire que sur le problème résiduel.

Pour certains problèmes, on peut prouver une garantie de performance, c'est-à-dire que pour une méthode donnée l'écart entre la solution obtenue et la solution optimale est borné. Dans ces conditions, à domaine égal, un algorithme est préférable à un autre si, à garantie égale ou supérieure, il est moins complexe.

CHAPITRE II

METHODES EXACTES

1. Introduction

Les méthodes exactes sont généralement préconisées pour résoudre les problèmes de petites tailles pour lesquelles le nombre de combinaisons possibles est suffisamment faible pour pouvoir explorer l'espace de solutions dans un temps raisonnable. Par exemple la technique de séparation et d'évaluation (connue sous le nom de branch & bound) permet l'exploration de l'espace de recherche par la construction d'une structure arborescente du problème qui se décompose en sous problèmes.

Cette arborescence est explorée de façon à éviter les branches ne contenant pas des solutions réalisables et les branches n'amenant pas à des solutions meilleures que la solution courante. Cette méthode a été utilisée la première fois par Johnson en 1981 pour résoudre le problème d'équilibrage des lignes d'assemblage [JOH 81].

Les méthodes exactes visent à trouver de manière certaine la solution optimale en examinant de manière explicite ou implicite la totalité de l'espace de recherche. Elles ont l'avantage de garantir la solution optimale néanmoins le temps de calcul nécessaire pour atteindre cette solution peut devenir très excessif en fonction de la taille du problème (explosion combinatoire) et le nombre d'objectifs à optimiser. Ce qui limite l'utilisation de ce type de méthodes aux problèmes bi-objectifs de petites tailles. Ses méthodes génériques sont : Branch & Bound, Branch & Cut, Branch & Price, Branch, Cut & Price.

D'autres méthodes sont moins générales, comme : La programmation dynamique, simplexe, la programmation linéaire en nombre entiers, l'algorithme A*. D'autres méthodes sont spécifiques à un problème donné comme l'algorithme de Johnson pour certain problèmes d'ordonnancement.

2. Approches complètes

Pour résoudre un problème combinatoire, on peut explorer l'ensemble des configurations de façon exhaustive et systématique, et tenter de réduire la combinatoire en utilisant des techniques d'«élagage», visant à éliminer le plus tôt possible des sous-ensembles de configurations en prouvant qu'ils ne contiennent pas de solutions, et des heuristiques de choix
visant à se diriger le plus rapidement possible vers les meilleures configurations. Ces approches sont complètes dans le sens où elles sont toujours capables de trouver la solution optimale ou, le cas échéant, de prouver l'absence de solutions.

En revanche, le temps de résolution dépend de l'efficacité des techniques d'élagage et des heuristiques considérées, et est exponentiel dans le pire des cas. Un point clé de ces approches

réside dans la façon de structurer l'ensemble de configurations, l'objectif étant de pouvoir éliminer des sous-ensembles de configurations de façon globale, sans avoir à examiner les configurations qu'ils contiennent une par une.

Exemple :

Résoudre le problème du sac-à-dos simple à n objets par cette méthode nécessite l'énumération de 2^n cas. Résoudre le problème du voyageur de commerce symétrique en nécessite (n-1)!/2 cas. Cahacun de ces deux problemes est en $O(2^n)$.

3. Programmation dynamique

La programmation dynamique est une technique algorithmique qui permet de résoudre une catégorie particulière de problèmes d'optimisation sous contraintes. Elle a été désignée par ce terme pour la première fois dans les années 1950 par le professeur Richard Bellman. Elle s'applique à des problèmes d'optimisation dont la fonction objectif se décrit comme "*la somme de fonctions monotones non décroissantes des ressources*".

La Programmation Dynamique est séduisante car son formalisme est assez générique, ce qui laisse libre cours à de multiples variantes et à la résolution de problèmes assez variés, qu'ils soient déterministes ou non. Cependant, seule la catégorie des problèmes d'optimisation séquentielle est concernée et la vérification de l'applicabilité du principe d'optimalité est indispensable avant toute utilisation de cette méthode.

La programmation dynamique est similaire à la méthode "*diviser pour 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 pour régner*" crée des Sous-problèmes qui sont complètement séparés et peuvent être résolus indépendamment l'un de l'autre.

Procédé général

a) Obtention de l'équation récursive liant la solution d'un problème à celles des sous-problèmes.
b) Initialisation de la table : cette étape est donnée par les conditions initiales de l'équation obtenue à l'étape a.
c) 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 a.
d) 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 général, 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 à l'étape c.

Algorithme 2.1. Algorithme général de la programmation dynamique.

Toutefois, ce ne sont pas tous les problèmes d'optimisation qui peuvent être résolus à l'aide de la programmation dynamique. Le problème à résoudre doit satisfaire le principe d'optimalité pour que la solution obtenue par la programmation dynamique soit optimale. (On reviendra dans le chapitre IV sur cette méthode avec plus de détails).

3.1. Historique

En dépit des travaux de Pierre Massé en 1944 pour planifier les investissements d'électricité de France [SAK84], qui avait pourtant, une idée de ce genre, la programmation dynamique est considérée comme le fruit des recherches menées dans les années 50 par le mathématicien américain Richard Bellman, chercheur à la Rand Corporation, pour résoudre certains problèmes d'affectation et d'optimisation [TIS05]. C'est grâce à son fameux théorème d'optimalité que la programmation dynamique vit le jour en 1957 et c'est lui qui l'a baptisée ainsi [SAK84]. En fait, les travaux de Bellman sont intervenus peu après que la découverte de l'algorithme du simplexe par G. B. Dantzig en 1947 ait donné un grand élan à la programmation linéaire [SAK84]. C'est certainement pour des raisons de mode que la programmation dynamique a été dénommée ainsi. L'adjectif « *dynamique* », outre qu'il offre une image avantageuse de cette technique, s'explique par le fait qu'il s'agit d'une méthode d'optimisation séquentielle. Pour certains spécialistes en la matière, le terme « optimisation récursive » eût été plus approprié car il résume excellemment l'idée de l'approche. Pour d'autres, le terme « programmation dynamique » peut paraître un peu étrange et est maintenant parfois utilisé dans un autre sens et que Bellman l'a choisi dans un souci de communication : son supérieur ne supportait ni le mot « recherches » ni le mot « mathématiques », il lui a semblé que l'utilisation du terme « programmation dynamique » donnait à son travail une apparence qui plaisait à son supérieur [TIS 05]. En 1962, Held et Karp [HEL 62] suivirent les pas de Bellman et appliquèrent l'idée de

la programmation dynamique à certains problèmes d'ordonnancement à une machine. Par la suite, White, en 1969, élabora une formulation générale de cette technique en étendant son application à plusieurs domaines [FRE 82].

3.2. Principe

Le théorème d'optimalité de Bellman, considéré comme le noyau de la programmation dynamique, est si simple qu'il paraît presque trivial lorsqu'on l'aura bien compris. Son importance et l'efficacité des méthodes d'optimisation séquentielles auxquelles il a donné naissance par la suite s'accentuent au fur et à mesure que l'on s'aperçoit que la vraie nature de nombreux problèmes est séquentielle : ils peuvent être décomposés en phases (donc en sous problèmes), chacune d'elles ne dépendant que de ses voisines les plus proches, et dans les cas favorables, seulement de l'antérieure ou de la postérieure. Autrement dit, Si la fonction objectif à optimiser satisfait cette hypothèse, alors le problème sera décomposable en phases ou *périodes* définissant chacune un sous problème dont la solution constitue une composante de la solution globale du problème tout entier.

A chacune d'elles correspond une solution dite *sous-politique (la solution globale est alors dite politique)*. Il s'agit donc d'un système qui évolue par incréments dans le temps d'un état initial e_i jusqu'à un état final e_f, en transitant par des états intermédiaires :

e_{11}, e_{12},… états possibles à la phase 1
e_{21}, e_{22},… états possibles à la phase 2
……
e_{k1}, e_{k2},… états possibles à la phase k
……..……
$e_{(n-1)1}$, $e_{(n-1)2}$,… états possibles à la phase n-1

Le nombre d'états diffère d'une phase à une autre selon le problème en question. Cette décomposition est dans la plupart des cas naturelle ou temporelle, c'est-à-dire que le temps est effectivement une variable de l'évolution du système, par contre, dans d'autres, elle est artificielle, c'est-à-dire que le temps n'intervient pas explicitement dans l'évolution du système. La transition d'un état de l'étape k à un état de l'étape k+1 est appelée *décision*.

A chaque décision u correspond une valeur de la fonction objectif qui représente un coût à minimiser ou un profit à maximiser (selon la fonction objectif). (Figure. 2.1).

Très grossièrement, c'est cette décomposition en sous-problèmes qui permet de la grande économie de calcul que procure la programmation dynamique vis-à-vis de la méthode à exploration exhaustive.

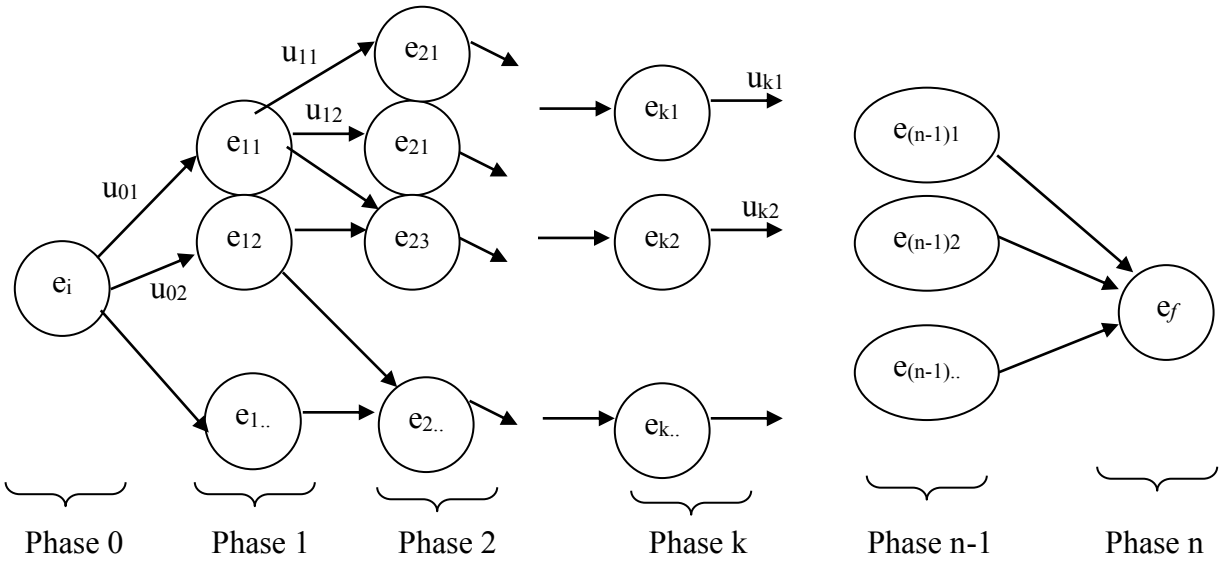


Figure. 2.1. Schéma général de la décomposition d'un problème d'optimisation.

La résolution du problème consiste alors à trouver une séquence de décisions ($u_0,u_1,u_2,\ldots,u_n$) dite *politique optimale* réalisant la valeur optimale de la fonction objectif. Si la valeur de cette dernière, en se trouvant à un état donné, ne dépend du passé qu'à travers son état prédécesseur (ou de son état successeur), cela veut dire que si la prise d'une décision à une étape donnée ne remet pas en cause les décisions des étapes antérieures (respectivement postérieures), alors la décomposition du problème sera possible. Ceci se traduit par l'établissement d'une relation de récurrence entre l'objectif d'une étape et celui de son voisin, ce qui permet de concrétiser le "*diviser pour régner*". Cette condition est appelée *hypothèse de Markov* [CAR88]. Cette hypothèse, qui est généralement satisfaite pour les fonctions additives, peut l'être également pour certaines fonctions multiplicatives ou autres.

3.3. Processus de décisions séquentiel

Si la fonction objectif satisfait l'hypothèse de Markov, on peut schématiser cette décomposition par un modèle mathématique : *le processus de décisions séquentiel*.

Un processus de décisions séquentiel est un triplet (X,U,g) où [CAR88] :

- X : un ensemble appelé espace des états ;
- U : un ensemble appelé espace des décisions ;
- g : une application de $X\times U\times N$ dans $X \cup\{\varepsilon\}$;

N : ensemble des entiers naturels ;

ε est l'état « null » ou « vide » et g est la fonction de transition du graphe ainsi formé.

L'ensemble des états de la phase k est noté : x(k) ;

$x(0)= \{e_i\}$ et pour $k>0$: $x(k)= \{e\in X : \exists s\in X, u\in U / e=g(s,u,k-1)\}$;

L'ensemble des décisions qui peuvent être prises à la phase k pour l'état e est noté $\Delta(e,k)$;

$\Delta(e,k) = \{ u \in U: g(e,u,k) \neq \varepsilon\}$;

Il s'agit alors d'un graphe valué ou réseau où les sommets sont les états du système, les arcs sont les décisions qui peuvent être prises et la fonction des couts des arcs est g.

Le nombre de phases est appelé *horizon* [CAR88] [ROB79] du processus de décisions séquentiel. Il peut être fini ou infini, selon le problème à traiter. Au cas où l'horizon est fini, le processus de décisions séquentiel est noté par le quadruplet (X,U,g,n). où n est l'horizon.

Le processus de décisions séquentiel est ainsi considéré comme un modèle mathématique par lequel on schématisera les problèmes d'optimisation évolutifs. En fait, il s'agit d'un outil puissant permettant de représenter les problèmes décomposables.

3.4. Sous-politique / Politique

Etant donné un processus de décisions séquentiel (X,U,g,n).

On appelle sous-politique d'ordre k pour l'état e de l'ensemble x_{n-k}, l'arrangement $(e,u_0,u_1,\ldots,u_{k-1})$ tel que : $u_0,u_1,u_2,\ldots, u_{k-1}$ sont des décisions :

$u_0 \in \Delta(e,n-k)$

$u_1 \in \Delta(e_1,n-k+1)$ avec $e_1 = g(e,u_0,n-k)$

$u_2 \in \Delta(e_2,n-k+2)$ avec $e_2 = g(e_1,u_1,n-k+1)$

……..

$u_{k-1} \in \Delta(e_{k-1},k+1)$ avec $e_{k-1} = g(e_{k-2},u_{k-2},k-2)$ [CAR88]

D'une manière littéraire simple, une sous-politique pour l'état e du système est la séquence de décisions menant à l'état final à partir de l'état e.

Une politique est une sous-politique d'ordre n [CAR 88]. C'est donc une sous-politique pour l'état initial e_i. Il faut remarquer ici que le terme « *politique* » en programmation dynamique substitut le terme « *solution* » en programmation linéaire. Ceci reflète l'aspect évolutif du problème et le caractère dynamique des solutions [ROB 79].

3.4.1. Sous-politique optimale / Politique optimale

L'objectif primordial des politiques admissibles d'un système est de pouvoir les évaluer et, par conséquent, pouvoir choisir celle qui est qualifiée de meilleure. Ainsi, chaque problème d'optimisation est caractérisé par une fonction à optimiser (minimiser ou maximiser selon les cas) dite « *fonction objectif* » ou encore « *critère* ». En général, il s'agit d'une fonction réelle F de l'ensemble P des sous-politiques dans R^+.

Une sous-politique p* d'ordre k est dite optimale si :

$\forall p \in P_k : F(p^*) \leq F(p)$ i.e. : $F(p^*) = \min_{p \in \mathcal{P}_k} \{ F(p)\}$ pour une fonction à minimiser;

$\forall p \in P_k : F(p^*) \geq F(p)$ i.e. : $F(p^*) = \max_{p \in \mathcal{P}_k} \{ F(p)\}$ pour une fonction à maximiser;

En général : $F(p^*) = \underset{p \in \mathcal{P}_k}{Opt}\{ F(p)\}$;

Où P_k est l'ensemble des sous-politiques d'ordre k. ($P_k \subset P$).

Une politique optimale est une sous-politique optimale d'ordre n.

3.4.2. Hypothèse de Markov

Soient (X,U,g,n) un processus de décisions séquentiel, F la fonction objectif associée. $(u_0,u_1,\ldots,u_{k-1}, u_k,u_{k+1},\ldots,u_{n-1})$ et $(u_0,u_1,\ldots,u_{k-1}, v_k,v_{k+1},\ldots,v_{n-1})$ deux politiques ayant le même préfixe $(u_0,u_1,\ldots,u_{k-1})$ jusqu'à un état e de la phase k.

Si $F(e, u_k,u_{k+1},\ldots,u_{n-1})$ est meilleure que $F(e, v_k,v_{k+1},\ldots,v_{n-1})$

alors $F(u_0,u_1,\ldots,u_{k-1}, u_k,u_{k+1},\ldots,u_{n-1})$ est meilleure que $F(u_0,u_1,\ldots,u_{k-1}, v_k,v_{k+1},\ldots,v_{n-1})$.

Le terme « est meilleure que » désigne ici :

- « ≤ » dans le cas de minimisation ;
- « ≥ » dans le cas de maximisation ;

Plus concrètement, cette propriété implique que si deux politiques u et v ont le même préfixe d'ordre k (mêmes k premières décisions) et si à partir de l'état résultant e de ces k premières décisions, la sous-politique $(e, u_k,u_{k+1},\ldots,u_{n-1})$ d'ordre n-k pour l'état e est meilleure que la sous-politique $(e, v_k,v_{k+1},\ldots,v_{n-1})$ d'ordre n-k pour l'état e alors u est meilleure que v.(Figure. 2.2.).

La quasi-totalité des fonctions objectifs satisfait cette propriété dite hypothèse de Markov [CAR88] et c'est cette hypothèse qui autorise la décomposition du problème et, par la suite, l'application de programmation dynamique.

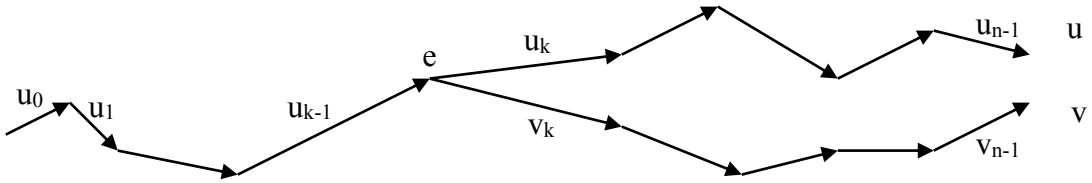


Figure. 2.2. Deux politiques ayant même préfixe illustrant la propriété de Markov.

Bien qu'elle paraisse naturelle, cette hypothèse n'est pas justifiée pour tous les problèmes. Dans un graphe, par exemple, lorsqu'on cherche le plus court chemin entre deux points : si le chemin le plus court entre A et B passe par C, alors le tronçon de A à C est le chemin le plus court entre A et C, i. e. l'hypothèse est tout à fait satisfaite. Par contre, elle ne l'est pas si on cherche le plus long chemin sans boucle entre deux points. En fait, si le chemin le plus long entre

A et B passe par C, alors le tronçon de A à C n'est pas forcément le chemin le plus long entre A et C.

3.4.3. Principe d'optimalité / Théorème d'optimalité

En bref, Bellman exprime ceci [ROB 79]:

« *Toute politique optimale ne peut être formée que de sous-politiques optimales.* »

Ceci s'exprime aussi par :

« *Dans une séquence optimale de décisions, quelle que soit la première décision prise, les décisions subséquentes forment une sous-séquence optimale, compte tenu des résultats de la première décision.*» [SAK 84]

En d'autres termes [ROB 79] :

« *Tout chemin optimal est constitué de portions de chemins elles-mêmes optimales.*»

Pour donner une démonstration par l'absurde simple et brève, on peut s'exprimer brièvement comme suit :

S'il n'en était pas ainsi pour une portion quelconque, celle-ci pourrait être remplacée par une autre meilleure, et, par conséquent, le chemin ne serait pas optimal, contrairement à l'hypothèse.

3.4.4. Récurrence avant. Récurrence arrière

Dans la résolution de certains problèmes, quand on procède par la méthode de la programmation dynamique, deux approches sont applicables [FRE82] :

Une approche descendante dite « *récurrence avant* » (forward dynamic programming) dans laquelle on procède comme suit : en partant de l'état initial du système, on résout les sous-problèmes d'ordres 1,2,…,n-1,n ; on déterminera ainsi les sous-politiques optimales respectives d'ordres 1,2,…,n-1,n.
Ainsi, on choisira la première décision de la politique optimale, puis la seconde, et ainsi de suite jusqu'à la n-ième décision.

Une approche ascendante dite « *récurrence arrière* » (*backward dynamic programming*) dans laquelle on procède comme suit : en partant de l'état final du système, on résout les sous-problèmes d'ordres 1,2,…,n-1,n ; on déterminera ainsi les sous-politiques optimales respectives d'ordres 1,2,…,n-1,n ; Ainsi, on choisira la dernière décision de la politique optimale, puis l'avant dernière, et ainsi de suite jusqu'à la première décision.

Il est clair que la première approche est toujours applicable et plus simple à modéliser, tandis que la seconde n'est pas applicable dans les problèmes en horizon infini et plus complexe

dans les cas stochastiques. On peut procéder indifféremment par les deux approches dans les problèmes « non ordonnés », néanmoins dans les problèmes ordonnés ou partiellement ordonnés, seule la première approche est faisable. Dans le cadre de ces conditions, on peut formuler les deux algorithmes généraux suivants (forward et backward) dans lesquels on tentera de donner une démarche abstraite à appliquer pour implémenter la plupart des cas :

Algorithme AVANT((X,U,g,n),F,p*,Opt)
Entrées : ***un PDS (X,U,g,n), une fonction objectif F***
Variables : ***Min : un réel, u : une décision, e : un état, p : une sous-politique.***
Sorties : ***une politique optimale p* =(u_0,u_1,….,u_{n-1}), une solution optimale Opt =F(p*).***
Dédut
p* = ∅
Opt = 0
Pour k = 0 à n-1
Pour tout e ∈ X(k)
Min=∞
Pour tout u ∈ Δ(e,k)
p=p*∪{u}
Si F(p) < Min alors
Min = F(p)
u_k = u
fin si
Fin pour
Fin pour
p* = p*∪{u_k}
Opt = Opt+ Min
Fin pour
Fin

Algorithme 2.2. Programmation dynamique en récurrence avant.

Remarque

Comme on peut le remarquer, cet algorithme s'applique aux problèmes de minimisation. Dans un problème de maximisation, l'initialisation « Min=∞ » devrait être remplacée par : « Max=0 » et la comparaison : « Si F(p)<Min » deviendrait : « Si F(p)>Max ».

Algorithme ARRIERE((X,U,g,n),F , p*,Opt)
Entrées : un PDS (X,U,g,n), une fonction objectif F
Variables : Min : un réel, u : une décision, e : un état, p : une sous-politique.
Sorties : une politique optimale p* =(u_0,u_1,….,u_{n-1}), une solution optimale Opt =F(p*).
Dédut

```
p*=∅
Opt = 0
Pour k=n-1 à 1 pas -1
    Pour tout e ∈ X(n-k)
            Min = ∞
        Pour tout u∈ Δ(e,n-k)
            p = p*∪{u}
            Si  F(p) < Min alors
                    Min = F(p)
                    uk = u
            fin si
        Fin pour
    Fin pour
  p* = p*∪{uk}
  Opt = Opt + Min
Fin pour
```

Fin

Algorithme 2.3. Programmation dynamique en récurrence arrière.

Ces algorithmes constituent des modèles généraux pour une certaine classe de problèmes, vu leur caractère générique. Pour leur mise au point, on aura recours aux spécifications du problème concerné (par exemple Opt peut être un entier ou un réel selon les problèmes), et aux structures de données adéquates pour représenter les décisions, les états et les sous-politiques. Il est clair que ces deux algorithmes sont itératifs. Une forme récursive pourrait également être tentée, elle aurait été plus appropriée, on l'a évitée ici dans un souci de simplification, on l'utilisera au moment de l'implémentation des algorithmes. La forme récursive est beaucoup plus simple à programmer, mais en termes de complexité temporelle, elle est plus lente que la forme itérative.

3.4.5. Complexité

Pour évaluer la complexité temporelle de ces deux algorithmes, nous devons d'abord définir l'opération fondamentale qui affecte formellement le temps d'exécution en fonction de la taille n

du problème. Or, ces algorithmes visent à calculer la valeur de l'objectif . Ce calcul se fait en comparant les valeurs successives obtenues. C'est la raison pour laquelle nous prenons la comparaison comme opération fondamentale.

Dans les deux algorithmes ci-dessous, le nombre de comparaisons est proportionnel au nombre de parties d'un ensemble ayant n éléments, c'est 2^n. Il est clair que la comparaison se situe dans deux boucles imbriquées, l'une parcourant les étapes, l'autre parcourant les états de chaque étape, le nombre d'itérations ainsi engendrées est alors n^2. D'où chacun de ces deux algorithmes est en $O(n^2 2^n)$. Ce qui montre clairement le caractère exponentiel de la méthode.

3.5. Un exemple de gestion des stocks

Dans cette section, on se propose de revenir sur les notions et les termes qu'on a évoqués à travers ce chapitre à l'aide d'un exemple illustratif comportant les divers aspects de l'application de la technique de la programmation dynamique. En effet, on traitera un exemple de gestion de stock, où la décomposition est temporelle, Un troisième exemple concernant l'ordonnancement de tâches indépendantes sera traité en détails dans le chapitre quatre y compris son implémentation informatique.

Enoncé

Le tableau suivant donne, pour les quatre périodes qui font l'objet de l'étude, les quantités d'un certain bien qu'un marchand aura à revendre, ainsi que les prix d'achat :

Période i	1	2	3	4
Demande d_i	5	4	3	4
Prix p_i	10	20	15	12

Tableau 2.1. Exemple de plan de stock.

Sachant que ce marchand dispose d'une capacité de stockage de 6 unités (le coût de stock est supposé nul), le problème consiste alors à déterminer une politique optimale d'achat, c'est-à-dire qu'on cherche à déterminer les quantités à acheter à chaque période de manière à minimiser le coût total d'achat pour les quatre périodes. On considère de plus que :

i. les achats se font en début de période ;
ii. on doit stocker les ventes de la période en cours ;
iii. on commence et on termine avec un stock nul ;

Résolution du problème

La décomposition de ce problème est temporelle en quatre périodes, elle est explicite dans l'énoncé, de plus, elle est, en quelque sorte, imposée.

Posons : e_k : la quantité restant en stock à la fin de la $k^{ème}$ période ;

u_k : la quantité achetée au début de la $k^{ème}$ période ;

La valeur de e_k décrit parfaitement l'état du système ; e_k est dite « variable d'état ».

La valeur de u_k caractérise la décision prise ; u_k est dite « variable de décision ».

L'état initial est $e_0 = 0$, l'état final est $e_4 = 0$;

Et pour : $1 \leq k \leq 4$ on a : $0 \leq e_k \leq 6 - d_k$; et $e_k = e_{k-1} + u_k - d_k$;

Ainsi,

$\Rightarrow$ à la première période on a deux états : 0, 1 , ;

$\Rightarrow$ à la deuxième période on a trois états : 0, 1, 2 ;

$\Rightarrow$ à la troisième période on a quatre états : 0, 1 , 2 , 3 ;

$\Rightarrow$ à la quatrième période on a trois états : 0, 1 , 2 ;

Dans ce cadre, une politique est une séquence de 4 décisions (u_0, u_1, u_2, u_3) représentant les quantités achetées aux différentes périodes.

A chaque quadruplet (u_0, u_1, u_2, u_3) est associée une valeur de la fonction objectif représentant le coût total d'achat.

On est donc tenu à déterminer la ou les politiques optimales encourant un coût total d'achat minimal.

Le processus de décisions séquentiel ainsi défini est schématisé par le réseau suivant qui illustre l'évolution de ce système. (Figure. 2.3):

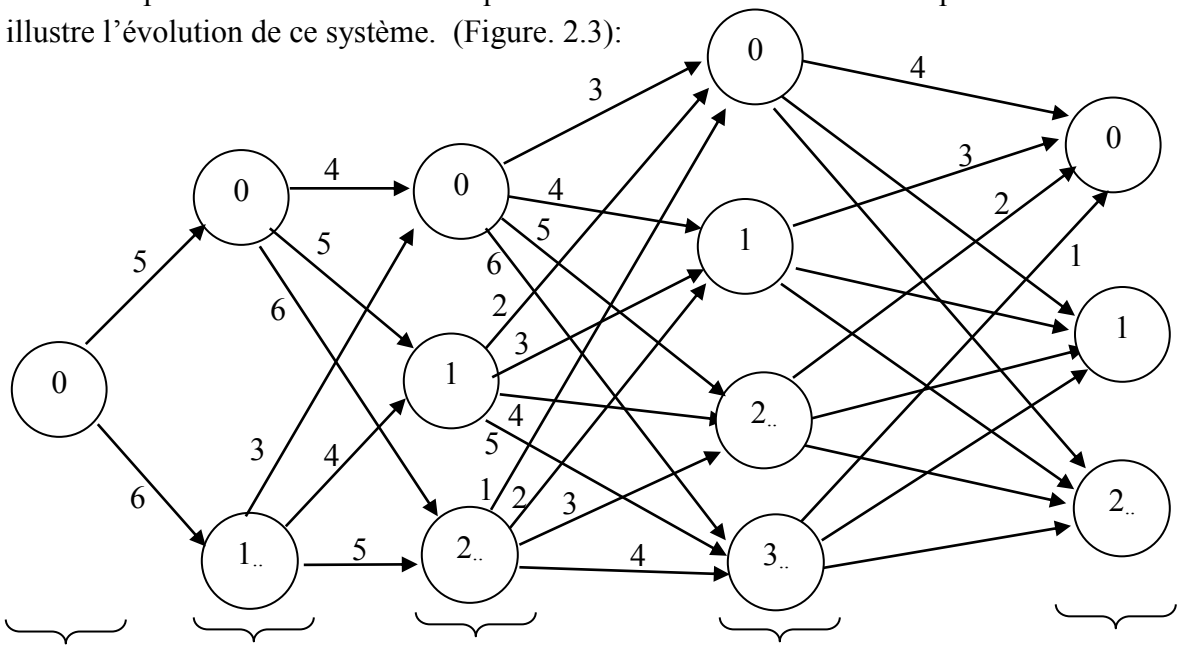


Figure. 2.3. Représentation du problème dynamique de gestion de stock.

A la première période « phase 0 », le stock est nul, la demande est 5, la capacité de stockage étant 6, pour satisfaire cette demande, on peut alors acheter 5 ou 6 unités, ce qui termine la première phase avec un stock de 0 ou 1 unité respectivement.

Donc il y a deux décisions possibles : $u_{01} = 5$; $u_{02} = 6$ qui mènent le système aux deux états respectifs $e_{01} = 0$; $e_{02} = 1$.

Le tableau suivant récapitule la démarche adoptée pour les quatre périodes. Pour chaque période k, est indiquée la demande d_k, les décisions possibles u_k, les états relatifs à cette décision et les couts cumulés en choisissant cette sous-politique. Les lignes marquées d'un * sont celles qui correspondent aux décisions optimales (tableau 2.2.) :

Période k	**Demande d_k**	**Décision u_k**	**Etat e_k**	**Coûts cumulés**
1	5	5	0	50 *
		6	1	60 *
2	4	4	0	50+80=130
		3		60+60=120 *
		5	1	50+100=150
		4		60+80=140 *
		6	2	50+120=170
		5		60+100=160 *
3	3	3	0	130+45=175 *
		2		150+30=180
		1		170+15=185
		4	1	120+60=185 *
		3		140+45=185 *
		2		160+30=190
		5	2	130+75=205 *
		4		150+60=210
		3		170+45=215
		6	3	120+90=210 *
		5		140+75=215
		4		160+60=220
4	4	4	0	175+48=223
		3		185+36=221 *
		2		205+24=229
		1		210+12=222

Tableau 2.2. Résolution des sous-problèmes de gestion de stock.

Ce tableau illustre clairement la procédure de calcul dans chaque phase du problème qui représente explicitement une période d'achat. En se trouvant dans un état e_k, la prise d'une décision u_k, mène le système économique ainsi conçu, à un état e_{k+1} avec une valeur

$f(u_0,u_1,\ldots,u_{k+1})$ de la fonction objectif. En lisant ce tableau du bas en haut, on déduit que la solution optimale est 221 qui correspond à $u_4 = 3$, ceci implique : $e_3 = 1$, avec $u_3 = 4$ ou $u_3 = 3$.
Ceci entraîne : $e_2 = 0$, avec $u_2 = 3$, d'où : $e_1 = 1$, avec $u_1 = 6$.
La politique optimale recherchée est donc : (6,3,3,3) ou bien (6,3,4,3) dont la valeur est 221.

Remarques

- ✓ Le tableau précédent permet non seulement de résoudre le problème donné comportant les quatre périodes, mais aussi tous ses sous problèmes issus des quatre périodes. Par exemple, la solution optimale de la période 2 est 120 avec la sous-politique (6,3).
- ✓ Le tableau comporte 24 lignes, autrement dit, on a résolu 24 sous problèmes pour parvenir à la politique optimale recherchée. En fait, il s'agit des 24 chemins possibles menant de e_0 à e_4. C'est aussi le nombre de quadruplets (a,b,c,d) de décisions qu'on a effectivement explorés. En revanche, dans une méthode énumérative, on aurait exploré 7^4 cas possibles. (on parle ici du nombre d'arrangements de 4 éléments (a,b,c,d) d'un ensemble de 7 éléments {0,1,2,3,4,5,6}). En termes de complexité, l'écart est, évidemment, fatal. (ceci pour, uniquement, quatre périodes. Quel serait cet écart pour un nombre plus grand ?).

On peut étendre ce problème à un horizon infini, voire même considérer des demandes d'achats stochastiques en faisant intervenir les probabilités. Mais, là, il faut faire appel aux outils appropriés, et tenir compte, surtout, de l'actualisation des prix.

4. Méthode par évaluation et séparation

Pour plusieurs problèmes, en particulier les problèmes d'optimisation combinatoire, 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 de solutions : il n'est guère évident d'effectuer cette énumération. La méthode de branch and bound (procédure par évaluation et séparation progressive) consiste à énumérer ces solutions d'une manière intelligente en ce sens que, en utilisant certaines propriétés du problème en question, cette technique arrive à éliminer des solutions partielles qui ne mènent pas à la solution que l'on recherche. De ce fait, on arrive souvent 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. Pour ce faire, cette méthode se dote d'une fonction qui permet de mettre une borne sur certaines solutions pour, soit les exclure, soit les maintenir comme des solutions potentielles. Bien entendu, la performance de branch and bound dépend, entre autres, de la qualité de cette fonction (de sa capacité d'exclure des solutions partielles le plus tôt possible) [IGN 65].

Par convenance, on représente l'exécution de la méthode de branch & bound à travers une arborescence. La racine de cette arborescence représente l'ensemble de toutes les solutions du problème considéré. Pour l'appliquer nous devons disposer :

a) d'un moyen de calcul d'une borne inférieure d'une solution partielle ;

b) d'une stratégie de subdiviser l'espace de recherche pour créer des espaces de recherche de plus en plus étroits ;
c) d'un moyen de calcul d'une borne supérieure pour au moins une solution.

Algorithme général

```
Créer racine racine ;
B&B (racine) ;
B&B (S) {
    Calculer UB(S) et LB(S) ;
    Si UB(S) = LB(S) alors retourner UB(S) ;
    Sinon diviser S en S1 et S2 ;
    Call B&B (S1) ;
    Call B&B (S1) ; }
```

Algorithme 2.4. Algorithme général de B&B.

Si une solution optimale est trouvée pour un sous-problème, elle est réalisable, mais pas nécessairement optimale, pour le problème départ. Comme elle est réalisable, elle peut être utilisée pour éliminer toute sa descendance.

CHAPITRE III

METHODES APPROCHEES

1. Introduction

Malgré l'évolution de l'informatique et des méthodes mathématiques, il y a des problèmes d'optimisation avec une taille prohibitive de l'espace de solutions admissibles. Dans de nombreuses applications réelles il est impossible de trouver la solution optimale en raison de la dynamique du système étudié, des contraintes et du nombre de variables. Compte tenu de ces difficultés, la plupart des spécialistes utilisent les techniques approchées de recherche. L'utilisation d'une méthode approchée d'optimisation ne garantit pas une solution optimale exacte mais une bonne solution dans un temps de calcul raisonnable.

Parmi les techniques approchées de recherche, les méthodes heuristiques exploitent le mieux le problème à optimiser. Elles produisent une solution non nécessairement exacte mais admissible dans un temps polynomial. De plus, les méthodes heuristiques peuvent être combinées avec d'autres méthodes d'optimisation. Dans [GUN 99b] les auteurs ont utilisé une méthode heuristique pour résoudre le problème d'équilibrage d'une ligne de désassemblage pour les ordinateurs. Une approche systémique a été utilisée pour mettre en évidence les caractéristiques uniques du processus de désassemblage. Les critères pris en considération ont été : la quantité demandée de chaque sous-ensemble, les temps opératoires de désassemblage, les directions de désassemblage et les relations de précédence entre les composants. Des fonctions de priorité ont été définies pour chaque critère.

2. Heuristique

Une heuristique est une méthode de calcul qui fournit rapidement une solution réalisable, pas nécessairement optimale ou exacte, pour un problème d'optimisation difficile. C'est un concept utilisé entre autres en optimisation combinatoire, en théorie des graphes, en théorie de la complexité des algorithmes et en intelligence artificielle.

Une heuristique s'impose quand les algorithmes de résolution exacte sont de complexité exponentielle, et dans beaucoup de problèmes difficiles. L'usage d'une heuristique est aussi pertinent pour calculer une solution approchée d'un problème ou pour accélérer le processus de résolution exacte. Généralement, une heuristique est conçue pour un problème particulier, en s'appuyant sur sa structure propre, mais les approches peuvent contenir des principes plus génériques. En général, un algorithme glouton est précisément une heuristique, par exemple :

- ✓ le rendu de monnaie par divisions successives ;
- ✓ la méthode du plus proche voisin pour le problème du voyageur de commerce.

3. Métaheuristiques

Les métaheuristiques forment un ensemble de méthodes utilisées en recherche opérationnelle pour résoudre des problèmes d'optimisation réputés difficiles. Résoudre un problème d'optimisation combinatoire, c'est trouver l'optimum d'une fonction, parmi un

nombre fini de choix, souvent très grand. Les applications concrètes sont nombreuses, que ce soit dans le domaine de la production industrielle, des transports, de l'économie, partout où se fait sentir le besoin de minimiser des fonctions numériques, dans des systèmes où interviennent simultanément un grand nombre de paramètres.

A ces problèmes de minimisation, les métaheuristiques permettent, dans des temps de calcul raisonnables, de trouver des solutions, peut-être pas toujours optimales, en tout cas très proches de l'optimum ; elles se distinguent en cela des méthodes dites exactes, qui garantissent certes la résolution d'un problème, mais au prix de temps de calcul prohibitifs pour nombres d'applications industrielles.

Une métaheuristique est un algorithme d'optimisation visant à résoudre des problèmes d'optimisation difficiles (souvent issus des domaines de la recherche opérationnelle, de l'ingénierie ou de l'intelligence artificielle) pour lesquels on ne connaît pas de méthode classique plus efficace. Les métaheuristiques sont généralement des algorithmes stochastiques itératifs, qui progressent vers un optimum global, c'est-à-dire l'extremum global d'une fonction, par échantillonnage d'une 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).

Il existe un grand nombre de métaheuristiques différentes, allant de la simple recherche locale à des algorithmes complexes de recherche globale. Ces méthodes utilisent cependant un haut niveau d'abstraction, leur permettant d'être adaptées à une large gamme de problèmes différents. Les métaheuristiques (M) sont souvent des algorithmes utilisant un échantillonnage probabiliste. Elles tentent de trouver l'optimum global (G) d'un problème d'optimisation difficile (avec des discontinuités D, par exemple), sans être piégées par les optima locaux (L) (Figure 1.11).

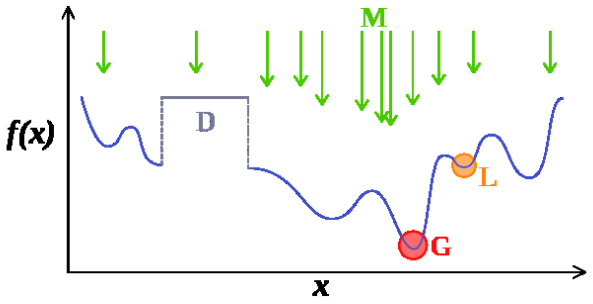


Figure 3.1. Principe général des Métaheuristiques [Wikipédia]

Méthode exacte vs méthode approchée

Une recherche exhaustive par énumération explicite de toutes les solutions pour résoudre un problème d'optimisation difficile est impensable en raison du temps de calcul induit. Dans le cas du problème du voyageur de commerce, par exemple, l'espace de recherche croit en (n-1)!, où n est le nombre de villes à visiter, ce qui dépasse rapidement les capacités de calcul de n'importe quel ordinateur. Avec seulement 50 villes, il faudra évaluer 49! trajets, soit 6.08×10^{62} trajets. C'est l'explosion combinatoire. Néanmoins, la résolution d'un tel problème d'optimisation peut se faire de manière exacte, en modélisant soigneusement le problème, puis en appliquant un algorithme ad-hoc, qui écarte d'emblée l'examen de certaines configurations, dont on sait d'ores et déjà qu'elles ne peuvent pas être optimales.
Les méthodes exactes ont permis de trouver des solutions optimales pour des problèmes de taille raisonnable. Mais malgré les progrès réalisés (notamment en matière de programmation linéaire en nombres entiers), comme le temps de calcul nécessaire pour trouver une solution risque d'augmenter exponentiellement avec la taille du problème, les méthodes exactes rencontrent généralement des difficultés avec les applications de taille importante.

Si les méthodes de résolution exactes permettent d'obtenir une ou plusieurs solutions dont l'optimalité est garantie, dans certaines situations, on peut cependant se contenter de solutions de bonne qualité, sans garantie d'optimalité, mais au profit d'un temps de calcul réduit.

Heuristique vs métaheuristique

Afin d'améliorer le comportement d'un algorithme dans son exploration de l'espace des solutions d'un problème donné, le recours à une méthode *heuristique* (du verbe grec heuriskein, qui signifie « trouver ») permet de guider le processus dans sa recherche des solutions optimales.
Dans [FEI 63] on définit une heuristique comme une règle d'estimation, une stratégie, une astuce, une simplification, ou toute autre sorte de système qui limite drastiquement la recherche des solutions dans l'espace des configurations possibles. Dans [NEW 57] on précise qu'un processus heuristique peut résoudre un problème donné, mais n'offre pas la garantie de le faire. Dans la pratique, certaines heuristiques sont connues et ciblées sur un problème particulier.
La **métaheuristique**, quant à elle, se place à un niveau plus général encore, et intervient dans toutes les situations où l'ingénieur ne connaît pas d'heuristique efficace pour résoudre un problème donné, ou lorsqu'il estime qu'il ne dispose pas du temps nécessaire pour en déterminer une. En 1996, I. H. Osman et G. Laporte définissaient la métaheuristique comme « un processus itératif qui subordonne et qui guide une heuristique, en combinant intelligemment plusieurs concepts pour explorer et exploiter tout l'espace de recherche. Des stratégies d'apprentissage sont utilisées pour structurer l'information afin de trouver efficacement des solutions optimales, ou presque-optimales ».
En 2006, le réseau Metaheuristics (metaheuristics.org) définit les métaheuristiques comme « un ensemble de concepts utilisés pour définir des méthodes heuristiques, pouvant être appliqués à une grande variété de problèmes. On peut voir la métaheuristique comme une

« boîte à outils » algorithmique, utilisable pour résoudre différents problèmes d'optimisation, et ne nécessitant que peu de modifications pour qu'elle puisse s'adapter à un problème particulier ».
Elle a donc pour objectif de pouvoir être programmée et testée rapidement sur un problème.
Tout comme l'heuristique, la métaheuristique n'offre généralement pas de garantie d'optimalité, bien qu'on ait pu démontrer la convergence de certaines d'entre elles. Etant non déterministe, elle incorpore souvent un principe stochastique pour surmonter l'explosion combinatoire. Elle fait parfois usage de l'expérience accumulée durant la recherche de l'optimum, pour mieux guider la suite du processus de recherche.

En conclusion :

- Très bons résultats sur certains types de problèmes ;
- Algorithmes faciles à mettre en œuvre ;
- Il faut faire les bons choix de paramétrage ;
- Solution non garantie ;
- Tendance : hybridation des méthodes.

4. Classification des métaheuristiques

Les métaheuristiques peuvent être classifiées selon divers critères, mais le plus souvent, elles sont catégorisées suivant le nombre de solutions générées : on distingue des métaheuristiques à solution unique et des métaheuristiques à population de solutions. Néanmoins, d'autres classifications pouvant être chevauchées pour bien cerner la méthode en question (figure 1.12).

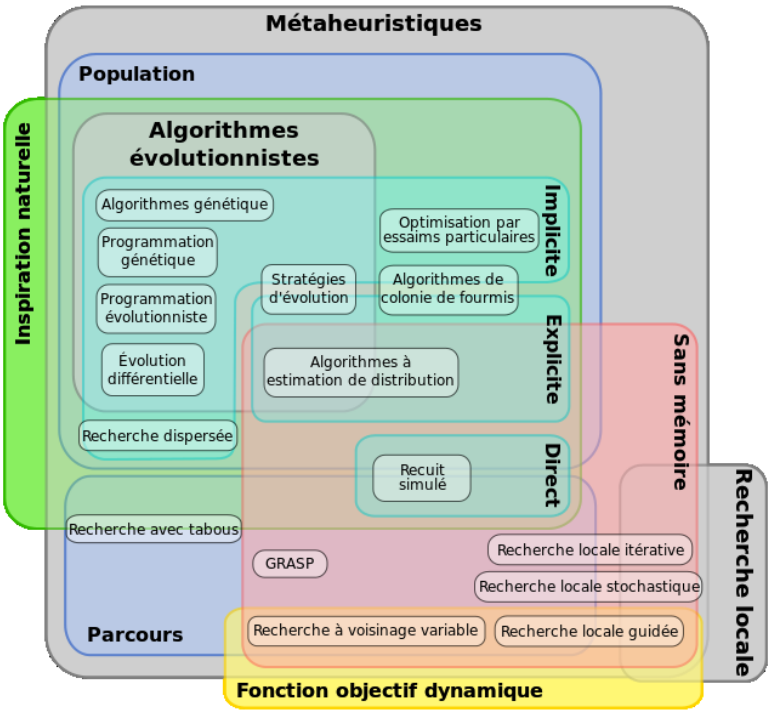


Figure 3.2. Classification des métaheuristiques [Wikipédia]

Exercices

Exercice 1

Donner la taille de l'espace de recherche de chacun des problèmes suivants puis proposer une heuristique gloutonne :

1) Problème du sac-à-dos.
2) Problème d'ordonnancement de n taches indépendantes sur une seule machine ;
3) Problème d'ordonnancement de n taches indépendantes sur deux machines ;
4) Problème de 8 reines sur un échiquier ;
5) Problème du loup, la chèvre et du chou ;
6) Problème de la chaine eulérienne ;
7) Problème de la chaine ha miltonienne.

Exercice 2

L'heuristique du plus proche voisin pour le PVC s'énonce comme suit :
On choisit un sommet (ville) arbitraire.
On part au sommet voisin le plus proche, puis de celui-là, à son plus proche voisin non visité, etc...;
Jusqu'à ce que tous les sommets aient été parcourus, où l'on revient au départ.

1) Donner un modèle puis une formulation mathématique pour le PVC.
2) Rappeler la taille de l'espace de recherche.
3) Ecrire l'algorithme de recherche exhaustive.
4) Ecrire l'algorithme de l'heuristique ci-dessus.
5) Exprimer sa complexité.

Exercice 3

Considérons les trois heuristiques suivantes pour le KSP (C ; n ; vi ; pi ; i=1,n).

H1: on commence par l'objet de plus petit poids p_i.

H2: on commence par l'objet de plus grande valeur v_i.

H3: on commence par l'objet ayant le plus grand rapport v_i/p_i.

1) Ecrire l'algorithme de chacune de ces heuristiques puis évaluer leurs complexités.
2) Comparer ces heuristiques a une méthode exhaustive puis a une méthode exacte telle que la PD.
3) Proposer votre propre heuristique puis comparer tous les resultats dans un tableau récapitulatif.
4) Tracer les courbes des temps d'exécutions et celles des résultats.
5) Que peut-on conclure ?

5. Métaheuristiques à solution unique

5.1. Recherche locale

Les techniques de recherche locale sont des méthodes approchées itératives qui explorent l'espace d'états en partant d'une solution admissible choisie à l'aide d'une heuristique. La recherche s'arrête après un nombre d'itérations ou quand la solution courante ne s'améliore plus. Le recuit simulé, la méthode Tabou, la méthode de descente sont des techniques de recherche locale qui seront présentées brièvement dans ce qui suit.

Notion de voisinage : Fonction de voisinage $N : S \rightarrow 2^S$, Indique les voisins d'une solution.

Exemple :
$S = \{0,1\}^k$, chaînes binaires de longueur k.
$s_1 \in N(s_2)$ si, et seulement si: Distance_Hamming(s_1,s_2) = 1

Algorithme général

a) Initialisation de $s \in S$;
b) Choisir s' dans N(s) et remplacer s par s' ;
c) Aller à l'étape (b) si la condition d'arrêt n'est pas vérifiée ;

Algorithme 3.1. Algorithme général de la recherche locale.

En général l'étape (b) distingue les recherches locales :

- ✓ On peut mémoriser la « meilleure » solution rencontrée ;
- ✓ Utilisation d'une évaluation incrémentale pour améliorer les temps de calcul ;

5.2. Le recuit simulé

Le recuit simulé est inspiré par l'étude de la stabilité thermique d'un système physique. La méthode part d'une solution initiale admissible et continue l'exploration de l'espace d'états en effectuant des perturbations mineures sur la solution courante. Si la nouvelle solution obtenue est améliorée alors elle est retenue. Si elle est détériorée par rapport au critère d'optimisation alors elle est retenue avec une probabilité inversement proportionnelle au nombre d'itérations (Figure 1.13). Le recuit simulé a l'avantage de couvrir un espace de recherche plus grand et d'éviter la convergence prématurée vers un optimum local. Plusieurs problèmes de job shop ont été traités par la méthode de recuit simulé.

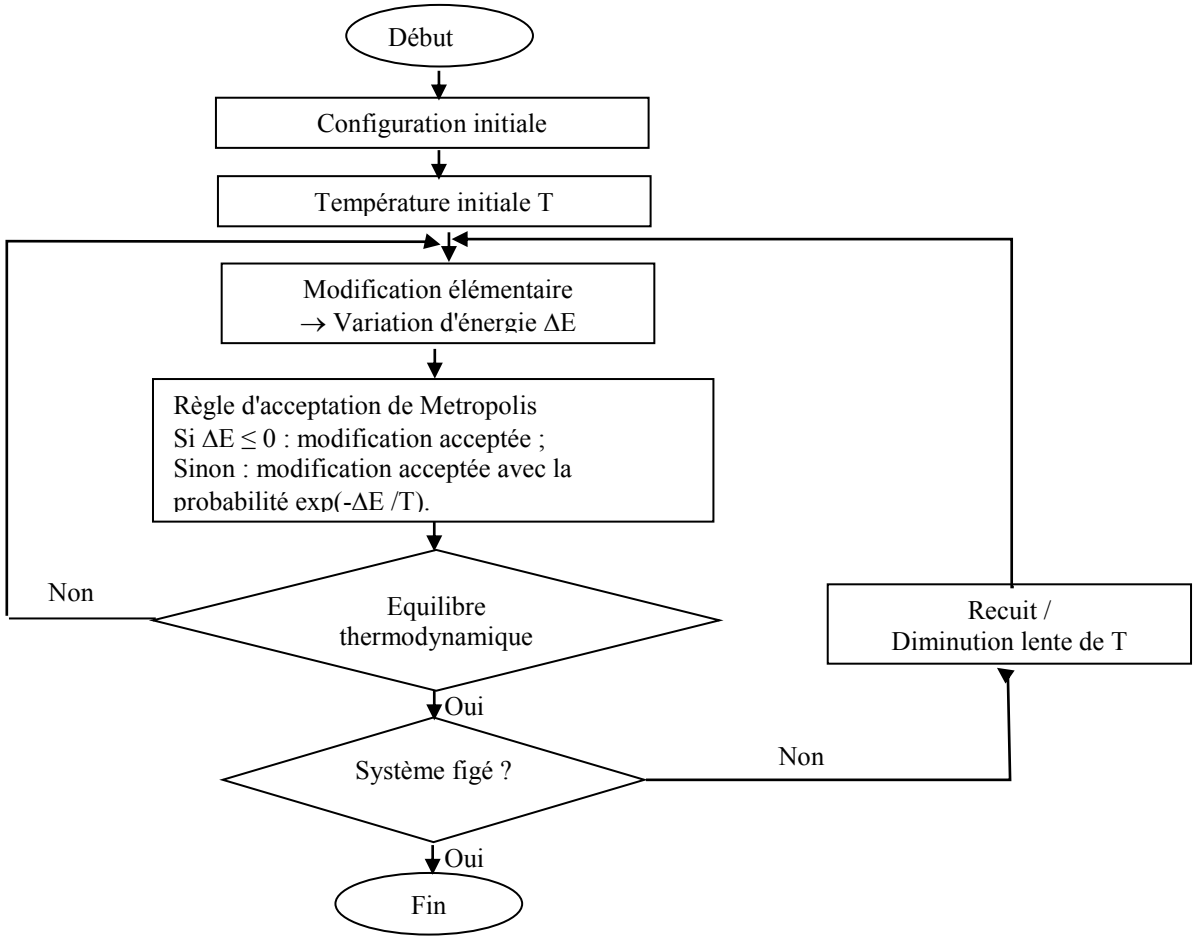


Figure 3.3. Organigramme général du recuit simulé.

5.3. La méthode tabou

La méthode Tabou est basée sur deux principes. Le premier est d'améliorer à chaque itération la valeur de la fonction objectif en croissant à chaque fois la meilleure solution voisine. Si celle-ci n'existe pas, le choix se fait sur le moins mauvais des voisins. Le deuxième principe consiste à garder en mémoire les dernières solutions choisies et de ne plus les prendre en considération. L'inconvénient de cette méthode est que si la solution atteinte est toute proche de l'optimum global, il y a possibilité de ne pouvoir pas échapper à l'optimum local atteint (Figure 1.14). La méthode Tabou a été utilisée dans l'ordonnancement des lignes flow shop ou job shop.

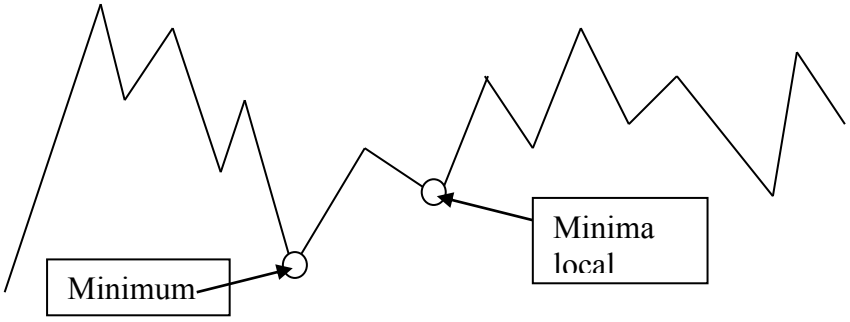


Figure 3.4. Principe de la méthode tabou.

Algorithme général

```
Début
    Choisir solution initiale s ∈ S ;
    Initialiser Tabou M ;
    répéter
        choisir s'∈ V(s) telle que :
        ( f (s') meilleure solution de V(s) ET Critère d'aspiration vérifié )
        OU f (s') meilleure solution de V(s) non taboue
        s ← s' ;
        update Tabou M ;
    jusqu'à Critère d'arrêt vérifié
fin
```

Algorithme 3.2. Algorithme général de la recherche tabou.

5.4. Méthode de la descente

Le principe de la méthode de descente (dite aussi *basic local search ou hill climbing*) consiste à partir d'une solution *s* et à choisir une solution *s'* dans un voisinage de *s*, telle que *s' améliore* la recherche (généralement telle que $f(s') < f(s)$).
On peut décider soit d'examiner *toutes* les solutions du voisinage et prendre la meilleure (ou prendre la première trouvée), soit d'examiner un sous-ensemble du voisinage (Figure 1.15).
La méthode de recherche locale la plus élémentaire est la *méthode de descente*.

Algorithme général

```
Procédure descente_simple (solution initiale s)
        Répéter
                Choisir s' dans N(s);
                Si f(s') < f(s) alors s ← s';
        Jusqu'à ce que f(s') ≥ f(s),  ∀ s' ∈ S;
Fin.
```

Algorithme 3.3. Algorithme général de la méthode de descente.

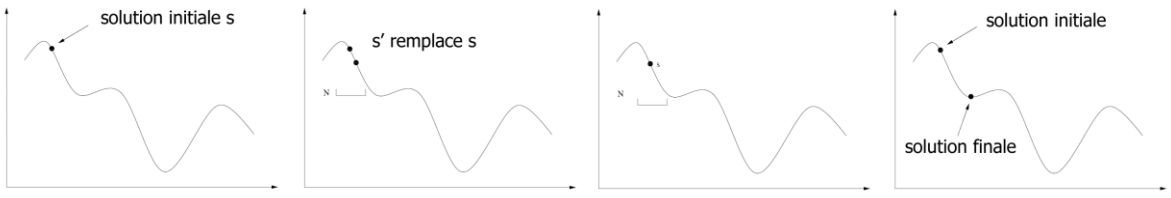


Figure 3.5. Evolution d'une solution dans la méthode de descente.

On peut varier cette méthode en choisissant à chaque fois la solution *s'* dans N(*s*) qui améliore le plus la valeur de *f*. C'est la *méthode de plus grande descente (steepest descent)*.

Exercices

Exercice 1

La variable du problème 3-SAT est un vecteur binaire de n bits. Considérons une fonction de voisinage qui associe à chaque solution x l'ensemble des solutions dont la distance de Hemming à x est 1.

1) Donner la taille de l'espace de recherche.
2) Donner le voisinage de la solution (1,0,0,1,1,0) pour n = 6.
3) Ecrire l'algorithme calculant le voisinage d'une solution x.
4) Ecrire l'algorithme d'une recherche aléatoire utilisant ce voisinage pour ce problème.

Exercice 2

Considérons, pour le PVC symétrique de n villes, une fonction de voisinage qui associe à chaque solution x l'ensemble des solutions obtenues par permutations de deux villes quelconques.

1) Donner la taille de l'espace de recherche.
2) Donner le voisinage de la solution (1,2,3,4,5,6) pour n = 6.
3) Ecrire l'algorithme calculant le voisinage d'une solution x.
4) Ecrire l'algorithme d'une recherche de recuit simulé pour ce problème.

Exercice 3

Soit le problème de programmation non convexe : Max $f(x)=x^3 - 60x^2 + 900x + 100$

$0<=x<=31$

(a) Utiliser les dérivées premières et secondes pour déterminer les points critiques (avec les solutions limites de la région réalisable) où x est un maximum ou un minimum local.
(b) Dessiner la fonction f.
(c) En utilisant x=15,5 comme solution initiale, faire la première itération du recuit simulé.
(d) En utilisant x=15,5 comme solution initiale, résoudre à l'aide de IOR Tutorial; relever, pour chaque itération, le nombre de candidats rejetés avant qu'il y en ait un sélectionné, ainsi que le nombre d'itérations où une solution qui n'améliore pas l'objectif est sélectionnée.

Exercice 4

En appliquant la méthode de recuit simulé pour certain problème, vous êtes arrivé à une itération où t = 2 et la valeur de la fonction objectif de la solution courante est 30. Cette solution a 4 voisins dont la valeur de la fonction objective donne 29, 34, 31 et 24. Pour chacune de ces solutions vous voulez déterminer la probabilité qu'elle soit choisie pour être la prochaine solution courante.

(a) Déterminer ces probabilités pour un problème de maximisation.
(b) Même question pour un problème de minimisation.

Exercice 5

Une entreprise dispose de plusieurs dépôts (Di) contenant chacun un certain nombre de containers. Différents magasins (Mj) commandent des containers. On connaît le coût de transport de chaque dépôt aux magasins.
Exemple :

	M1	M2	M3	DEPOTS
D1	5	3	4	8
D2	6	7	2	9

Les demandes magasins sont 4, 5 et 8 containers.
Quelle est l'organisation des livraisons des containers pour minimiser le coût total de transport ?
Ce problème peut se modéliser par programmation linéaire en nombres entiers et sa résolution peut se faire par séparation et évaluation (Chercher une solution).
Proposez une solution pour le problème en se basant sur la méthode de recherche taboue.

Exercice 6

Soit le problème de coloration des arrêtes d'un graphe G(X, E) avec X l'ensemble des sommets et E l'ensemble des arrêtes. L'objectif est de colorer les arrêtes du graphe tel que deux arêtes ayant une extrémité commune soient de couleurs différentes. Soit ci la couleur de l'arrête i. Le but de l'exercice est d'appliquer la méthode de recherche taboue pour le problème.
Donnez la représentation d'une solution du problème, montrer à l'aide d'un graphe cette représentation.
Considérons trois couleurs c1, c2 et c3 et selon votre représentation, donnez une solution s initiale. Puis déroulez l'algorithme tabou pour deux itérations.
Pour les deux itérations, donnez-le contenu de la liste taboue.

Exercice 7

1) Ecrire un algorithme d'affectation de fréquences utilisant la méthode du recuit simulé.
2) Pour cela définir la fonction énergie qui doit être minimisée.
3) Définir une transformation élémentaire dans l'espace des configurations.
4) Indiquer dans quelle(s) condition(s) une transformation élémentaire peut être acceptée.
5) Donner une méthode permettant de construire la configuration initiale sur laquelle le recuit simulé est appliqué.

A quel moment en théorie, dans le déroulement de l'algorithme, la température devrait-elle être réduite ?

6. Métaheuristiques à population de solutions

6.1. Colonies de fourmis

L'algorithme de colonie de fourmis est une métaheuristique basée sur le comportement collectif des fourmis. Un parallèle a été établi entre la recherche de nourriture et la résolution de problèmes complexes. En effet, pour rechercher la nourriture, les fourmis se dispersent

(elles n'ont qu'une vision locale de l'environnement) et après l'avoir trouvée, elles retournent au nid en laissant des phéromones pour marquer leur passage. Les phéromones sont des substances volatiles que les fourmis utilisent pour communiquer. Les fourmis ainsi attirées répandront à leur tour des phéromones et le chemin entre le nid et la zone de nourriture sera renforcé.
Cependant, grâce au phénomène d'évaporation des pistes de phéromones (si le trajet n'est pas ou peu emprunté, les phéromones tendent à disparaître), elles aboutissent assez vite à un trajet très marqué, représentant le plus court chemin, comparé à d'autres directions qui disparaissent (Figure 1.16).

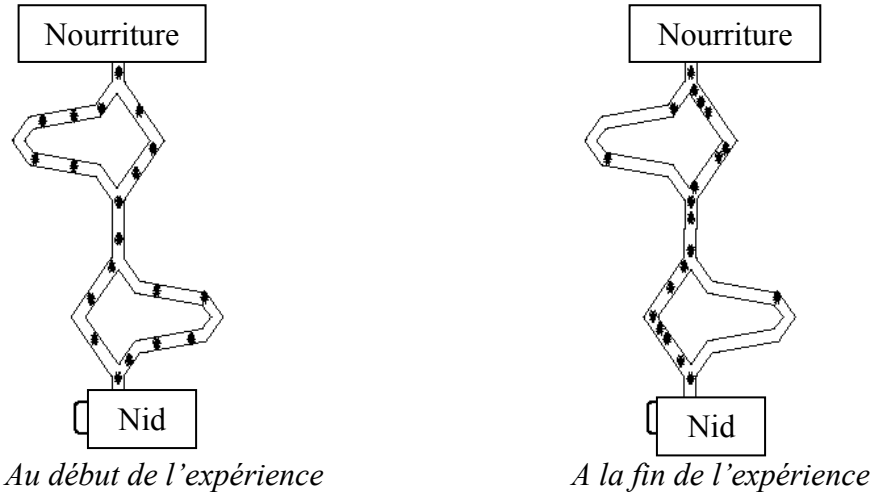


Figure 3.6. Expérience de sélection de branches par une colonie de fourmis.

Le premier algorithme conçu selon ce modèle était destiné à résoudre le problème du voyageur de commerce, et c'est sous cette forme que nous le présentons dans les lignes suivantes. Nous dirons ensuite quelques mots de la généralisation à laquelle cet algorithme a donné lieu. Le principe consiste à « lancer » des fourmis, et à les laisser élaborer pas à pas la solution, en allant d'une ville à l'autre. C'est donc un algorithme qui repose sur la *construction* progressive de solutions, un peu comme dans la méthode GRASP, qui inclue également une phase de construction. Afin de ne pas revenir sur ses pas, une fourmi tient à jour une liste Tabou, qui contient la liste des villes déjà visitées. Soit A un ensemble de *k* fourmis :

```
Répéter
        Pour i=1 à k faire
                ConstruireTrajet(i);
        Fin Pour
        MettreàJourPheromones();
Jusqu'à ce que le critère de terminaison soit satisfait.
```

Algorithme 3.4. Algorithme général de la méthode de colonie de fourmis.

Dans la procédure *ConstruireTrajet*(i), chaque fourmi se construit une route en choisissant les villes selon une règle de transition aléatoire très particulière :

Si $p_{ij}^k(t)$ est la probabilité qu'à l'itération *t* la fourmi *k* choisisse d'aller de la ville *i* à la ville *j*, alors on a :

$$p_{ij}^k(t) = \frac{(\tau_{ij}(t))^\alpha \cdot (\eta_{ij})^\beta}{\sum_{l \in J_i^k} (\tau_{il}(t))^\alpha \cdot (\eta_{il})^\beta} \quad \text{si } j \in J_i^k$$

$$p_{ij}^k(t) = 0 \text{ Sinon}$$

Où : $\tau_{ij}(t)$ désigne le taux de phéromone sur la route *ij* à l'itération *t ;*

η_{ij} Désigne l'inverse de la distance séparant les villes *i* et *j* ;

α et β sont deux paramètres contrôlant respectivement l'influence du taux de phéromone sur le trajet *ij*, et l'influence de la distance sur le trajet *ij*.

En d'autres termes, plus il y a de phéromone sur le trajet reliant deux villes, plus la probabilité est grande que la fourmi emprunte ce trajet. Mais ceci est contrebalancé par la longueur du trajet. α et β permettent de régler l'effet de ces paramètres. Lorsque toutes les fourmis ont construit une solution, la procédure *MettreàJourPheromones*() modifie les taux de phéromone τ sur les routes en fonction des trajets effectivement empruntés par les fourmis, selon la formule :

$$\Delta\tau_{ij}^k(t) = \frac{Q}{L^k(t)} \text{ si le trajet (i,j) est dans la tournée de la fourmi } k.$$

Q est une constante, et $L^k(t)$ est la longueur totale de la tournée de la fourmi *k*. On constate donc que plus la route suivie par la fourmi a été courte, plus grande est donc la quantité de phéromone laissée derrière elle. Pour éviter que des chemins ne se forment trop vite, et ne convergent trop rapidement vers des optima locaux, on introduit le concept d'évaporation des pistes de phéromone, au travers du paramètre ρ ($0 < \rho \leq 1$) dans la formule complète de mise à jour suivante :

$\tau_{ij}(t+1) = (1-\rho) \cdot \tau_{ij}(t) + \Delta\tau_{ij}(t)$ avec $\Delta\tau_{ij}(t) = \sum_{x=1}^{k} \Delta\tau_{ij}^x(t)$, et *k* le nombre de fourmis.

Il est également possible d'introduire des modifications des taux de phéromone par un processus démon, indépendamment des itérations des fourmis. Ceci permet de biaiser l'évolution du système, afin d'introduire le résultat de certaines recherches locales, ou pour éviter des convergences prématurées. Comme dans beaucoup de métaheuristiques, un équilibre doit être trouvé entre l'intensification (qui conforte une solution en particulier) et la diversification (qui explore l'espace d'états). Une intensification d'un chemin trop rapide (en augmentant fortement le dépôt de phéromones) peut conduire à un maximum local tandis qu'une diversification trop prononcée (les fourmis choisissent des villes au hasard) n'aboutit pas à une solution satisfaisante en des temps corrects.

Exercices

Exercice 1

1) Réécrire l'algorithme d'optimisation par colonie de fourmis pour résoudre le problème du voyageur de commerce.
2) Exprimer la complexité de cet algorithme.
3) Dérouler les deux premières itérations pour le PVC de 4 villes A, B, C, D tels que :
   AB=6 AC=5 AD=8 BC=4 BD=4 CD=5
   Pour chacun des cas suivants :
   a) Q=20, α=1, β=1 , τ0=1 ;
   b) Q=20, α=1, β=5 , τ0=1 ;
   c) Q=20, α=5, β=1 , τ0=1 ;

   Prendre ρ = 0.01 , ρ = 0.01 puis ρ = 0.1.
   Que peut-on conclure ?

4) Modifier l'algorithme obtenu en (1) pour :
   a) La version élitiste ;
   b) La version min-max ;
   c) La version 2-opt ;

Exercice 2

1) Ecrire un algorithme d'optimisation par colonie de fourmis pour résoudre le problème du sac-a-dos simple.
2) Généraliser cet algorithme au cas multidimensionnel.

Exercice 3

Ecrire un algorithme d'optimisation par colonie de fourmis pour résoudre :
a) le problème sat ;
b) Le problème des 8 reines ;
c) Le problème d'affectation ;

6.2. Algorithmes génétiques

Les algorithmes génétiques (AGs) sont des algorithmes fondés sur les mécanismes de la sélection naturelle et de la génétique, utilisant les principes de la survie des structures les mieux adaptées. Ces algorithmes fabriquent des chromosomes qui codent chacun une solution potentielle à un problème donné à chaque étape (appelée génération), ces chromosomes se combinent, se mutent et sont sélectionnés en fonction de leurs qualités à répondre au problème afin de les explorer dans la génération suivante avec l'espoir d'améliorer la performance qui en résulterait.

Le principal apport de ce type d'algorithmes probabilistes et heuristiques est la robustesse. En effet, un tel algorithme est capable de résoudre des problèmes d'optimisation combinatoire difficiles (NP-complets) en effectuant une recherche efficace et indépendante de la taille du problème traité, dans un espace contenant un ensemble de solutions potentielles, chose qui dépassait la puissance des algorithmes déterministes limités aux petits espaces de recherche.

6.2.1. Historique

Depuis son apparition au 19° siècle, la fameuse théorie d'évolution des espèces de Darwin n'eut cessé d'étaler des polémiques non seulement au sein des onologues mais aussi aux chercheurs de divers domaines touchés par cette théorie. En outre, l'avènement de métaheuristiques inspirées des phénomènes naturels pour pallier les lacunes des méthodes exactes, poussèrent certains spécialistes à tirer profit de cette théorie en le projetant dans le domaine de l'optimisation combinatoire. Pour récapituler, voici les jalons les plus marquants de l'historique des algorithmes génétiques que nous avons pu collecter :

1860 : Charles Darwin publia son célèbre livre intitulé : *L'origine des espèces au moyen de la sélection naturelle ou la lutte pour l'existence dans la nature*. Dans ce livre, Darwin rejette l'existence «de systèmes naturels figés».

1966 : Programmation évolutionnaire par L. J. Fogel.

1973 : Stratégie d'évolution de I. Rechenberg.

1975 : Dans les années 1960, John Holland étudia les systèmes évolutifs et, en 1975, il introduit le premier modèle formel des algorithmes génétiques (*the canonical genetic algorithm AGC*) dans son livre : *Adaptation in Natural and Artificial Systems*. Ce modèle servira par la suite de base aux recherches ultérieures.

1989 : David Goldberg publie un ouvrage de vulgarisation des algorithmes génétiques.

Années 1990 : Programmation d'une panoplie d'algorithmes génétiques transcris en C++, appelée GAlib. Cette librairie contenait des outils pour des problèmes d'optimisation en utilisant les AG. Elle est conçue pour servir de support de programmation

6.2.2. Objectifs

Les algorithmes génétiques appartiennent à la famille des algorithmes évolutionnaires. Leur but est d'obtenir une solution approchée à un problème d'optimisation, lorsqu'il n'existe pas de méthode exacte (ou que la solution est inconnue) pour le résoudre en un temps raisonnable. Ceci consiste à déterminer les extrêmes d'une fonction :

$f: X \longrightarrow R$, où X est un ensemble quelconque appelé espace de recherche et f est appelée fonction d'adaptation ou fonction d'évaluation ou encore fonction *fitness*.

- Les algorithmes génétiques représentent une méthode approchée polynomiale et vinrent pallier le handicap exponentiel des méthodes exactes.
- Par leur caractère générique et formel, ils eurent pu être non seulement une méthode approchée pour un certain type de problème mais une métaheuristique qui pourrait être projetée à tout type de problème
- Inspirés d'un phénomène naturel, ils vinrent concurrencer les métaheuristiques existantes en termes de qualité de solution et temps de calcul.

6.2.3. Domaines d'application

Les champs d'application des AG sont très vastes. Outre l'économie, ils sont utilisés pour l'optimisation de fonctions (DE Jong 1980), en programmation génétique (Koza (1992)), pour le contrôle de pipelines (Goldberg (1981)), en théorie du contrôle optimal (Krishnakumar et Goldberg (1992), Michalewicz, Janikow et Krawczyk (1992) et Marco *et al.* (1996) et plus récemment Jamshidi et al. (2003), ou encore en théorie des jeux répétés (Axelrod (1987)) et dynamiques (Özyildirim (1996, 1997) et Özyildirim et Alemdar (1998)). [THO03]

Les algorithmes génétiques sont également utilisés dans l'informatique décisionnelle, ils sont mis en œuvre dans certains outils de data mining pour rechercher une solution optimum à un problème par *mutation* des attributs (des variables) de la population étudiée. Ils sont utilisés par exemple dans une étude d'optimisation d'un réseau de points de vente ou d'agences (banque, assurance, …) pour tenter de répondre aux questions :

- Quelles sont les variables (superficie, effectif, …) qui expliquent la réussite commerciale de telle ou telle agence ?
- En modifiant telle variable (*mutation*) de telle agence, améliore-t-on son résultat ?

Dans l'application industrielle**,** un premier exemple est une réalisation effectuée au sein de l'entreprise Motorola. Le problème pour lequel Motorola a utilisé les algorithmes génétiques concerne les tests des applications informatiques. En effet, lors de chaque changement apporté à une application, il convient de tester l'application afin de voir si les modifications apportées n'ont pas eu d'influence négative sur le reste de l'application. [THO03]

Les raisons de ce grand nombre d'applications sont la simplicité et l'efficacité de ces algorithmes.

Lerman et Ngouenet (1995) distinguent quatre principaux points qui font la différence fondamentale entre ces algorithmes et les autres méthodes : [GOL89a]

a) Les algorithmes génétiques utilisent un codage des paramètres, et non les paramètres eux-mêmes.
b) Les algorithmes génétiques travaillent sur une population de points, au lieu d'un point unique. cela introduit donc du parallélisme.
c) Les algorithmes génétiques n'utilisent que les valeurs de la fonction étudiée, pas sa dérivée, ou une autre connaissance auxiliaire.
d) Les algorithmes génétiques utilisent des règles de transition probabilistes, et non pas déterministes, ce qui permet de les qualifier de stochastiques.

6.2.4. Fonctionnement

Les algorithmes génétiques (**AG**s) sont des algorithmes d'optimisation stochastique fondés sur les mécanismes de la sélection naturelle et de la génétique. Leur fonctionnement est extrêmement simple. On part avec une population de solutions potentielles (*chromosomes*) initiales arbitrairement choisies. On évalue leur performance (*fitness*) relative. Sur la base de ces performances on crée une nouvelle population de solutions potentielles en utilisant des opérateurs évolutionnaires simples : la sélection, le

croisement et la mutation (figure 2.4.). On recommence ce cycle jusqu'à ce que l'on trouve une solution satisfaisante.

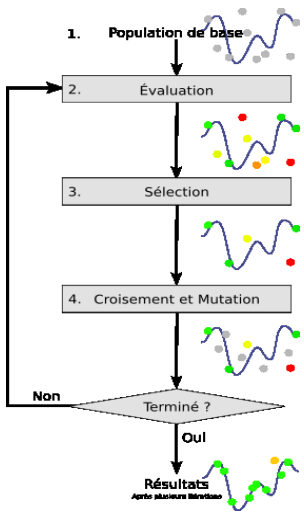


Figure 3.7. Principe général de fonctionnement des AGs.
(Source : Wikipédia)

Cet algorithme nécessite, bien entendu, un bon choix de codage, paramètres, structures de données adéquats avant son implémentation pour aboutir à de bons résultats. Nous revenons sur ces éléments plus loin dans ce chapitre.

Si les choses vont dans un bon chemin, pendant l'exécution d'un AG, on devrait voir la fonction objective évoluer (décroitre ou croitre selon le problème en question) en fonction des générations (figure 2.5.).

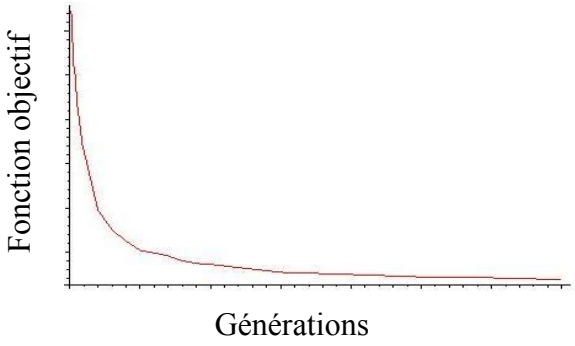


Figure 3.8. Évolution de l'objectif avec les générations dans un AG.

6.2.5. Terminologie

Comme toute autre métaheuristique, les AGs possèdent leurs propres termes qui représentent en quelques sortes le "langage parlé" pour pouvoir y projeter les vocables du problème en question. Nous citerons ci-après un ensemble non exhaustif de cette terminologie.

6.2.6. Individu

Les individus correspondent aux « solutions » du problème à optimiser. Ces solutions doivent être « codées » pour que le traitement puisse être effectué par l'algorithme génétique. Cette représentation codée d'une solution est appelée chromosome, et est composée de gènes. Chaque gène peut représenter une variable, un élément de la solution, ou encore une partie plus abstraite. La manière la plus utilisée de codage par algorithme génétique est le codage en vecteurs. Chaque solution est représentée par un vecteur. Ce vecteur peut être binaire ou encore de n'importe quel type discret dénombrable (entier, caractères, etc.). On pourrait également utiliser un type continu (ex : nombres réels), mais dans ce cas, il faut également revoir les opérations qui modifient le contenu des chromosomes (la fonction qui crée aléatoirement les chromosomes et les opérateurs génétiques). La simplicité veut que les chromosomes soient uniformes, c'est-à-dire que tous les gènes sont du même type. Cependant, si on tient compte encore une fois des opérations qui modifient le contenu des chromosomes, on peut assez aisément construire des vecteurs d'éléments de type différents. On demande habituellement que les chromosomes soient tous de même longueur, basés sur la même architecture, les gènes homologues étant au même endroit sur leur chromosome respectif. De fait, le codage par vecteur est si utilisé (très grande simplicité comparée aux autres méthodes de codage de chromosome) que les algorithmes sont souvent identifiés comme étant des méthodes de traitement vectoriel. Cette affirmation n'est pas tout a fait vraie car d'autres types de codage existent, bien que n'étant pas très fréquents. Par exemple, l'utilisation des algorithmes génétiques pour faire de la programmation génétique utilise un codage en arbre (ce qui permet entre autres d'avoir des chromosomes de longueurs différentes).

6.2.7. Population

C'est l'ensemble des individus, ou encore l'ensemble des chromosomes d'une même génération. Habituellement, la taille de la population reste constante tout au long de l'algorithme génétique. Cependant, ce caractère contredit la théorie d'évolution elle-même qui énonce explicitement que toutes les populations des espèces évoluent, c a d se reproduisent, se multiplient en respectant la règle de base : les individus les plus adaptés survivent, les moins adaptés a leur environnement disparaissent au fil des générations. SINON, le nombre d'habitants de notre planète 1000 ans J.-C. était près de 6 milliards !

6.2.8. Population initiale

Habituellement, au départ d'un algorithme génétique, il faut créer une population d'individus. Ces individus sont générés par une fonction simple. Cette fonction affecte à chaque individu qu'elle génère une valeur aléatoire pour chacun de ses gènes. L'algorithme génétique peut également utiliser comme population de départ une population déjà créée a priori qui peut être le résultat d'une autre stratégie, la solution serait dans ce cas certes meilleure puisqu'on part d'une solution approchée qui substitue une solution aléatoire, on parle alors d'hybridation de méthodes.

6.2.9. Fitness d'un individu

Le calcul de la qualité d'un individu est essentiel aux algorithmes génétiques. Cette fonction donne, en valeur numérique (habituellement réelle), la qualité d'un individu. C'est selon cette valeur numérique que sont calculées les chances de sélection de cet individu. La fonction de *fitness* doit avoir 0 comme plancher, pour ne pas fausser le calcul des pourcentages. Les algorithmes génétiques étant une technique d'optimisation, ils cherchent la qualité maximale, donc l'optimisation de la fonction de qualité. Si on cherche plutôt à minimiser une fonction, il faudra la modifier de sorte que la fonction de qualité se maximise. Il serait bien entendu possible de conserver une fonction de qualité qui fonctionne à l'envers et de modifie à la place le calcul des probabilités, mais ceci rendrait l'algorithme beaucoup plus difficile à décoder pour les utilisateurs externes.

6.3. Codage

Le codage est un processus très important des algorithmes génétiques, il consiste en une représentation formelle des individus pour faciliter l'implémentation d'un AG en adoptant un description adéquate aux opérateurs génétiques et à la fonction fitness. On a prouvé empiriquement que le codage jour un rôle primordial dans la vitesse et l'efficacité des AGs plusieurs codages sont employés. Voici quelques exemples **:**

6.3.1. Codage binaire

Le gène est codé par un caractère binaire, 0 ou 1. C'est le plus courant et celui qui a été employé lors de la première application des algorithmes génétiques (Figure 2.6). d'autres codages sont également possibles tels que les entiers et ça reste toujours fonction du problème à traiter.

Exemple

Chromosome A

0	1	0	0	1	1	0	1

Chromosome B

1	1	1	0	0	1	0	1

Figure 3.9. Schéma du codage binaire.

6.4. Fonction d'évaluation

Pour calculer le coût d'un point de l'espace de recherche, on utilise une fonction d'évaluation. L'évaluation d'un individu ne dépendant pas de celle des autres individus, le résultat fournit par la fonction d'évaluation va permettre de sélectionner ou de refuser un individu pour ne garder que les individus ayant le meilleur coût en fonction de la population courante : c'est le rôle de la fonction fitness. Cette méthode permet de s'assurer que les individus performants seront conservés, alors que les individus peu adaptés seront progressivement éliminés de la population. **[GOL89b]**

Cependant, en général, on ne fait pas la différence entre fonction fitness et fonction d'évaluation. Cette fonction, propre au problème, est souvent simple à formuler lorsqu'il y a peu de paramètres. Au contraire, lorsqu'il y a beaucoup de paramètres ou lorsqu'ils sont corrélés, elle est plus difficile à définir.

6.5. Opérateurs de reproduction

Les algorithmes génétiques sont basés sur un phénomène naturel : l'***évolution***. Plus précisément, ils supposent, qu'a priori, deux individus adaptés à leur milieu donnent, par recombinaison de leurs gènes, des individus mieux adaptés. Pour ce faire, trois opérateurs sont à disposition : la ***sélection***, le ***croisement*** et la ***mutation***, plus un opérateur optionnel, l'***élitisme***. [GOL91]

6.5.1. Sélection

La sélection sert à choisir dans l'ensemble de la population les individus qui participeront à la reproduction. Plusieurs méthodes existent et sont, généralement, basées sur la théorie de Darwin. Ainsi les meilleurs individus ont plus de chance de survivre et de se reproduire.

Roulette

Cette méthode exploite la métaphore d'une roulette de casino. La roue est divisée en autant de secteurs que d'individus dans la population. La taille de ces secteurs est proportionnelle à l'adaptation de chaque individu. En faisant tourner la roue, l'individu pointé à l'arrêt de la boule est sélectionné. Les individus les mieux adaptés ont donc plus de chance d'être tirés au sort lors du déroulement du jeu figure 2.7.).

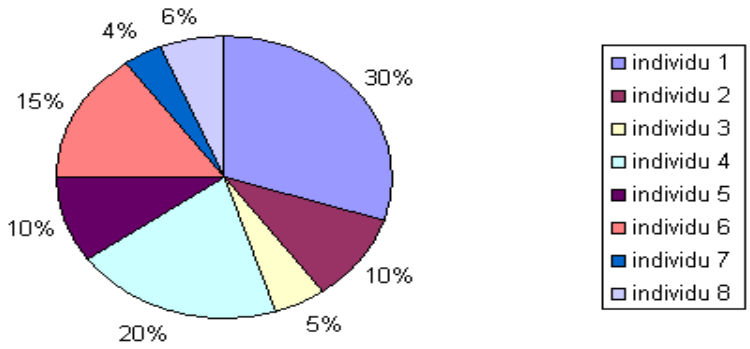


Figure 3.10. Schéma de la sélection par la roulette.

D'autres procédures de sélection sont également réalisables, on en cite la sélection par rang et la sélection par tournoi [GOL 89b].

Elitisme

A la création d'une nouvelle population, il y a de grandes chances que les meilleurs chromosomes soient perdus après les opérations d'hybridation et de mutation. Pour éviter cela, on utilise la méthode d'élitisme. 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.

6.5.2. Le croisement

Le croisement combine les gènes des deux individus parents pour donner deux nouveaux chromosomes d'individus enfants. La zone de croisement est généralement choisie aléatoirement dans les chromosomes. Les méthodes de croisement sont liées au codage. Le croisement de chromosomes codé en binaire ne sera pas le même que celui d'un chromosome codé par valeur entière mais leur principe est identique.

Le croisement 1 point avec codage binaire

Dans le chromosome, un point de croisement est choisi. La première partie du chromosome de l'individu dit parent 1 est alors copiée sur un individu de la prochaine génération, enfant 1, celle du parent 2 est copie sur un enfant 2. Pour la deuxième partie du chromosome, les parents échangent leurs enfants, ainsi le parent 1 est copié sur l'enfant 2 et le parent 2 sur l'enfant 1 (figure 2.8.).

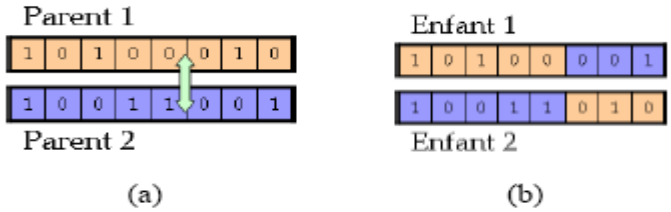


Figure 3.11. Schéma de croisement 1 point avec le codage binaire.

Croisement deux points

On choisit au hasard deux points de croisement. Par la suite, nous avons utilisé cet opérateur car il est généralement considéré comme plus efficace que le précédent [Beasley, 1993b]. Néanmoins nous n'avons pas constaté de différence notable dans la convergence de l'algorithme (figure 2.9).

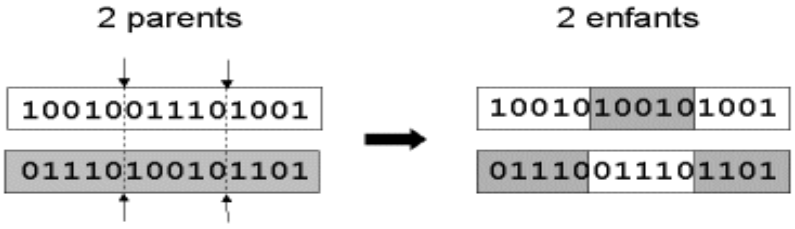


Figure 3.12. Schéma de croisement en 2 points.

Notons que d'autres formes de croisement existent, du croisement en *k* points jusqu'au cas limite du croisement uniforme.

Il est à noter également qu'un mono croisement est également possible. Ce dernier consiste à produire un seul enfant d'un seul parent. Ce type de croisement présente l'avantage de ne pas générer de gênées létaux qui conduisent souvent à priver le croisement de réaliser la diversification dans la recherche et qui a bien évidemment un cout en termes de temps de calcul. Nous avons appliqué ce type de croisement au PVC dans [HEM 15].

6.5.3. La mutation

Comme les individus les mieux adaptés sont les plus susceptibles d'être choisis lors de la sélection, la perte de certains gènes est inévitable avec le temps. La mutation est l'opérateur qui permet d'éviter la dégénérescence de la population et d'enrichir le pool de gènes. Cette dégénérescence peut se traduire par une convergence des individus vers un optimum local, d'où l'importance de la mutation.

Classiquement, la mutation modifie aléatoirement (Inversion d'un bit au hasard ou remplacement au hasard d'un caractère par un autre), un petit nombre de gènes, avec un faible *taux de probabilité* (n'est pas élevé à 5%). Bien entendu, comme pour le croisement, la mutation dépend du problème.

Mutation en codage binaire
Un ou plusieurs gènes, selon la taille du chromosome, et un taux gènes mutés gènes totales très faible sont choisis aléatoirement. Ils sont alors inversés (1→0 et 0→1) (figure 2.10.).

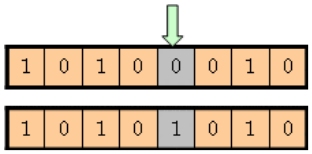


Figure 3.13. Schéma de mutation avec le codage binaire.

6.5.4. Réinsertion

Comment réinsérer le fils dans le groupe ? Les solutions sont :

- Eliminer le moins bon du groupe
- Eliminer l'individu qui ressemble le plus au nouvel individu
- Remplacer un des deux parents
- La nouvelle génération remplace l'ancienne génération (sauf le meilleur individu de l'ancienne génération)

Ce qui est certain est qu'il ne faut pas supprimer le meilleur individu du groupe, sous peine de voir l'adaptation globale diminuer !

6.6. Principaux paramètres des AG

Les opérateurs de l'algorithme génétique sont guidés par un certain nombre de paramètres structurels donnés. La valeur de ces paramètres influence la réussite ou non et la rapidité d'un

algorithme génétique. Nous allons maintenant discuter rapidement le rôle de ces paramètres dans la version simple que nous avons retenue des AG. **[**THO 03**]**

6.6.1. Taille de la population

Si la taille de la population est trop grande le temps de calcul de l'algorithme peut s'avérer très important, en revanche, si elle est trop petite, il peut converger trop rapidement vers un mauvais chromosome. Cette importance de la taille est essentiellement due à la notion de *parallélisme implicite* qui implique que le nombre d'individus traité par l'algorithme est au moins proportionnel au cube du nombre d'individus.

6.6.2. Probabilité de croisement

Elle dépend de la forme de la fonction de fitness. Son choix est en général heuristique. Plus elle est élevée, plus la population subit de changements importants. Les valeurs généralement admises sont comprises entre 60% et 95%.

6.6.3. Probabilité de mutation

Ce taux est généralement faible puisqu'un taux élevé risque de conduire à une solution sous-optimale en perturbant celle qui est optimale.

Plutôt que de réduire p_m, une autre façon d'éviter que les meilleurs individus soient altérés est d'utiliser l'élitisme : Ainsi, peut-on choisir, par exemple, de reproduire à l'identique les 5% meilleurs de la population à chaque génération, l'opérateur de reproduction ne jouant alors que sur les 95% restant.

6.6.4. Critère d'arrêt

C'est la condition qui détermine la fin de l'évolution de la population. Plusieurs modes de ce critère peuvent être adoptés selon les résultats empiriques et les contraintes du problème utilisé, parmi lesquels nous trouvons :

- Arrêt après un grand nombre d'itérations proportionnel à la taille du problème.
- Arrêt après consommation d'un temps estimé proportionnel à la taille du problème.
- arrêt s'il y aurait pas amélioration de la qualité pendant un certain nombre d'itérations qui doit encore être dépendant de la taille du problème.
- arrêt si une certaine valeur seuil prédéfinie de la solution recherché est atteinte.
- arrêt si la différence entre les deux solutions les plus récentes dépasserait un certain seuil prédéfini.

6.7. Exemple : Application au problème du voyageur de commerce

Nous allons maintenant nous intéresser à une application plus concrète : le problème du voyageur de commerce. Cet exemple est un classique appartenant à la classe des problèmes NP-complets.

Il consiste à visiter un nombre N de villes en un minimum de distance sans passer deux fois par la même ville. Il s'agit donc d'optimiser le coût d'un parcours dans un graphe complet possédant un certain nombre de sommets, en passant une et une seule fois par

chacun. Des méthodes déterministes existent déjà pour résoudre le problème, mais le temps de calcul est très long : elles reviennent à parcourir toutes les solutions possibles et à déterminer la moins coûteuse.
Le but sera ici de montrer comment modéliser le problème à partir d'algorithmes génétiques et des divers opérateurs que nous avons à disposition, qui ont été définis antérieurement.
Ceci nous conduira ensuite à montrer les avantages de cet outil par rapport à une résolution de type déterministe.

Représentation du problème
Le problème du voyageur de commerce peut se modéliser à l'aide d'un graphe complet de *n* sommets dont les arrêtes sont pondérées par un coût strictement négatif. Pour l'implantation de notre algorithme, l'instance sera modélisée comme distance euclidienne sur *n* points du plan, pour construire la matrice de coût.

L'espace de recherche
L'espace de recherche est l'ensemble des permutations de **{1, 2, ..., n }**.
Un point de cet espace de recherche est représenté par une de ces permutations.

Codage des points de l'espace de recherche
Une première idée serait de coder chaque permutation (i.e : chaque point de l'espace de recherche) par une chaîne de bits.
Par exemple, pour *n* = 10 : 0011 0111 0000 0100 0001 0010 0101 0110 représente la permutation : 3 7 0 4 1 2 5 6
On s'aperçoit bien que chaque élément de l'ensemble de permutation est codé sur :
i = E (ln (*n*) / ln (2)) bits , E étant la fonction de partie entière.
Sachant qu'une permutation est de taille *n,* la chaine de bits sera alors de taille *n* * i.

Représentation d'une solution
Comme nous l'avons déjà dit, le voyageur de commerce doit revenir à son point de départ et passer par toutes les villes une fois et une seule. Nous avons donc codé une solution par une structure de données comptant autant d'éléments qu'il y a de villes, c'est à dire une permutation. Chaque ville y apparait une et une seule fois. Il est alors évident que selon la ville de départ que l'on choisit, on peut avoir plusieurs représentations différentes du même parcours (figure 2.12.).

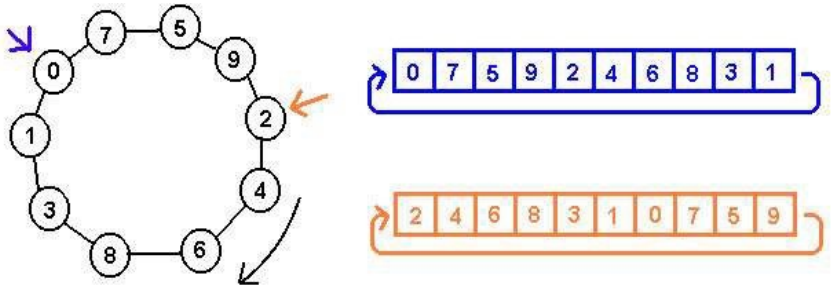


Figure 3.14. Codage d'une solution par un tableau.

Sélection

Nous utilisons ici la méthode de sélection par roulette. On calcule d'abord la valeur moyenne de la fonction d'évaluation dans la population :

$$\overline{f} = \frac{1}{m}\sum_{i=0}^{m-1} f(P_i),$$

Où P_i est l'individu i de la population et la taille de la population. La place d'un individu P_i dans la roulette est proportionnel à $\frac{\overline{f}}{f(P_i)}$. On sélectionne alors $m/2$ individus pour la reproduction. Il y a aussi la possibilité d'avoir une politique d'«élitisme». C'est à dire qu'à chaque étape de sélection le meilleur chromosome est automatiquement sélectionné (cf. partie 2).

Croisement

Etant donné deux parcours il faut combiner ces deux parcours pour en construire deux autres. Nous pouvons suivre la méthode suivante (figure 2.13.) :

1. On choisit aléatoirement deux points de découpe.
2. On interverti, entre les deux parcours, les parties qui se trouvent entre ces deux points.
3. On supprime, à l'extérieur des points de coupe, les villes qui sont déjà placées entre les points de coupe.
4. On recense les villes qui n'apparaissent pas dans chacun des deux parcours.
5. On remplit aléatoirement les trous dans chaque parcours.

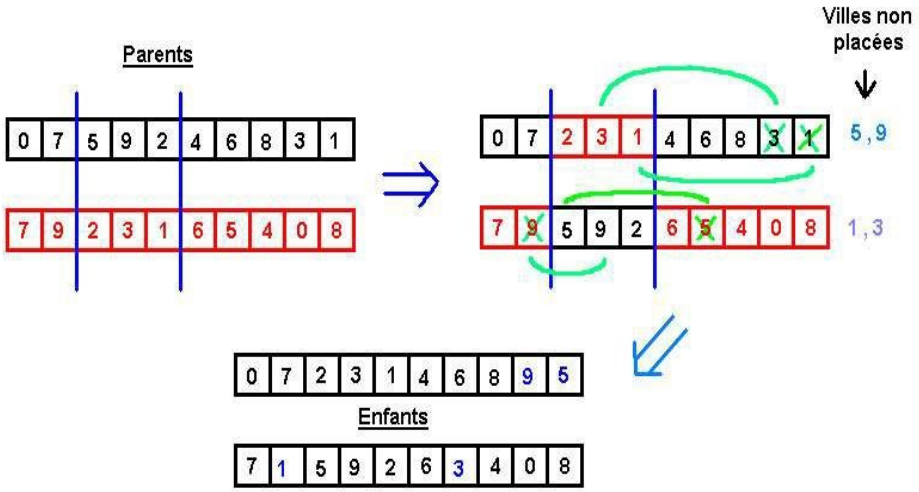


Figure 3.15. Exemple de croisement.

Mutation

Il s'agit ici de modifier un des éléments d'un point de l'espace de recherche, soit d'une permutation. Dans notre cas, cela correspond donc à une ville. Quand une ville doit être mutée, on choisit aléatoirement une autre ville dans ce problème et on intervertit les deux villes (figure 2.14).

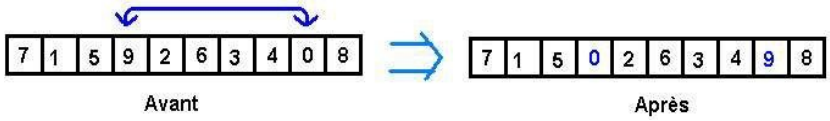


Figure 3.16. Exemple de mutation.

Calcul des fitness

Le seul impératif sur la valeur d'adaptation est qu'elle soit croissante avec l'adaptation de la solution au problème. Un parcours valide renverra une valeur supérieure à un individu (solution potentielle) qui n'est pas solution au problème. On pourra décider par la suite qu'une solution non correcte sera de fitness négative et dans l'autre cas sera alors positive. Lorsque cette dernière éventualité se produit, il faut que le comportement suivant soit respecté, à savoir : plus notre chemin sera court, plus la fitness qui lui est associée sera forte. On en déduit alors rapidement qu'il faut que la valeur d'adaptation varie dans le sens inverse de la distance correspondante au parcours. Par souci de simplicité, on pourra éventuellement choisir comme valeur d'adaptation, l'inverse de la longueur du parcours. On s'aperçoit bien alors que cette algorithme est aléatoire (ou dit approximatif) dans le sens où elle se base sur des méthodes de calculs non déterministes. L'expérience (i.e la programmation) montre que les résultats obtenus sont convaincants, nous parvenons à obtenir un ensemble de bonnes solutions en un temps raisonnable.

Algorithme

Il s'agit dans un premier temps d'établir un ensemble de gênes G. Ceci étant fait, il faut alors définir deux fonctions, qui respectivement permettront de croiser deux ensembles de gênes et de réaliser une mutation sur deux ensembles de gênes.

```
Croiser(G g1,G g2)
{ //résultat du croisement G r ;
    //l'idée est tout simplement pour chaque variable de choisir aléatoirement la variable
     de g1 ou de g2.
    Pour chaque variable v de r
        Si un nombre aléatoire de 0 à 99 est inférieur à 50 alors
            Copier la variable correspondante à v de g1 dans v
        Sinon
            Copier la variable correspondante à v de g2 dans v
        Fin Si
    Fin Pour
    Retourner r ; }
```

Algorithme 3.5. Algorithme de croisement pour le PVC.

```
Muter (G g)
{ Pour chaque variable v de g
        Si un nombre aléatoire de 0 à 99 est inférieur à un certain nombre entre 0 et 99
// On échange deux variables car on ne peut avoir 2 villes semblables dans le même
  parcours.
            Echanger v et vAutre;
    Fin Pour }
```

Algorithme 3.6. Algorithme de mutation pour le PVC.

Dès lors, il nous permutations dénotant un parcours (dans le cas de notre exemple du voyageur de commerce).

Voici donc une procédure sélection qui renvoie les éléments les plus intéressants d'un génomes (ce sera dans le cas du voyageur de commerce, les éléments de fitness la plus grande). Le choix est sinon laissé au programmeur.

```
SelectionNaturelle(G g[],N)
{
// On trie les éléments à la fitness la plus grande vers ceux elle est la plus petite.
    Trier g par les g[i] dans l'ordre décroissant. Retourner les N premiers éléments de g
}
```

Algorithme 3.7. Algorithme de sélection pour le PVC.

A présent que nous avons tous les éléments dont nous avons besoin, nous pouvons construire l'algorithme final :

```
AlgoGene(G g[N])
{ Tant que (g[i] != NULL)        // Tant qu'il y a des parcours .
        G r[N*(N-1)] ;            // N pour gi, N-1 pour gj .
        k=0;
        Pour chaque gi dans g[N]
            Pour chaque gj dans g[N]
                Si gi!=gj alors
                   r[k] =Croiser(gi,gj);
                   Muter(r[k++]);
                Fin si
            Fin Pour
         Fin Pour
         g=SelectionNaturelle(r,N);
    Fin Tant Que }
```

Algorithme 3.8. Algorithme génétique pour le PVC.

Comparaison de complexités

Ce problème est un représentant de la classe des problèmes NP-complets. L'existence d'un algorithme de complexité polynomiale reste inconnue. Les algorithmes pour résoudre le problème du voyageur de commerce peuvent être répartis en deux classes :
- les algorithmes déterministes qui trouvent la solution optimale ;
- les algorithmes d'approximation qui fournissent une solution presque optimale.

Complexité via une méthode de résolution classique (méthode déterministe) :
Un calcul rapide de la complexité montre qu'elle est en *O(n!)* où *n* est le nombre de villes.
En supposant que le temps pour effectuer un trajet est d'une µs, le tableau 2.3 témoigne de l'explosion combinatoire.

NB VILLES	***NB POSSIBILITES***	***TEMPS DE CALCUL***
5	120	120 µs
10	181440	0.18 ms
15	43 MILLIARDS	12 h
20	60 E +15	1928 ans
25	310 E + 21	9,8 Milliards A.

Tableau 3.3. Complexité de la méthode déterministe.

Complexité via une méthode de résolution avec algorithmes génétiques (méthode approximative):

Un calcul de complexité de l'algorithme abstrait nous montre qu'elle est en $O(n^2)$ ou n est le nombre de villes.

Détail du calcul :

O (Algo Gene) = O(n ²) + O (Muter) + O (Croiser) + O(SelectionNaturelle)

= O (n²) + O(n) + O (2n) + O(n)

= O (n²)

NB VILLES	NB POSSIBILITES	TEMPS DE CALCUL
5	25	25 µs
10	100	100 µs
15	225	225 µs
20	400	400 µs

Tableau 3.4. Résultats de la méthode approchée.

Les résultats entre la méthode déterministe et approximative ne sont pas comparables en soi, elles utilisent des méthodes différentes pour arriver à des résultats différents. Comme dis précédemment, dans un cas, on cherche la solution optimale, dans le tableau 2.3, une solution presque optimale.

On s'aperçoit bien à l'aide de ces calculs que pour parvenir à ces solutions on consomme respectivement un temps exponentiel (explosion combinatoire) et un temps polynomial (quadratique) par rapport au nombre de villes.

Exercices

Exercice 1

On veut trouver l'entier (de 0 à 16) qui maximise la fonction $f(x) = \frac{1}{4}\left|15x^2 - x^3\right| + 4$

Résoudre ce problème manuellement par un algorithme génétique en utilisant un codage binaire et une population de 6 individus.

Exercice 2

Vous disposez de 10 cartes numérotées de 1 à 10. Vous devez choisir une façon de diviser celles-ci en 2 piles de telle sorte que la somme des numéros des cartes de la première pile soit aussi proche que possible de 36 et que le produit des numéros des cartes restantes soit aussi proche que possible de 360. Chaque carte pouvant être soit dans P_1, soit dans P_2, il y a 1024 façons de les trier. Quelle est la meilleure ?

– Trouvez un encodage pour l'ensemble de ces solutions.

– Trouvez une fonction permettant d'évaluer la qualité d'une solution.

– Décrivez une implantation d'un algorithme génétique basé sur les deux points précédents,

dans les grandes lignes.

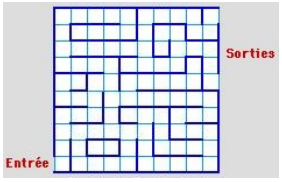
Exercice 3

Proposez une méthode permettant de résoudre le problème des 6 reines à l'aide d'un algorithme génétique.

1. Définissez un codage du problème.
2. Définissez la fonction de fitness associée.

Exercice 4

Proposez une méthode permettant d'obtenir un chemin vers la sortie d'un labyrinthe à l'aide d'un algorithme génétique.



Labyrinthe

1. Définissez un codage du problème.
2. Définissez la fonction de fitness associée

Exercice 5

Utiliser manuellement par un AG pour trouver le maximum de $f(x) = -x^2 + 4x$ dans l'intervalle [1 , 3] avec une précision de 1/10. (Utiliser un codage binaire et une population de 4 individus).

Proposer un codage binaire d'un AG pour trouver Max $f(x ,y) = x^2 + y^2$ dans [-2 , +2]x[-2 , +2] avec une précision de 1/1000 pour x et y.

Exercice 6

On souhaite réaliser une tour 2D à l'aide de N briques de largeur : 2,3,5 et 7 unités. On souhaite construire la tour qui soit à la fois la plus haute et la plus solide. Pour évaluer ce dernier critère, on dispose d'un simulateur capable d'appliquer à la tour l'effet combiné de la gravité et de secousses d'une amplitude donnée en paramètre. Après le calcul, le simulateur retourne une tour, potentiellement différente, résultant du déplacement éventuel des briques durant la simulation ainsi que le nombre de briques ayant bougées. Le simulateur permet également de connaitre la hauteur de la tour (hauteur du plus haut élément).

1. Déterminez une fonction de fitness ainsi que la procédure d'évaluation associée.
2. On suppose qu'une tour ne peut pas faire plus de 32 unités de largeur. On vous propose le codage suivant :
   – Une solution est une suite de N mots de 7 bits.
   – un mot représente le type de brique (codé sur 2 bits) et une position horizontale (sur 5 bits) à laquelle on dépose une brique. La brique est alors déposée le plus bas possible sur la colonne choisie.

   Exemple : 01 00100 correspond a` une brique de taille 3 disposé a 4 unités du point extrême gauche. Quelles sont les qualités et les défauts de ce codage ?

Proposez une autre façon de représenter une tour en cherchant à supprimer les défauts que vous avez identifiés à la question précédente.

CHAPITRE IV

HYBRIDATION DE METAHEURISTIQUES

1. Introduction

L'idée de faire coopérer différents types de méthodes n'est pas nouvelle. Très vite, il est apparu que toutes les méthodes n'avaient pas les mêmes propriétés et on a cherché à profiter des avantages des différentes méthodes et atténuer leurs inconvénients.

2. Coopération méta/méta

Les coopérations étaient à l'origine essentiellement réalisées entre différentes métaheuristiques. A peu près tous les modes d'approches ont été proposés pour ce type de coopération, fait que les métaheuristiques hybrides sont devenues maintenant assez classiques dans le domaine de l'optimisation [BAS 05].

3. Coopération méta/exact

Nous avons vu que les méthodes exactes permettaient de résoudre de petits problèmes tandis que les métaheuristiques sont capables d'appréhender de grands problèmes sans pouvoir donner la solution optimale ou prouver que la solution fournie est optimale. Cependant les méthodes exactes peuvent néanmoins être utiles lorsque des sous-problèmes peuvent être extraits du problème global. Leur résolution permet en effet de contribuer à la recherche de la solution globale, soit en combinant judicieusement différents sous-problèmes, soit en hybridant résolution exacte de sous-problèmes et résolution heuristique du problème complet [DHA 05]. Cette constatation constitue le point de départ d'une approche hybride, assez originale dans le contexte des problèmes d'optimisation combinatoire visant à combiner résolutions exactes et métaheuristiques permettant de conserver aux mieux les avantages de chacune des approches. Nous nous intéressons à la coopération méta/exacte, afin d'obtenir toujours de meilleurs résultats.

4. Classification des méthodes hybrides

L'hybridation est une tendance observée dans de nombreux travaux réalisés sur les métaheuristiques ces dernières années. Elle permet de tirer profit des avantages cumulés des différentes métaheuristiques, à tel point que les métaheuristiques que nous avons vues jusqu'à présent ne soient plus que des canevas, des points de départ, pour commencer à résoudre un problème d'optimisation. On parle d'hybridation quand la métaheuristique considérée est composée de plusieurs méthodes se répartissant les tâches de recherche. La taxinomie des métaheuristiques hybrides se sépare en deux parties : une classification *hiérarchique* et une classification *plate*. Cette classification est applicable aux méthodes déterministes aussi bien qu'aux métaheuristiques.

La classification hiérarchique se fonde sur le niveau (bas ou haut) de l'hybridation et sur son application (en relais ou concurrente). Dans une hybridation de *bas niveau*, une fonction donnée d'une métaheuristique (par exemple, la mutation dans un algorithme évolutionnaire)

est remplacée par une autre métaheuristique (par exemple une recherche avec tabou). Dans le cas du *haut niveau*, le fonctionnement interne « normal » des métaheuristiques n'est pas modifié. Dans une hybridation en *relais*, les métaheuristiques sont lancées les unes après les autres, chacune prenant en entrée la sortie produite par la précédente. Dans la concurrence (ou *coévolution*), chaque algorithme utilise une série d'agents coopérants ensembles (Figure 1.18.)

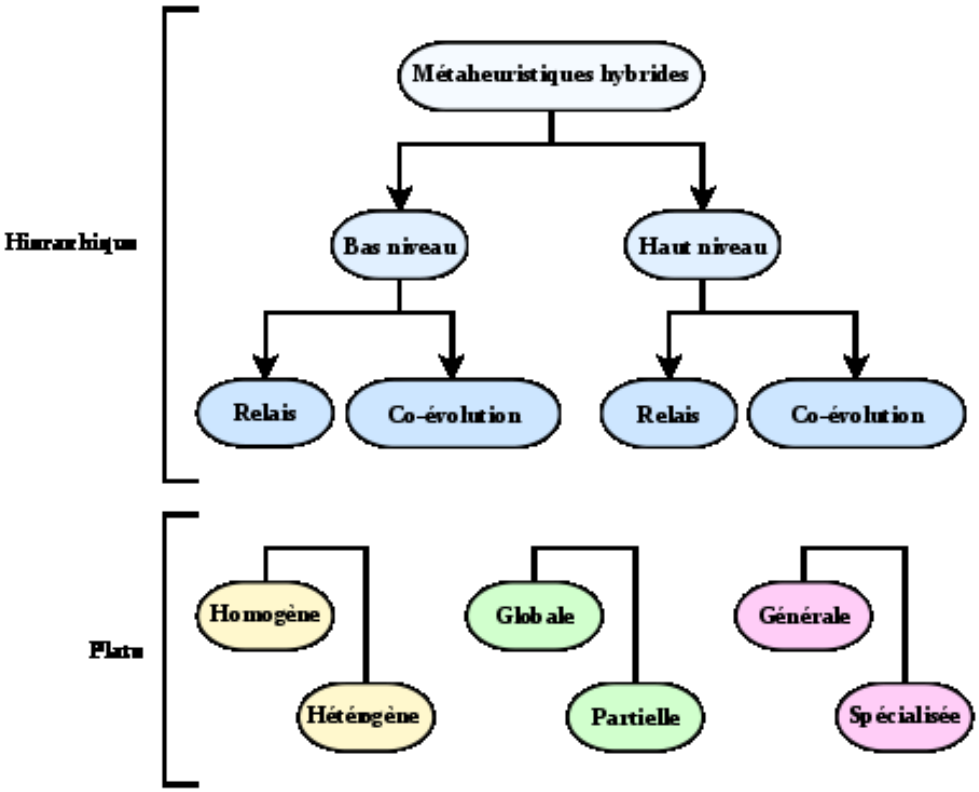


Figure 4.1. Taxinomie des métaheuristiques hybrides (Wikipédia).

Cette première classification dégage quatre classes générales :

- bas niveau et relais (abrégé *LRH* en anglais) (figure 1.19).
- bas niveau et coévolution (abrégé *LCH*), (figure 1.20),
- haut niveau et relais (*HRH*), (Figure 1.21).
- haut niveau et coévolution (*HCH*). (Figure 1.22).

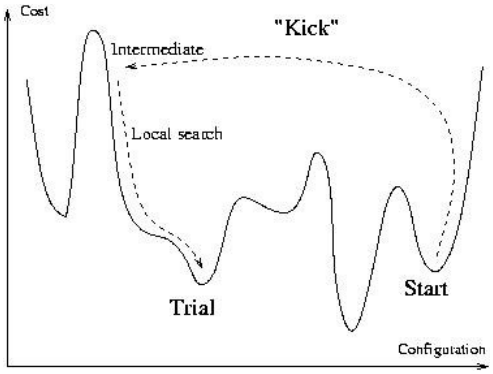


Figure 4.2. Exemple LRH: Hybridation recherche locale dans le recuit simulé.

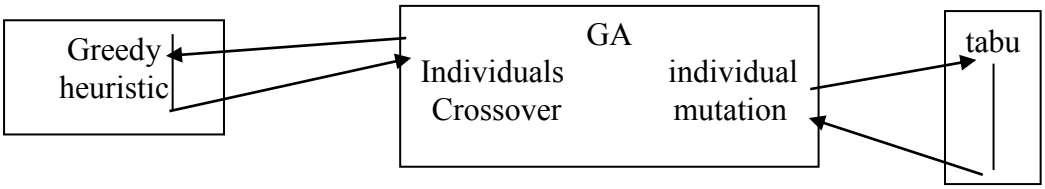


Figure 4.3. Un exemple d"hybridation LCH.

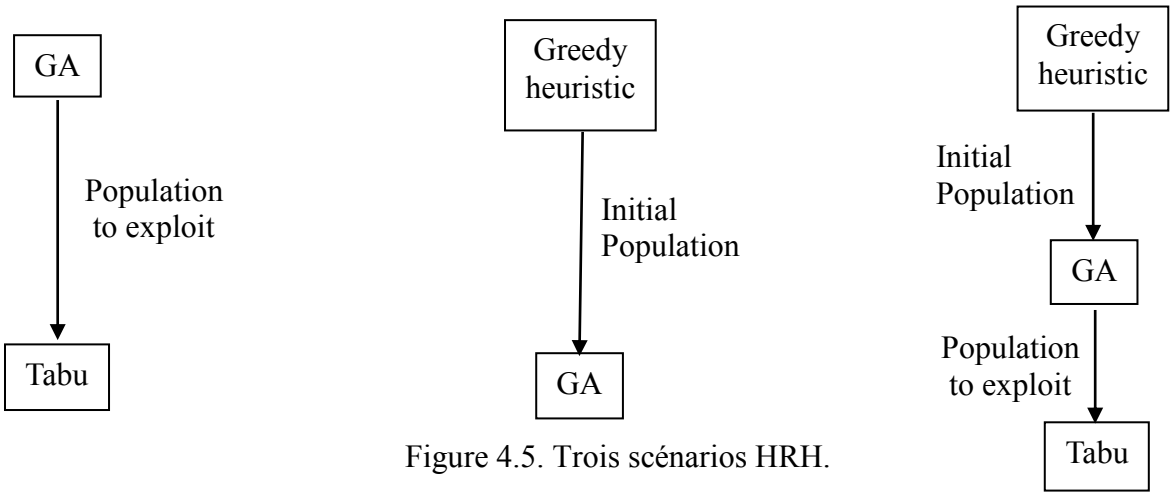


Figure 4.5. Trois scénarios HRH.

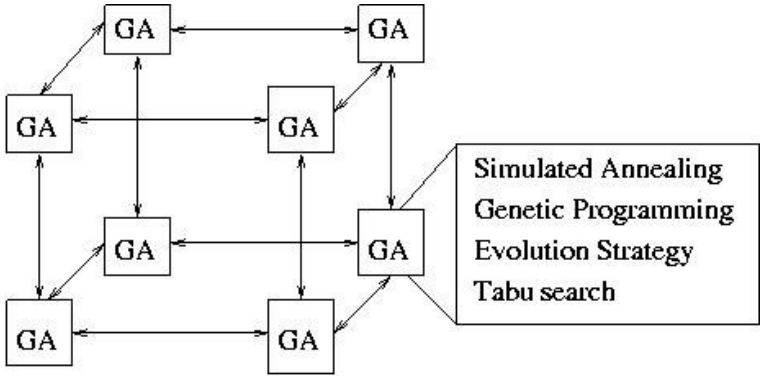


Figure 4.6. Un exemple d'un scénario HCH avec un algorithme génétique.

La seconde partie dégage plusieurs critères, pouvant caractériser les hybridations :

- si l'hybridation se fait entre plusieurs instances d'une même métaheuristique, elle est homogène, sinon, elle est hétérogène ;
- si les méthodes recherchent dans tout l'espace de recherche, on parlera d'hybridation globale, si elles se limitent à des sous-parties de l'espace, d'hybridation partielle ;
- si les algorithmes mis en jeu travaillent tous à résoudre le même problème, on parlera d'approche générale, s'ils sont lancés sur des problèmes différents, d'hybridation spécialisée.

Ces différentes catégories peuvent être combinées, la classification hiérarchique étant la plus générale. Une des techniques les plus populaires d'hybridation concerne l'utilisation de métaheuristique de type « trajectoire locale » avec des métaheuristiques à population. La plupart des applications réussies d'algorithmes génétiques, ou à colonies de fourmi, sont complétées par une phase de recherche locale, car c'est ce qui leur manque à l'origine. Les individus, les fourmis, tentent d'identifier les zones prometteuses de l'espace des solutions, lesquelles sont ensuite explorées plus en détail par des méthodes de recherche locale (par l'une des métaheuristiques que nous avons déjà vues, ou par une heuristique classique, comme celle du plus proche voisin, ou par un algorithme de type *2-opt*).[BAC 98]

Classification à plat des méthodes hybrides.
Les figures 1.23 et 1.24 illustrent deux scénarios d'hybridation: le premier est hétérogène collaboratif mêlant branch & bound, un AG, le recuit simulé et une recherche tabou. Le second est homogene a relais melant plusieurs instances de recherche tabu ensuit un AG.

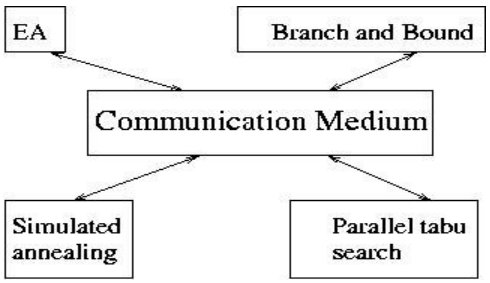


Figure 4.7. Un exemple d'une hybridation hétérogène.

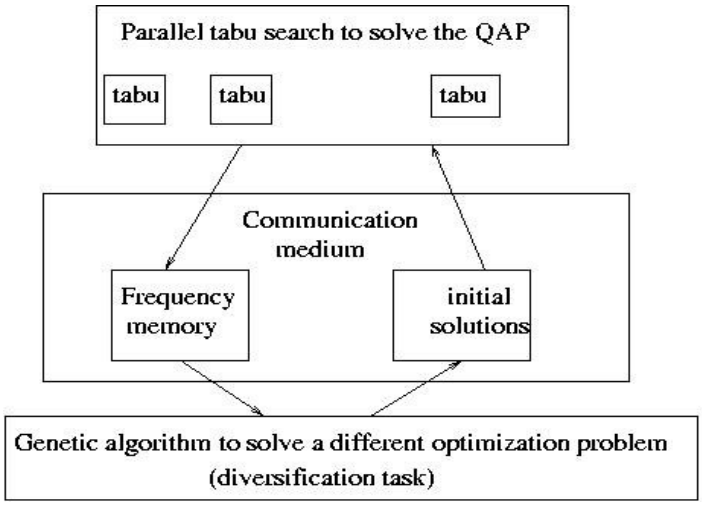


Figure 4.8. Un exemple d'une hybridation homogène.

Ces deux types de recherche, globale et locale, peuvent d'ailleurs s'exécuter de façon asynchrone, sur des processeurs différents.
Le problème d'allocation d'espace a fait l'objet d'une étude combinant une métaheuristique de type évolutionnaire, avec une recherche locale à voisinage variable. Ce problème consiste à répartir un certain nombre de personnes et de ressources matérielles dans un certain nombre de pièces contient généralement de nombreuses contraintes : nécessité que des personnes se retrouvent ensemble dans une même pièce, que certaines personnes disposent de certaines ressources avec elles dans la pièce, etc.
Le voisinage variable permet, au travers de quelques opérations élémentaires (allocation, désallocation, réallocation, échange…) de fournir des solutions à la fois faisables (au regard du nombre de contraintes) et variées.

Une autre manière d'hybrider consiste à exécuter en parallèle 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. Cette technique est d'autant plus utilisée que s'est approfondie la recherche sur les architectures parallèles. Toutes les métaheuristiques classiques ont fait l'objet d'une implémentation parallèle, y compris l'algorithme du recuit simulé qui, bien que de nature séquentielle, a pu être parallélisé en le divisant en processus élémentaires.

Enfin une troisième forme d'hybridation combine les métaheuristiques avec des méthodes exactes. Une méthode exacte peut ainsi donner lieu à une technique efficace pour la détermination du meilleur voisin d'une solution (ce qui peut s'avérer plus judicieux que de choisir la meilleure solution parmi un petit échantillon de voisins). Dans le domaine de la programmation par contraintes, certaines techniques de retour-arrière non déterministes (backtracking) ont été associées à des métaheuristiques.

5. Travaux réalisés

En 2010, dans le cadre de son projet d'habilitation, Laetitia JOURDAN [JOU 10] avait élaboré une taxonomie exhaustive et minutieuse d'hybridations méta/méta et méta/exacte. Concernant cette dernière catégorie, elle a distingué les classes suivantes :

- ✓ Utilisation d'un algorithme exact pour explorer des larges voisinages dans un algorithme de recherche locale.
- ✓ Utilisation des solutions de bonne qualité afin de réduire l'espace de recherche de la méthode exacte.
- ✓ Exploitation des bornes de la méthode exacte pour une heuristique constructive.
- ✓ Utilisation des informations fournies par les relaxations des problèmes linéaires pour orienter un algorithme de recherche locale ou constructif.
- ✓ Utilisation d'une méthode exacte pour une fonction spécifique de la métaheuristique.

L'état de l'art de Stützle et Dumitrescu survole les grandes 'branches' de la coopération méta/exacte, mais les exemples proposés utilisent, dans la plupart des cas, des recherches locale ou Tabou pour métaheuristique.

Cette taxinomie propose une approche plus générale de la classification, qui permet de regrouper les coopérations méta/méta et méta/exacte. De plus, la classification proposée dans cette section permet d'inclure facilement des nouveaux schémas de coopération.

Design	classification	Problème
LTH (PD(RLI))	(approchée, partielle, générale) séquentielle	Voyageur de Commerce
HRH (B&P + HS)	(approchée, partielle, générale) séquentielle	Partitionnement large échelle
LRH (RL(PD))	(approchée, partielle, générale) séquentielle	Voyageur de commerce asymétrique
HRH (RT + PL)	(approchée, partielle, générale) séquentielle	Coupes irrégulières
HRH (HG + LTH (RT(B&B)))	(approchée, partielle, générale) séquentielle	Design réseaux de vols directs
HRH(HRH(HG + RL) + B&C)	(exacte, globale, générale) séquentielle	Routage de véhicules avec fenêtres de temps
HRH(RS + LTH(RL(B&B)))	(approchée, partielle, générale) séquentielle	Routage de véhicules avec fenêtres de temps
LTH(AG(B&B))	(approchée, partielle, générale) parallèle statique	Voyageur de commerce
HRH(RL + PL)	(approchée, partielle, générale) séquentielle	Problème de fabrication de câbles à fibre optique (ordonnancement)
HTH(B&P+HRH(PLM+RL)))	(exacte, globale, spécialiste) séquentielle	Routage de véhicules
LTH(RL(PD))	(approchée, partielle, générale)	Flow-shop à une machine

	séquentielle	(somme pondérée des retards)
HRH(RT + RD)	(approchée, partielle, générale) séquentielle	Fonctions continues à multi-minima
HRH(RT + NMS)	(approchée, partielle, générale) séquentielle	Fonctions continues à multi-minima
LTH(AG(B&B))	(approchée, partielle, générale) parallèle statique	Emploi du temps
HRH(AG + NMS)	(approchée, partielle, générale) séquentielle	Fonctions continues
LTH(RT(B&B))	(approchée, partielle, générale) séquentielle	Tournées de véhicules
HTH(CF+HTH(B&B + HS))	(approchée, partielle, générale) séquentielle	Design d'accès au réseau local
HRH(PL + AG)	(approchée, partielle, générale) séquentielle	Affectation généralisée
LRH(B&B + AG)	(exacte, globale, générale) séquentielle	Max-SAT
HRH(HS + B&B)	(approchée, partielle, générale) séquentielle	Flow-shop
LRH(MST(LTH(AG(B&B|ACM))))	(approchée, globale, générale) séquentielle	Voyageur de commerce
HTH(AG + B&C)	(approchée, partielle, générale) séquentielle	Tournée de véhicules multi-objectif
LTH (PLM(PG))	(exacte, globale, spécialiste) séquentielle	Jeux de données MIPLIB3
HRH (PL + AM)	(approchée, partielle, générale) séquentielle	Problème de l'arbre de Steiner
HRH (αB&B + CSA)	(approchée, partielle, spécialiste) parallèle statique	Prédiction de structure de protéines
LRH(CF(PL))	(approchée, globale, spécialiste) séquentielle	Affectation quadratique
HRH(B&B + RT)	(approchée, partielle, générale) séquentielle	Affectation quadratique
HTH(RL+PC)	(approchée, partielle, générale) séquentielle	Ordonnancement de projets avec contrainte de ressources
HTH(RL+PC)	(approchée, partielle, générale) séquentielle	Problème du voyageur de commerce
HRH(AD + B&B)	(exacte, globale, générale) séquentielle	Affectation quadratique
HRH(HS + (HTH(B&B+AG)))	(exacte, globale, générale) séquentielle	Flow-shop hybride

HRH(HS+B&B)	(approchée, partielle, générale) séquentielle	Problème p-median
LTH(RL(B&B))	(approchée, partielle, générale) séquentielle	Routage de véhicules
HRH(B&B + RN)	(approchée, partielle, générale) séquentielle	Ordonnancement de diffusion d'informations biobjectif
HRH(MS + CF)	(approchée, partielle, spécialiste) séquentielle	Flow-shop biobjectif sur 2 machines
LRH (RLI(AS))	(approchée, globale, spécialiste) séquentielle	Variante du problème de découpe à une dimension
HRH (AS + RT)	(approchée, partielle, générale) séquentielle	0-1 Sac à dos multidimensionnel
HRH (AG+B&B)	(exacte, partielle, général) Parallèle	Flowshop bi-objectif

Tableau 4.1. Recueil des références des œuvres d'hybridation [JOU 10].

6. Grammaire des méthodes hybrides

Dans le but de spécifier les méthodes hybrides Talbi et al. [TAL 02] avait proposé une grammaire bien détaillée avec toutes ses composantes de la théorie des langages, en l'occurrence : le vocabulaire terminal, le vocabulaire non terminal, l'axiome et les règles de production. Ce pendant cette grammaire ne spécifie pas l'hybridation métaheuristiques / exactes. En effet, en 2010, L. Jourdan [TAL 02] avait repris et étendu cette grammaire pour qu'elle définisse, en outre, les méthodes hybrides de type meta/exact.

< algorithme coopératif >→ < conception >< implémentation >
< conception > → < hiérarchique >< plat >
< hiérarchique > → < LRH > | < LTH > | < HRH > | < HTH >
< LRH >→ LRH (< métaheuristique >(< exact >)|(< exact >(< métaheuristique >))
< LTH >→ LTH (< métaheuristique >(< exact >)|(< exact >(< métaheuristique >))
< HRH >→ HRH ((< métaheuristique >+< exact >)|(< exact >+< métaheuristique >))
< HTH >→ HTH ((< métaheuristique >+< exact >)|(< exact >+< métaheuristique >))
< plat >→ (< résolution >, < optimisation >, < fonction >)
< résolution >→ exacte | approchée
< optimisation >→ globale | partielle
< fonction > → générale | spécialiste
< implémentation > → séquentielle | parallèle < type parallèle >
< type parallèle > → statique | dynamique | adaptative
< métaheuristique > → RS | AG | AM | SE | PG | RN | AD | RL | RLI | RT | HG | CF | RD | HS | CSA | < algorithme coopératif >
< exact > → B&B | αB&B | B&C | B&P | B&C&P | PL | PLM | PD | AS | RB | MS | <algorithme coopératif >

Grammaire des méthodes hybrides [JOU 10]

L'auteur de cette grammaire fournit une un outil formel de spécification détaillée des méthodes hybrides en les considérants comme les mots d'un langage formel avec toutes ses composantes : vocabulaire terminal, vocabulaire non terminal, axiome et règles de dérivations. Cette grammaire aura certainement une grande ampleur pour ceux qui s'intéressent aux méthodes hybrides.

BIBLIOGRAPHIQUES

REFERENCES BIBLIOGRAPHIQUES

[ALV 10] R. Alvarez et al., Minimizing weighted earliness–tardiness on a single machine with a common due date using quadratic models, Sociedad de Estadística e Investigación Operativa 2010. Springer, Top (2012) 20:754–767.

[BAC 98] Vincent Bachelet, Zouhir Hafidi, Philippe Preux, El-Ghazali Talbi. Vers la coopération des métaheuristiques. Calculateurs parallèles. Vol. 09. N. 02. 1998.

[BAK 09] Kenneth R. Baker, Dan Trietsch, Principles Of Sequencing And Scheduling. A JOHN WILEY & SONS, INC. PUBLICATION. 2009.

[BAK 13] Kenneth R. Baker, Minimizing Earliness and Tardiness Costs in Stochastic Scheduling. European Journal of Operational Research. 2013.

[BAK 74] Baker. K.R. Introduction to Sequencing and Scheduling. John Wiley & Sons, 1974.

[BAK 78] Baker. K.R and Schrage L.E., Finding an optimal sequence by Dynamic Programming an Extension to precedence-related tasks. Oper. Res. 26, 111-120. 1978.

[BAK 88] Baker. K.R and Scudder G. D. 1988. Sequencing with earliness and tardiness penalties. A review. Oper. Res. 38, 22-36. 1988.

[BAS 05] M. Basseur, Conception D'algorithmes Coopératifs Pour L'optimisation Multiobjectif : Application Aux Problèmes D'ordonnancement De Type Flow-Shop». Université des sciences et technologies de Lille U.F.R. D'I.E.E.A. thèse pour obtenir le grade de Docteur de l'U.S.T.L. 2005.

[BEL 85] Belouadah H. Scheduling to minimize Total Cost. MSC. Thesis University of Keele. U.K. 1985.

[BLU 03] Christian Blum, Andrea Roli. Metaheuristics in Combinatorial Optimization: Overview and Conceptual Comparison. ACM Computing Survey, Vol. 35 No 3, Sept. 2003.

[CAR 88] Carlier J. et Chrétienne P. Problèmes d'ordonnancement Modélisation complexité/algorithmes. Masson. 1988.

[CHA 02] C. N. Cha et al. , Single-Machine Job Scheduling about a Common Due Date with Arbitrary Earliness / Tardiness Penalties Using a Genetic Algorithm. Asia Pacific Management Review (2002) 7(2), 239-254.

[CHE 13] T. C. E. Cheng, Shih-Chang Tseng, Peng-Jen Lai, Wen-Chiung Lee, Single-Machine Scheduling with Accelerating Learning Effects, Mathematical Problems in Engineering, Volume 2013, Article ID 816235, 7 pages.

[CHE 91] P. Cheeseman, B. Kanelfy, andW. Taylor. Where the really hard problems are. In Proceedings of IJCAI'91, Morgan Kaufmann, Sydney, Australia, pages 331–337, 1991.

[CHU 96] Chu C. and Proth J. M. L'ordonnancement et ses applications. MASSON. Paris, 1996.

[CLA 96] D.A. Clark, J. Frank, I.P. Gent, E. MacIntyre, N. Tomv, and T. Walsh. Local search and the number of solutions. In Proceedings of CP'96, LNCS 1118, Springer Verlag, Berlin, Germany, pages 119–133, 1996.

[CON 98] R. K. Congram, C. N. Potts, and S. L. Van De Velde. An iterated dynasearch algorithm for the single-machine total weighted tardiness scheduling problem. INFORMS Journal on Computing, 14 :52–67, 1998.

[DAV 95] A. Davenport. A comparison of complete and incomplete algorithms in the easy and hard regions. In Proceedings of CP'95 workshop on Studying and Solving Really Hard Problems, pages 43–51, 1995.

[DHA 05] Optimisation Combinatoire Multiobjectif: Apport Des Méthodes Coopératives Et Contribution A L'extraction De Connaissances. C. Dhaenens-Flipo. Université des sciences et technologies de Lille, U.F.R. D'I.E.E.A. thèse pour obtenir le grade d'habilitation à diriger des recherches de l'U.S.T.L. 2005.

[DOR 00] Marco Dorigo and Gianni Di Caro. Ant Algorithms for Discrete Optimization. Université Libre de Bruxelles. 2000.

[DRE 08] Zvi Drezner, Tabu Search and Hybrid Genetic Algorithms for Quadratic Assignment Problems. Local Search Techniques: Focus on Tabu Search, Book edited by: Wassim Jaziri, ISBN 978-3-902613-34-9, pp. 278, October 2008, I-Tech, Vienna, Austria.

[DUN 05] T. Dunker, G. Radons, and E. Westkämper. Combining evolutionary computation and dynamic programming for solving a dynamic facility layout problem. European Journal of Operational Research, 165(1) :55–69, 2005.

[ESQ 99] Esquirol P. Lopez P. L'ordonnancement, série: production et techniques quantitatives appliqués à la gestion, ECONOMICA, Paris. 1999.

[FEL 01] M. Feldman, and D. Biskup, "Benchmarks for scheduling on a single machine against restrictive and unrestrictive common due dates", Computer & Industrial Engineering, 28 (2001) 787-801.

[FEL 03] Feldmann M. and Biskup D. Single Machine Scheduling For Minimizing Earliness And Tardiness Penalties By Meta-Heuristic Approaches. Computers & Industrial Engineering 44 (2003) 307-323.

[FOU 05] Olivier Fourdrinoy, Hybridation des méthodes de résolution pour SAT. Centre de Recherche en Informatique de Lens, CNRS FRE 2499. Université d'Artois. RJCIA 2005.

[FRE 82] French S. Sequencing and Scheduling: An Introduction to the Mathematics of the Job-Shop, Wiley, New York. 1982.

[GAG 01] Caroline Gagné, Marc GRravel, Wilson L. Price. Optimisation par colonie de fourmis pour un problème d'ordonnancement industriel avec temps de réglages dépendants de la séquence. 3° Conférence Francophone de Modélisation et Simulation "Conception, Analyse et Gestion des Systèmes Industriels". Avril 2001.

[GEN 03] Michel Gendreau, Gilbert Laporte, Frédéric Semet. The maximal expected coverage relocation problem for emergency vehicles. Juillet 2003.

[GEN 96] I.P. Gent, E.MacIntyre, P. Prosser, and T.Walsh. The constrainedness of search. In Proceedings of AAAI-96, AAAI Press, Menlo Park, California, 1996.

[GLO 98] Fred Glover, Manuel Laguna. Tabou search. 1998.

[GOL 89a] D. E. Goldberg. Genetic Algorithms. AddisonWesley, 1989. ISBN: 0-201-15767-5.

[GOL 89b] D. E Goldberg. Genetic algorithms and Walsh functions. part 1 and 2. Complex Systems, 3:129–171, 1989.

[GOL 89c] D.E Goldberg. Genetic Algorithms in Search, Optimization and Machine Learning. Reading MA Addison Wesley, 1989.

[GOL91] D.E Goldberg. Real-coded genetic algorithms, virtual alphabets and blocking. Complex Systems, 5:139–167, 1991.

[GON 10] Miguel A. Gonz´alez et al., Tabu Search and Genetic Algorithm for Scheduling with Total Flow Time Minimization. COPLAS 2010: ICAPS Workshop on Constraint Satisfaction Techniques for Planning and Scheduling Problems.

[HAC 13] Hanaa Hachimi, Hybridations d'algorithmes métaheuristiques en optimisation globale et leurs applications. THÈSE DE DOCTORAT EN COTUTELLE. Université Mohammed V - Agdal, Rabat. 2013.

[HAO 99] Jin-Kao Hao, Philippe Galinier, Michel Habib. Métaheuristiques pour l'optimisation combinatoire et l'affectation sous contraintes. Revue d'intelligence artificielle, 1999.

[HEI 71] M. Heidari, V. T. Chow, P. V. Kokotovifa, Da. D. Meredith. Discrete differential dynamic programing approach to water resources systems optimization. Water Resources Research, 7(2) :273–282, 1971.

[HEL 62] Helde H. Karp R.M. A Dynamic Programming Approach to Sequencing Problems. SIAM. J. 10, 196-210. 1962.

[HEM 10] Allaoua Hemmak, Ibrahim H. Osman, Variable Parameters Lengths Genetic Algorithm for Minimizing Earliness-Tardiness Penalties of Single Machine Scheduling With a Common Due Date, Electronic Notes in Discrete Mathematics. 36 (2010) 471–478.

[HEM 13a] Allaoua Hemmak, Brahim Bouderah, Hybrid Algorithm for Optimization Problems Applied to Single Machine Scheduling, International Journal of Computer Applications. Volume 66– No.24, 9-15. March 2013.

[HEM 13b] Allaoua Hemmak, Brahim Bouderah, Sieve Algorithm - A New Method for Optimization Problems, Int. J. Advance. Soft Comput. Appl., Vol. 5, No. 2, 1-21. July 2013.

[HEM 15] Allaoua Hemmak, Brahim Bouderah, A mono crossover genetic algorithm for TSP, Global Journal on Technology, Issue 7 (2015) 109-115.

[HIN 04] Hino C. M., Ronconi D. P. Mendes A. B. 2004. minimizing earliness and tardiness penalties in a single machine with a common due date. European

Journal. O. R. 160 (2005) 190-201.

[HOG 96] T. Hogg. Refining the phase transition in combinatorial search. Artificial Intelligence, pages 127–154, 1996.

[HOG 98] T. Hogg. Exploiting problem structure as a search heuristic. Intl. J. of Modern Physics C, 9 :13–29, 1998.

[HOL 75] J. H. Holland: Adaptation In Natural And Artificial Systems, University of Michigan Press (1975).

[HOO 91] J. A. Hoogeveen, S. L. Van, De Velde, Scheduling around a small common due date. European Journal of O. R. 55 (1991) 237-242.

[IBA 92] Ibaraki T. Nakamura Y. 1992. A Dynamic Programming Method For Single Machine Scheduling. European. Journal. O. R. 76 (1994) 72-82.

[ICH 01] Soumia Ichoua. Problèmes de gestion de flottes de véhicules en temps réel. Université de Montréal, Octobre 2001.

[IGN 65] Ignall. E. , Schrage. L. Application of The Branch and Bound Technique to some Flow-Shop Scheduling Problems. Oper. Res. 13, 400-412. 1965.

[JAC 15] Sohie Jacqui, Thèse de doctorat: Hybridation des métaheuristiques et de la programmation dynamique pour les problèmes d'optimisation mono et multi-objectifs: application à la production d'énergie, EDSPI, novembre 2015.

[JAF 13] Ahmad Jafarnejad et al., Optimizing the Earliness and Tardiness Penalties in the Single-machine Scheduling Problems with Focus on the Just in Time.

International Journal of Academic Research in Business and Social Sciences July 2013, Vol. 3, No. 7 ISSN: 2222-6990 315.

[JOH 05] Johann Dréo, Alain Pétrowski, Patrick Siarry, Eric Taillard. Métaheuristiques pour l'optimisation difficile. Eyrolles, 2005.

[JOU 09] L. Jourdan et al., Hybridizing exact methods and metaheuristics: A taxonomy. European Journal of Operational Research 199 (2009) 620–629.

[JOU 10] Laetitia JOURDAN, Métaheuristiques Coopératives : du déterministe au stochastique, Habilitation à Diriger les Recherches de l'Université Lille I, 2010.

[KAH 93] H. G. Kahlbacher. Scheduling With Monotonous Earliness And Tardiness, European journal of O. R. 64 (2) (1993) 258-277.

[KAN 76] Rinnooy Kan A. H. G. Machine Problems: Classification, Complexity and Computations. Martinus Nijhoff, The Hague, Holland. 1976.

[KAR 72] Karp. R. M. Reducibility AMONG Combinatorial Problems. In: R. E. Miller & J. W. Thatcher (eds) Complexity of Computer Computations. Plenum Press, New York. 85-103. 1972.

[KEN 88] Kenneth R. Baker , Gardy D. Scudder. 1988. Sequencing with Earliness and Tardiness Penalties : A Review. 1988.

[LAE 09] Joerg Laessig, Oliver Kramer, Common Due-Date Problem: Exact Polynomial Algorithms for a Given Job Sequence.

[LAN 60] Land A. H., Doig. A. G. An Automatic Method for Solving Discrete Programming Problems. Econometrica 28, 497-520. 1960.

[LAS 14] Jörg Lässig, Abhishek Awasthi, Oliver Kramer, Common Due-Date Problem: Linear Algorithm for a Given Job Sequence, IEEE 17th International Conference on Computational Science and Engineering. 2014.

[LAW 64] Lawler E. L. On Scheduling Problems with Deferral Costs. Management Sci. 11, 280-520. 1964.

[LAW 77] Lawler E.L. A pseudoplynomial algorithm for sequencing jobs to minimize total tardiness. Annals of Discrete Mathematics, 1,331-342. 1977.

[LEI 05] Mme DRDI Leila (INRS-ETE, 2005) LES ALGORITHMES GÉNÉTIQUES.

[LI 08] Y. Li, J. Li, D. Zhao. Dynasearch algorithms for solving time dependent traveling salesman problem. Journal of Southwest Jiaotong University, 2 :009, 2008.

[LI 90] G. Li, R. Matthew. New approach for optimization of urban drainage systems. Journal of Environmental Engineering, 116(5) :927–944, 1990

[LIA 03] Liao C. J. Chen W. J. Single machine scheduling with periodic maintenance and nonresumable jobs. Computers and O. R. 30,1335-1347. 2003.

[LIA 07] Ching-Jong Liao, Che-Ching Cheng, A variable neighbourhood search for minimizing single machine weighted earliness and tardiness with common due date, Computers & Industrial Engineering 52 (2007) 404–413.

[LIA 92] Liao C-J, You C-T An Improved Formulation on the Job-Shop Scheduling Problem. J Oper. Res. 43, 1047-1054. 1992.

[LIN 07] Lin S. W., Chou S. Y., Ying K. C. A Sequential Exchange Approach For Minimizing Earliness-Tardiness Penalties Of Single-Machine Scheduling With A Common Due Date. European Journal O. P. (2007).

[LUG 02] LUGER, George F., Atrificial Intelligence: Structures & Strategies for complex problem solving, 4ème édition, Addison-Wesley, 2002, 856p.

[MAC 98] E. MacIntyre, P. Prosser, B. Smith, T. Walsh. Random constraints satisfaction: theory meets practice. In CP98, LNCS 1520, pages 325–339. Springer Verlag, Berlin, Germany, 1998.

[MAR 02] Posner M. E. Nicholas G. H. Generating experimental data for computational testing with machine scheduling applications. Operations Research, pp 854-865. 2002.

[MIT 95] Genetic Algorithms: An Overview, Melanie Mitchell, Santa Fe Institute. 1399 Hyde Park Road Santa Fe, NM 87501, 1995.

[MUT 96] Y. Mutsunori, I. Toshihide. The use of dynamic programming in genetic algorithms for permutation problems. European Journal of Operational Research, 92: 387– 401, 1996.

[NEA 08] Andreas C. Nearchou, A differential evolution approach for the common due date early/tardy job scheduling problem, Computers & Operations Research 35 (2008) 1329 – 1343.

[NEG 02] NEGNEVITSKY, Michael, Artificial Intelligence: A guide to Intelligent Systems. Addison-Wesley, 2002, 394p.

[OUY 92] Z. Ouyang, S.M. Shahidehpour. A hybrid artificial neural network-dynamic programming approach to unit commitment. Power Systems, IEEE Transactions on, 7(1) : 236–242, 1992.

[PAP 94] C. Papadimitriou. Computational Complexity. Addison Wesley, 1994.

[PAR 98] Y. Park, J. Park, and J. Won. A hybrid genetic algorithm/ dynamic programming approach to optimal long-term generation expansion planning. Elsevier Science, 20 : 295–303, 1998.

[PHA 13] Pham D. T. et al., A Survey on Hybridizing Genetic Algorithm with Dynamic Programming for Solving the Traveling Salesman Problem. 2013 Online Proceedings on Trends in Innovative Computing (ICT).

[RON 10] DEBORA P. RONCONI, MARCIO S. KAWAMURA, The single machine earliness and tardiness-scheduling problem: lower bounds and a branch-and-bound algorithm, computational and applied mathematics, Volume 29, N. 2, pp. 107–124, 2010.

[SAK 84] Sakarovitch M. Optimisation Combinatoire. Méthodes mathématiques et algorithmiques, Programmation discrets. HERMANN. Paris. 1984.

[SBI 03] Sbihi A., Les méthodes hybrides en optimisation combinatoire : algorithmes exacts et heuristiques. Thèse de doctorat, Université Paris-I Panthéon-Sorbonne, 2003.

[SNI 06] M. Sniedoviech, S. Voss. The corridor method: a dynamic programming inspired metaheuristic. Control and Cybernetics, 35 :551–578, 2006.

[SOU 06] F. Sourd. Dynasearch for the earliness–tardiness scheduling problem with release dates and setup constraints. Operations Research Letters, 34(5) :591–598, 2006.

[TAI 02] Éric Taillard. Principes d'implémentation des métaheuristiques, in chapitre 2 de J. Teghem, M. Pirlot (dir.), Optimisation approchée en recherche opérationnelle. Hermès, 2002, p. 57-79.

[TAL 02] E.-G. Talbi. A taxonomy of hybrid metaheuristics. Journal of Heuristics, 8(2) :541-564, 2002.

[TAN 75] W. H. Tang, B. C. Yen, L. W. Mays. Optimal risk-based design of storm sewer networks. Journal of the Environmental Engineering Division, 101(3) :381–398, 1975.

[THO 03] Thomas Vallée et Murat Yıldızoğlu, Présentation des algorithmes génétiques et de leurs applications en économie. v. 4.2, Décembre 2003.

[TIS 05] Tison. S. Programmation Dynamique. Cours master informatique. UFR IEEA. 2005.

[TOS 05] J. Tospornsampan, I. Kita, M. Ishii, and Y. Kitamura. Optimization of a multiple reservoir system operation using a combination of genetic algorithm and discrete differential dynamic programming. A case study in mae klong system. Thailand. Paddy and Water Environment, 3(1) , 29-38, 2005.

[VOI 04] Sophie Voisin, Application des algorithmes génétiques. Rapport DEA, 2004 , P 16-21.