**Sup Galilée - Spécialité Informatique**
**Module ROA**

# Recherche Opérationnelle et Applications
# Résolution exacte ou à garantie expérimentale

Pierre Fouilhoux

Sup Galilée - Sorbonne Paris Nord

2020-2021

1. Résolution exacte ou à garantie

2. Heuristiques primales ou Matheuristique

3. Algorithmes gloutons

4. Meta-heuristiques

1. Résolution exacte ou à garantie
1.1 Trouver des solutions...
1.2 Solution à garantie
1.3 Solution avec garantie expérimentale
1.4 Résolution exacte

2. Heuristiques primales ou Matheuristique

3. Algorithmes gloutons

4. Meta-heuristiques

# Trouver des solutions...

On dit souvent que tous les moyens sont bons pour découvrir une bonne solution d'un problème complexe comme les PMD. On peut diviser ces "moyens" en plusieurs catégories :

• Heuristique spécifique : suivant les problèmes une idée heuristique conduit parfois à de bonnes solutions (par exemple reproduire l'expérience d'un spécialiste)

• Méta-heuristique : il s'agit du cadre d'heuristiques qui reproduisent des principes généraux d'amélioration de solutions réalisables. Elles sont souvent utilisables pour des instances de très grandes tailles (là où les méthodes exactes échouent).

• Heuristique primale (ou matheuristique) : dans le cadre de la programmation linéaire, une relaxation continue fournie une solution fractionnaire (que l'on appelle solution primale) qui peut-être "proche" d'une solution entière réalisable. On peut ainsi imaginer en quelque sorte "d'arrondir" ces solutions fractionnaires.

• On peut noter également que les moteurs d'inférences de la programmation par contrainte permettent parfois de déduire des solutions réalisables de manière générique et efficace. Dans ces cas, on peut construire des algorithmes de branchement efficace (appelés parfois méthode Dive&Fix).

# Solution à garantie théorique

Les méthodes précédentes ont été décrites dans le but de donner des idées pour produire des solutions réalisables dites "approchées". Il est bien plus intéressant de produire des solutions pour lesquelles on peut indiquer son éloignement à l'optimum. On parle alors de *solutions garanties* (provably good solutions).

• *Relation Min-Max* : on appelle une relation Min-Max un encadrement de la solution optimale d'un problème par une borne min et une borne max obtenue par un même algorithme. Cet algorithme est le plus souvent obtenu en utilisant la théorie de la dualité ou des propriétés combinatoires particulière.

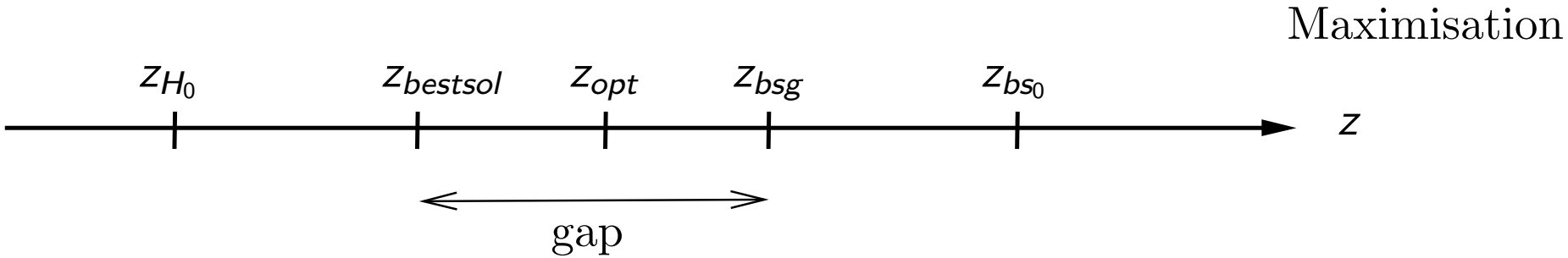
• *Algorithme d'approximation* : on appelle algorithme d'$\alpha$-approximation un algorithme (polynomial ou à la rigueur efficace) donnant une solution qui est au pire à $\alpha$% de l'optimum. Dans le cas d'une maximisation, cela siginifie que la valeur $z_{approx}$ est telle que $\alpha z_{opt} \leq z_{approx} \leq z_{opt}$.
Ces algorithmes sont le plus souvent très spécifiques et sont basés sur des principes algorithmes parfois très fins. Il est également possible que des technique d'arrondis ou des heuristiques primales puissent fournir des algorithmes d'approximation intéressant. Ces techniques demandent des preuves théoriques poussées et fournissent ainsi des algorithmes efficaces à garantie théorique. En revanche, ils ne donnent que rarement des solutions intéressantes.

# Solution approchée avec garantie expérimentale

On parle de *garantie expérimentale* dans le cas où un algorithme permet de donner à la fois une solution approchée et une borne sur son écart à l'optimum. C'est le cas pour les algorithmes de branchement-évaluation. En effet, dans le cas maximisation, si l'on dispose d'une solution réalisable (borne inférieure) produite par une méthode heuristique initiale $z_{H_0}$ et d'une évaluation du problème $z_{bs_0}$, on obtient un encadrement de la solution optimale $z_{opt}$ :

$$z_{H_0} \leq z_{opt} \leq z_{bs_0}$$

Au cours de l'algorithme, on découvre de nouvelles solutions réalisables et de meilleure borne (supérieure). En fait, à tout moment, on peut considérer la meilleure solution réalisable rencontrée $z_{bestsol}$ et la plus petite valeur de borne supérieure globale rencontrée $z_{bsg}$. On peut donc considérer tout au long du déroulement de l'algorithme l'augmentation de la borne inférieure et la réduction de la borne supérieure : cela reduit l'écart.



# Solution approchée avec garantie expérimentale

On mesure cet écart en tant qu'écart relatif appelé souvent *gap* :

$$gap = \frac{z_{bsg} - z_{bestsol}}{z_{bestsol}}.$$

Ainsi, dans le pire des cas, la borne supérieure est égale à $z_{opt}$, ce qui signifie que la meilleure solution réalisable trouvée est au plus loin de l'optimum : on aurait ainsi $(1 + gap)z_{bestsol} = z_{opt}$ ce qui signifie qu'au pire $z_{bestsol}$ est à $gap\%$ de l'optimum.

On peut ainsi obtenir à tout moment une garantie expérimentale sur les solution approchées. Si le gap est peut important, par exemple inférieur à 5%, on peut souvent considérer que l'on a obtenu une très bonne solution (éventuellement même optimale !) : industriellement, les valeurs que l'on manipulent en entrée du problème sont parfois imprécises ou soumises à de fortes variations, il n'est pas systématiquement nécessaire d'obtenir une meilleure solution ! Inversement, si le gap est mauvais (supérieur à 5%), cela peut-être dû au fait que la solution approchée trouvée est mauvaise... mais cela peut aussi provenir d'une mauvaise évaluation.

## Résolution exacte

Le contenu de ce module est clairement de présenter des outils permettant la résolution exacte des PMD. En revanche, nous présenterons également comment les tecniques exactes permettent également d'obtenir des valeurs approchées avec garantie lorsque les dimensions des problèmes deviennent trop importantes.

1. Résolution exacte ou à garantie

2. Heuristiques primales ou Matheuristique

3. Algorithmes gloutons

4. Meta-heuristiques

# Heuristique primale (ou Matheuristique)

Si on l'on dispose d'une formulation linéaire en nombres entiers pour un problème, il est alors possible d'utiliser une solution fractionnaire pour déterminer une solution réalisation : on parle alors d'**arrondi** (rounding) ou d'**heuristique primale** ou encore de **Matheuristique**.

Par exemple, prenons la formulation aux arêtes (ou par les cliques) pour le problème du stable maximum où la fonction est de poids 1 pour tout sommet. Considérons la solution fractionnaire $x$ de la relaxation linéaire.
On peut appliquer itérativement l'algorithme d'arrondi suivant : on prend les sommets par ordre de valeur $x$ décroissantes.
Tant qu'il reste un sommet $u \in V$ tel que $x_u$ n'est pas entier, on le met à 1 (ce qui entraîne tous ses voisins à être fixé à 0).
Le vecteur obtenu est alors une solution entière du problème.

Suivant le problème et l'arrondi utilisé, cette méthode ne fournit pas toujours une solution réalisable : il faut alors soit la corriger, soit la refuser. Il est aussi possible de "tirer au sort" un arrondi (random rounding).
On peut également utiliser une méthode métaheuristique à partir d'une solution primale afin de l'améliorer : c'est une méthode très efficace !

1. Résolution exacte ou à garantie

2. Heuristiques primales ou Matheuristique

3. Algorithmes gloutons

4. Meta-heuristiques

# Algorithmes gloutons

Les algorithmes gloutons (greedy algorilhms en anglais) sont des algorithmes pour lesquels, à chaque itération, on fixe la valeur d'une (ou plusieurs) des variables décrivant le problème sans remettre en cause les choix antérieurs.

Le principe est donc de partir d'une solution incomplète (éventuellement totalement indéterminée) que l'on complète de proche en proche en effectuant des choix définitifs : à chaque étape, on traite (on ≪ mange ≫) une partie des variables sur lesquelles on ne revient plus.

Par exemple, l'algorithme de Kruskal pour la recherche d'un arbre couvrant de poids minimum est glouton : on traite successivement les arêtes sans revenir sur les décisions prises à propos des arêtes précédentes. Cet algorithme conduit néanmoins à une solution exacte !

Mais ce n'est que rarement le cas. Pour le problème du voyageur de commerce par exemple, on peut proposer l'algorithme glouton suivant : on part d'un sommet $u$ pris au hasard et on le relie à son plus proche voisin qu'on appelle $u_2$ ; à une itération courante, si on appelle $u_i$ le sommet atteint à l'itération précédente, on repart de $u_i$ et on le relie à son plus proche voisin parmi les sommets non encore rencontrés ; lorsqu'on atteint tous les sommets, on ferme le cycle hamiltonien à l'aide de l'arête $\{u_n, u_1\}$ si on appelle $u_n$ le dernier sommet rencontré. On peut remarquer que la dernière arête utilisée peut être la plus longue du graphe... rendant ainsi la solution très mauvaise.

# Algorithmes gloutons

Si les algorithmes de ce type sont souvent rapides, en revanche la solution qu'ils déterminent peut être arbitrairement loin de la solution. On les utilise néanmoins fréquemment pour obtenir rapidement une solution réalisable. Par exemple, elle servent à initialiser une méthode itérative. Mais dans certains types d'instances réelles, la solution gloutonne est parfois très bonne.

1. Résolution exacte ou à garantie

2. Heuristiques primales ou Matheuristique

3. Algorithmes gloutons

4. Meta-heuristiques
4.1 Méthodes de recherche locale par voisinages
4.2 Les méthodes évolutives

# Metaheuristiques

Si certaines heuristiques sont spécifiques à un problème, d'autres ont pour vocation de pouvoir être adaptées à divers problèmes. On appelle parfois ces dernières des *méta-heuristiques* ou encore heuristiques générales. En fait, il s'agit de principes algorithmiques permettant d'obtenir une solution en respectant certains principes de construction.

Les principales sont les méhodes gloutonnes ; les méthodes de recherche locale (ou d'améliorations itératives) telle que la méthode tabou (ou ses améliorations comme Grasp) ou les méthodes de descentes (recuit simulé) ; les méthodes évolutives (algorithmes génétiques) ; et la simulation (objet ou continue).

Les méta-heuristiques se divisent en général entre :

- heuristique gloutonne : c'est le cadre des heuristiques où l'on ne remet jamais en question une décision prise précédemment. En général, pour les problèmes difficiles (NP-difficiles), ces heuristiques sont peu performantes. Il faut néanmoins noter qu'elles sont souvent les seules que l'on peut utiliser dans les cas où l'on doit traiter des données en temps réel (cas online). D'autre part, ces heuristiques sont parfois optimales (par exemple l'algorithme de Prim pour la recherche d'arbre couvrant).
- méthodes itératives qui tentent de reproduire le mécanisme des méthodes d'optimisation continue. Elles reposent sur la définition de voisinage : c'est-à-dire de déterminer comment passer d'une solution réalisable à une autre solution réalisable "voisine" dans le but d'en déterminer une meilleure. Les plus célèbres sont la descente stochastique, le recuit simulé et la méthode Taboo.
- méthodes évolutives Elles reproduisent des phénomènes biologiques ou physiques où l'on voit apparaître des solutions noouvelles en "mélangeant" des informations provenant d'un lot de solutions réalisables. A chaque itération de cette méthode, on obtient ainsi un nouveau lot de solutions réalisables construits à partir du lot précédent. Les plus célèbres : algorithme génétique, les colonies de fourmis, les essaims de particules...

# Méthodes de recherche locale

Le principe des *méthodes de recherche locale* (ou méthodes d'amélioration itérative) est inspirée des méthodes d'optimisation continue. Ces méthodes consistent à déterminer itérativement la solution d'une fonction continue en utilisant des outils comme les dérivées partielles ou les gradients, suivant que la fonction soit ou non dérivable.

La description de ces méthodes à partir d'une solution de départ $X_0$, à engendrer une suite (finie) de solutions $(X_n)$ déterminées de proche en proche, c'est-à-dire itérativement ($X_{i+1}$ étant déterminée à partir de $X_i$). Le choix de la solution $X_{i+1}$ se fait dans un ensemble "localement proche" de la solution $X_i$ et de manière à avoir une solution $X_{i+1}$ "plus intéressante" au sens de la recherche locale. Les deux expressions mises entre guillemets dans la phrase précédente se formalisent à partir de la notion de *voisinage* et de *recherche dans un voisinage*.

La construction essentielle de ces méthodes reposent en effet sur la détermination d'un bon voisinage. Ensuite, suivant la façon de choisir une solution dan le voisinage, on obtient différentes méthodes de recherche locale : méthode tabou, descente "pure", descente stochastique, recuit simulé,...

# Voisinage d'une solution

On définit généralement le voisinage d'une solution à l'aide d'une transformation élémentaire (ou locale). On appelle *transformation* toute opération permettant de changer une solution $X$ de $S$ en une autre solution $X'$ de $S$. Une transformation sera considérée comme élémentaire (ou locale) si elle ne modifie que "faiblement" la structure de la solution à laquelle on l'applique.

Autrement dit, les transformations locales constituent un sous-ensemble de l'ensemble des transformations. Elles sont locales en ce sens qu'elles ne perturbent pas globalement la structure de la solution qu'elles changent, mais ne la modifient que localement. Par exemple, si X est un entier codé sous la forme d'une chaîne de 0-1, une transformation locale peut consister à changer un élement en 0-1 de la chaîne en son complémentaire.

# Voisinage d'une solution

Le choix de la transformation élémentaire dépend a priori du problème à traiter. On peut soumettre le choix de la transformation élémentaire à deux critères qu'elle devra respecter. Dans la mesure où cette transformation est appliquée de nombreuses fois, on doit d'abord pouvoir en évaluer rapidement les conséquences, ce qui entraîne qu'elle doit être relativement simple (c'est pour cette raison qu'on considère plus volontiers des transformations locales que globales).

D'autre part, elle doit permettre d'engendrer tout l'ensemble $S$, ou du moins tout l'ensemble dans lequel on cherche la solution optimale. C'est la propriété **d'accessibilité**. On veut aussi pouvoir revenir sur ses pas à la solution d'origine : propriété de **réversibilité**. Par exemple, la transformation consistant à changer, dans une chaîne de $n$ 0-1, un 0-1 en son complémentaire permet d'engendrer n'importe quelle configuration en au plus $n$ changements bien choisis à partir de n'importe quelle solution initiale.

# Voisinage d'une solution

La transformation élémentaire étant choisie, on peut définir le voisinage d'une solution. Etant donnée une transformation locale, le *voisinage* $V(X)$ d'une solution $X$ est l'ensemble des solutions que l'on peut obtenir en appliquant à $X$ cette transformation locale.

La notion de voisinage dépend donc de la transformation locale considérée. Ainsi, pour la transformation élémentaire évoquée plus haut dans l'exemple, le voisinage d'une chaîne $X$ de $n$ 0-1 est l'ensemble des $n$ chaînes de $n$ 0-1 possédant exactement $n-1$ 0-1 en commun avec $X$. Une autre transformation pourrait induire un voisinage de cardinal différent. Par exemple, celle définie par le changement simultané de deux 0-1 conduirait à un voisinage possédant $\frac{n(n-1)}{2}$ éléments...
En pratique, on recherche en général des voisinages de tailles réduites dans laquelle l'exploration peut être réalisée en temps polynomial ($O(n), O(n^2)$ au pire $O(n^3)$).
Mais le tout est d'avoir un voisinage bien pensée et efficace.

Un autre problème sous-jacent au voisinage est la façon de coder une solution. En effet, il peut y avoir différentes façon de coder la valeur d'une solution ce qui induit plusieurs techniques de recherche. Par exemple, un tour du TSP peut être codé soit par les arêtes utilisées, soit par la liste des villes dans l'ordre de la visite...

# Quelques idées de voisinages possibles

Pour définir le voisinage, on trouve fréquemment une ou plusieurs des opérations suivantes mises en oeuvre, selon la nature du problème et du codage des solutions :

- **complémentation** (remplacement) : pour les solutions codées sous forme d'une chaîne en 0-1. On a évoqué plus haut cette transformation : on remplace un 0-1 quelconque de la chaîne par son complémentaire. Par exemple,

1 0 0 1 1 1 0 0 1 devient 1 0 0 1 1 0 0 0 1

par complémentation du sixième élément. On peut généraliser cette transformation, lorsque la solution est codée sous la forme d'une chaîne de caractères : en remplaçant un (éventuellement plusieurs) caractère(s) quelconque(s) de la chaîne par un autre caractère (par autant d'autres caractères).

- **échange** : lorsque la solution est codée sous la forme d'une chaîne de caractères, l'échange consiste à intervertir les caractères situés en deux positions données de la chaîne. Ainsi

A B C D E F G devient A E C D B F G

par échange des positions 2 et 5.

# Quelques idées de voisinages possibles

- **insertion-décalage** : supposons de nouveau que la solution est codée sous la forme d'une chaîne de caractères. L'insertion-décalage consiste alors à choisir deux positions $i$ et $j$, à insérer en position $i$ le caractère situé en position $j$ puis à décaler tous les caractères anciennement situés entre $i$ (inclus) et $j$ (exclus) d'un cran à droite si $i < j$, d'un cran à gauche sinon. Par exemple

A B C D E F G devient A B F C D E G

par insertion-décalage avec $i = 3$ et $j = 6$.

- **inversion** : supposons encore que la solution est codée sous la forme d'une chaîne de caractères. L'inversion consiste à choisir deux positions $i$ et $j$ avec $i < j$, puis à inverser l'ordre d'écriture des caractères situés aux positions $i$, $i + 1$,...,$j - 1$, $j$. Par exemple

A B C D E F G devient A E D C B F G

par inversion avec $i = 2$ et $j = 5$.

# Quelques idées de voisinages possibles

Ces transformations locales dépendent d'un (pour la première) ou de deux (pour les autres) paramètres. On peut concevoir des transformations locales dépendant d'un plus grand nombre de paramètres. Pour de nombreux problèmes, les conséquences de ces changements sont faciles à évaluer et la répétition de ces transformations (en les choisissant convenablement) un nombre de fois suffisant permet bien d'engendrer n'importe quelle solution. Ces transformations (ou les codages adoptés pour représenter les solutions) ont parfois un inconvénient, provenant généralement du fait que les configurations ainsi engendrées ne sont pas toujours toutes pertinentes (un exemple étant celui du remplacement d'un caractère par un autre, dans le cas où la chaîne représentant une solution réalisable ne peut admettre plusieurs fois un même caractère).

En pratique, un voisinage doit s'inspirer de la structure même du problème. Il est par exemple important de choisir le bon codage de la solution pour $X$. Par exemple, pour le problème du voyageur de commerce, il est possible de coder la solution par l'ordre des villes visité ou par un vecteur en 0-1 sur les arêtes du graphe...

# Exemple de voisinage : le problème du voyageur de commerce

Pour le problème du voyageur de commerce, il est habituel de considérer comme transformation élémentaire celle qui consiste à choisir deux arêtes non adjacentes dans le cycle hamiltonien (de la solution courante) et de les remplacer par les deux arêtes qui permettent de reconstituer un cycle hamiltonien.

Cette transformation est appelée 2-opt. Elle définit, pour chaque solution, un voisinage de $\frac{n(n-3)}{2}$ éléments. Elle peut en fait être considérée comme un cas particulier de l'inversion évoquée plus haut. On peut la généraliser en envisageant une transformation qu'on pourrait appeler $k$-opt et qui consisterait à remplacer simultanément $k$ arêtes par $k$ autres convenablement choisies.

# Initialisation et réitération des méthodes de recherche locale

Pour appliquer une méthode de recherche locale, il faut une solution de départ. Celle-ci peut être calculée tout à fait aléatoirement, ou bien provenir d'une autre méthode approchée, par exemple d'un algorithme glouton.

Lorsque la solution initiale est (au moins partiellement) aléatoire, on peut alors appliquer plusieurs fois la descente en changeant à chaque fois la configuration de départ, et ne conserver finalement que la meilleure solution rencontrée depuis la première descente. Cette répétition permet alors d'atténuer l'inconvénient majeur de la recherche locale, c'est-à-dire d'être locale.

Il est également possible de concevoir une méthode gloutonne qui compléterait une un morceau réalisable d'une solution. Par exemple, dans le cas d'un vecteur à $n$ 0-1, on ne conserverait que quelques valeurs des 0-1 et on s'autoriserait à remplacer les autres de manière gloutonnes. Ainsi, on peut utiliser le schéma suivant appelé GRASP (Greedy Random Adaptative Procedure), initié par Feo et Resende en 1989.
GRASP est composée de 2 étapes : une étape de construction gloutonne, suivie par une étape de recherche locale. On réitère ensuite ces 2 étapes en utilisant à pour le départ de l'une la solution courante de la fin de l'autre. Cette idée permet de cumuler les avantages de plusieures méthodes.

# La méthode Tabou

Même si les premières idées concernant la méthode Tabou datent peut-être de 1977, on peut plus sûrement la faire remonter aux alentours de 1986. Elle a été proposée indépendamment par F. Glover d'une part, par P. Hansen puis P. Hansen et B. Jaumard d'autre part (sous un nom différent).

L'idée de départ est simple. Elle consiste à se déplacer de solution en solution en s'interdisant de revenir en une configuration déjà rencontrée. Plus précisément, supposons qu'on a défini un voisinage $V(X)$ pour chaque solution $X$. Supposons en outre qu'on dispose à toute itération de la liste $T$ de toutes les configurations rencontrées depuis le début de l'exécution de la méthode. Alors, à partir de la configuration courante $X$, on choisit dans $V(X) \setminus T$ la solution $X'$ qui minimise la fonction $H$, puis on ajoutant $X'$ à la liste $T$. Autrement dit, on choisit parmi les configurations voisines de $X$ mais non encore rencontrées celle qui "descend" le plus fortement si $X$ n'est pas un minimum local (par rapport à la transformation élémentaire qui définit le voisinage), ou celle qui "remonte" le moins sinon, et on ajoute $X'$ à $T$ pour s'interdire d'y revenir.

# La liste Tabou

En fait, il est rarement possible de pouvoir mettre en oeuvre ce principe : conserver toutes les configurations rencontrées consomme en général trop de place mémoire et trop de temps de calcul pour savoir quelle configuration choisir dans le voisinage de la solution courante, puisqu'il faut comparer chaque voisin à chaque élément de la liste $T$.

D'un autre côté, supposons qu'on passe de $X$ à $Y$ avec $Y \in V(X)$ et $H(Y) > H(X)$, il alors existe en $Y$ au moins une direction de descente dans son voisinage, celle qui redescend en $X$ ! Et on risque ainsi de boucler si on ne s'interdit pas, au moins provisoirement, un retour en $X$.

Pour éviter ces inconvénients sans être obligée de conserver en mémoire toutes les configurations rencontrées depuis le début, la méthode Tabou préconise la tactique suivante. Au lieu d'ajouter la solution courante $X$ à la liste $T$ des configurations interdites (taboues), on se contente, quand on remonte, de conserver en mémoire la transformation élémentaire qui a permis de passer de la configuration courante à la suivante et on s'interdit d'appliquer son inverse : ce sont donc désormais des mouvements qui sont tabous et non plus des configurations.

# La liste Tabou

Cette modification présente à son tour certains inconvénients. En interdisant des mouvements, on s'interdit aussi d'aller vers certaines solutions qui pourraient être intéressante. Pour ne pas trop appauvrir le voisinge de la onfiguration courante, on limite la taille de la liste $T$ que l'on gère comme une file (premier entré, premier sorti). En pratique, on choisit souvent une taille assez petite, qu'il faut décider en la paramétrant. Suivant les cas, on préconise autour de la 10aine, de la centaine, rarement plus. Une taille plus petite risque de ne pas pouvoir empêcher le bouclage, et une taille plus grande semble en général trop appauvrir les voisinages.

Il est nécessaire de prévoir un critère d'arrêt : par exemple un nombre d'itérations que l'on s'autorise à effectuer, où la stagnation de la meilleure valeur trouvée depuis un certain nombre d'itérations...

# La liste Tabou (Améliorations)

Il existe peu de paramètres à fixer sur une méthode tabou (à part la taille de la liste). Mais comme l'on explore systématiquement tout un voisinage, ce voisinage joue un rôle majeur.
De nombreuses variantes peuvent être ajoutées à cette méthode : manipuler plusieurs listes Tabou, interdire toute transformation inverse, associer à chaque transformation élémentaire un compteur indiquant pendant combien d'itérations elle est taboue,...

On peut également s'autoriser à passer outre le caractère tabou d'un mouvement dans certains cas. On appelle ce procédé *l'aspiration* : on lui donne un paramètre $m$ qui indique au bout de combien d'itération on s'autorise à passer à une solution taboue mais qui est intéressante par rapport à la valeur courante. Il existe d'autres techniques intéressantes pour améliorer la puissance de la méthode tabou, en particulier, l'intensification et la diversification. Toutes les deux se basent sur l'utilisation d'une mémoire à long terme et se différencient selon la façon d'exploiter les informations de cette mémoire.

# La liste Tabou (Améliorations)

*L'intensification* se fonde sur l'idée d'apprentissage de propriétés favorables : les propriétés communes souvent rencontrées dans les meilleurs configurations visitées sont mémorisées au cours de la recherche, puis favorisées pendant la période d'intensification. Une autre manière d'appliquer l'intensification consiste à mémoriser une liste de solutions de bonne qualité et à retourner vers une des ces solutions.

La *diversification* a un objectif inverse de l'intensification : elle cherche à diriger la recherche vers des zones inexplorées. Sa mise en oeuvre consiste souvent à modifier temporairement la fonction de coût pour favoriser des mouvements n'ayant pas été effectués ou à pénaliser les mouvements ayant été souvent répétés.

L'intensification et la diversification jouent donc un rôle complémentaire.

# Les méthodes de descente

L'idée générale d'une méthode de descente est de toujours prendre dans un voisinage une solution meilleure que la solution courante. En ce qui concerne l'exploration du voisinage de la solution courante, plusieurs, attitudes peuvent être adoptées : exploration aléatoire, exploration systématique pour déterminer un voisin meilleur, ou exploration exhaustive pour déterminer le meilleur voisin.

# Descente "simple" et descente stochastique

La méthode de descente simple est inspiré de la minimisation de fonctions continues, la descente simple consiste tout simplement à choisir systématiquement un sommet du voisinage qui améliore le plus la solution courante. L'algorithme s'arrête donc quand il n'est plus possible d'améliorer la solution.
Cette descente simple n'est intéressante que dans le cas où le voisinage est suffisemment petit, car il faut systématiquement explorer le voisinage.

L'exploration stochastique (aléatoire) consiste à choisir aléatoirement une solution voisine, puis de tester si elle améliore la solution courante :

Initialiser une valeur réalisable pour $X$
Tant que nécessaire faire
    Choisir aléatoirement et uniformément $Y$ dans $V(X)$
    si $H(Y)$ ¡ $H(X)$ alors $X \leftarrow Y$

Cette descente aléatoire évite de visiter systématiquement un voisinage qui serait trop grand.

Le critère d'arrêt peut être un nombre d'itérations sans amélioration jugé suffisant, ou l'obtention d'une solution acceptable. Le défaut de ces méthodes de descentes pures est de rester dans un minimum local sans trouver le minimum global.

# Descentes itérées

Une méthode très simple et très efficace est de relancer plusieurs fois la méthode de descente stochastique à partir de la même solution gloutonne ou de plusieurs solutions gloutonnes. On appelle cette méthode **méthodes de descentes itéreées**.

# Algoritme du recuit simulé

Le recuit simulé (ou méthode de Monte-Carlo) provient de la physique moléculaire où des molécules se positionnent de façon à minimiser leur énergie quand la température baisse (c'est le principe de frappe des métaux en ferronerie : chauffer, puis refroidir avant de modeler le métal). L'algorithme de principe est le suivant où $RND$ est une fonction qui donne un échantillon uniforme et indépendant dans $[0, 1]$ :

$T$ est une "température"
$X$ est l'état initial,
$RX \leftarrow X$ (état record)
Tant que nécessaire faire
    Choisir aléatoirement et uniformément $Y \in V(X)$
    si $H(Y) < H(RX)$ alors $RX \leftarrow Y$
    si $H(Y) < H(X)$ alors $X \leftarrow Y$
    sinon
        si $RND \leq exp(-\frac{H(Y)-H(X)}{T})$ alors $X \leftarrow Y$
    Générer une nouvelle température $T$

A chaque itération, on prend un voisin au hasard. Si ce voisin est meilleur, il devient le nouvel état courant. Sinon, il devient le nouvel état courant avec une certaine probabilité qui dépend à la fois de la différence des coûts et de la température courante. Pour cette loi de probabilité, on utilise souvent cette fonctoon appelée "dynamique de Metropolis", mais d'autres fonctions sont possibles.

# Algoritme du recuit simulé

La température de départ doit être suffisamment élevée pour permettre d'accepter régulièrement des mauvaises transitions au début de l'algorithme. Cette température décroît progressivement vers 0 durant le déroulement de l'algorithme, diminuant ainsi la probabilité d'accepter des transitions défavorables.
Généralement, l'algorithme doit s'arrêter quand l'état est "gelé". C'est-à-dire que l'état courant n'est plus modifié (la température est trop basse pour accepter de mauvaises transitions).
En pratique, on peut décider d'une baisse de la température par pallier réguliers où la température s'abaisse dès qu'un objectif est atteint (nombre d'itérations, valeurs, stagnation,...).

Si la température baisse suffisamment lentement, et si le système de voisinage vérifie certaines propriétés, le recuit converge en probabilité vers l'optimum. En effet, Hajek (1988) a énoncé le théorème suivant :
Théorème de convergence (Hajek 1988) : Si le système de voisinage vérifie les propriétés d'accessibilité et de réversibilité, et si à partir d'un certain rang $n$, $T \geq \frac{D}{ln(n+1)}$ où $D$ représente la profondeur maximale d'un minimum local, alors le recuit simulé converge en probabilité vers l'ensemble des solutions optimales.
Malheureusement, il est difficile (voire impossible) en pratique de vérifier ces hypothèses.

# Les méthodes évolutives

Le terme "algorithmes évolutifs" englobe une autre classe assez large de métaheuristiques. Ces algorithmes sont basés sur le principe du processus d'évolution naturelle. Les algorithmes évolutifs doivent leur nom à l'analogie entre leur déroulement et le mécanisme de sélection naturelle et de croisement des individus d'une population vivante sexuée.

Un algorithme évolutif typique est composé de trois éléments essentiels :
1) une population constituée de plusieurs individus représentant des solutions potentielles (configurations) du problème donné.
2) un mécanisme d'évaluation de l'adaptation de chaque individu de la population à l'égard de son environnement extérieur
3) un mécanisme d'évolution composé d'opérateurs permettant d'éliminer certains individus et de produire de nouveaux individus à partir des individus sélectionnés.

# Les méthodes évolutives

Du point de vue opérationnel, un algorithme évolutif débute avec une population initiale souvent générée aléatoirement et répète ensuite un cycle d'évolution suivant les principes en 3 étapes séquentielles :
- Evaluation : mesurer l'adaptation (la qualité) de chaque individu de la population
- Sélection : sélectionner une partie des individus
- Reproduction : produire de nouveaux individus par des recombinaisons d'individus sélectionnés

Ce processus se termine quand la condition d'arrêt est vérifiée, par exemple, quand un nombre de cycles (générations) ou quand un nombre d'évaluations est atteint ou quand des solutions suffisamment bonnes sont trouvées. Si l'on s'imagine que le processus suit le principe d'évolution naturelle, la qualité des individus de la population doit s'améliorer au fur et à mesure du processus.

# Les méthodes évolutives

Parmi les composantes d'un algorithme évolutif, la population et la fonction d'adaptation correspondent respectivement à la notion de configuration et à la fonction d'évaluation dans la recherche locale. La notion de mécanisme d'évolution est proche de celle du mécanisme de parcours du voisinage de la recherche locale mais les opérateurs sont sensiblement différents. En effet, un algorithme évolutif comporte un ensemble d'opérateurs tels que la sélection, la mutation et éventuellement le croisement.
La *sélection* a pour objectif de choisir les individus qui vont pouvoir survivre ou/et se reproduire pour transmettre leurs caractéristiques à la génération suivante. La sélection se base généralement sur le principe de conservation des individus les mieux adaptés et d'élimination des moins adaptés.
Le *croisement* ou recombinaison cherche à combiner les caractéristiques des individus parents pour créer des individus enfants avec de nouvelles potentialités dans la génération future.
La *mutation* effectue de légères modifications de certains individus.

# Cadre des algorithmes génétiques

Suivant l'imagination des ses utilisateurs, on appelle ces méthodes "algorithmes génétiques", "programmation évolutive", "stratégies d'évolution", "essaim de particules", "colonie de fourmis",... Nous prendrons ici comme exemple, largement suffisant, des algorithmes dits "génétiques". Notons qu'il s'agit ici d'une présentation combinatoire de ces algorithmes qui ont aussi une version en optimisation continue.

Les algorithmes génétiques peuvent se définir à partir d'un codage sous forme de chaînes 0/1 de longueur fixe et un ensemble d'opérateurs "génétiques" : la mutation, l'inversion et le croisement Un individu sous ce codage est un chromosome, un gène la composante de base du chromosome et un allèle la valeur effective d'un gène (0 ou 1 ici). En d'autres termes, un chromosome, un gène et un allèle représentent respectivement une solution (configuration), un attribut de la solution et la valeur de l'attribut.

# Cadre des algorithmes génétiques

Les opérateurs génétiques sont définis de manière à opérer stochastiquement sur le codage sans aucune connaissance sur le problème. Par exemple, le croisement "bi-points" consiste à choisir aléatoirement deux points de croisement et à échanger les segments des deux parents déterminés par ces deux points. La mutation consiste simplement à changer aléatoirement la valeur de certains gènes. Le rôle de la mutation dans les algorithmes génétiques est essentiellement de réintroduire de nouvelles valeurs pour des gènes alors que le croisement réalise uniquement des recombinaisons de valeurs existantes.
Un cycle d'évolution complet d'un algorithme génétique est formé par l'application des opérateurs de sélection, croisement et mutation sur une population de chromosomes. L'universalité d'un tel algorithme pose des problèmes d'efficacité en pratique. En effet, en tant que méthode d'optimisation, un algorithme génétique basé sur des opérateurs génétiques "aveugles" est rarement en mesure de produire des résultats comparables à ceux de la recherche locale. Une technique pour remédier à ce problème consiste à spécialiser l'algorithme génétique au problème donnée. Plus précisément, à la place des opérateurs aléatoires, la mutation et le croisement sont adaptés en se basant sur des connaissances spécifiques du problème. De cette manière, la recherche est mieux guidée et donc plus efficace.

# Mise en œuvre des algorithmes génétiques

Le choix du codage est important et souvent délicat. L'objectif est bien sûr d'abord de pouvoir coder n'importe quelle solution. Mais il est souhaitable, au-delà de cette exigence, d'imaginer soit un codage tel que toute chaîne de caractères représente bien une solution réalisable du problème, soit un codage qui facilite ensuite la conception du croisement de telle sorte que les "enfants" obtenus à partir de la recombinaison de leurs "parents" puissent être associés à des solutions réalisables, au moins pour un grand nombre d'entre eux.

L'idée majeure à conserver dans un algorithme génétique, en opposition aux méthodes de descentes, est de travailler plus sur la structure des solutions que sur leurs valeurs (on parle parfois de "schéma"). C'est-à-dire qu'il faudrait pouvoir conserver une bonne structure au travers des individus : typiquement un morceau de solutions donnant une bonne valeur. D'autre part, le codage devrait être tel qu'une petite variation dans le chromosome n'entraîne pas une trop grande variation dans la configuration associée à ce chromosome. Ainsi, une représentation en binaire des entiers ne s'accorde pas bien à ce principe.

En général, il n'est pas facile de construire un codage essayant de tenir compte de tous ces critères, si bien qu'on est parfois amené à ne pas prendre en. considération certaines de ces indications.

# La sélection

La sélection des individus sur lesquels on va appliquer le croisement fait intervenir la fonction à minimiser : la probabilité de choisir l'individu $X$ sera d'autant plus grande que $H(X)$ sera faible. Une façon habituelle de faire intervenir sont de répartir les individus/solutions $X_1, X_2, ..., X_p$ selon des catégories et des tirages au sort.
La probabilité de sélectionner $X$, est proportionnelle à $1 - \frac{H(X_i)}{F}$, avec $F = \sum_{i=1}^{p} H(X_i)$ et vaut donc $\frac{F - H(X_i)}{F(p-1)}$. Ce qui donne à chaque individu sa chance d'être sélectionné. Attention il faut des valeurs de $H$ positives. Il faut souvent adapter cette idée en fonction des fonctions $H$.
Le but de la sélection est de savoir quels individus sont conservés et lesquels serviront à la reproduction.

# Le croisement

L'objectif du croisement est de recombiner d'une certaine façon les chromosomes de deux (rarement plus) parents procréateurs afin de former les chromosomes d'un ou de deux (rarement plus) enfants. Le croisement s'inspire du mécanisme observé dans le crossing-over de la génétique (et est d'ailleurs aussi appelé ainsi) : on extrait une partie du code associé à chacun des parents, et on réorganise ces parties entre elles de façon à former de nouveaux individus qui jouent le rôle des enfants.
Un croisement que l'on rencontre souvent est le ≪ croisement à un point ≫. Afin de décrire celui-ci, imaginons que l'on décide de croiser deux individus $A$ et $B$ (les parents) représentés chacun par un chromosome, respectivement $C_A$ et $C_B$. Le croisement à un point consiste à déterminer aléatoirement une position (un gène) après laquelle on coupe $C_A$ et $C_B$ : on obtient donc quatre morceaux de chromosomes $C_A^1$ et $C_A^2$ issus de $C_A$, et $C_B^1$ et $C_B^2$ issus de $C_B$. On peut alors facilement former deux nouveaux individus $D$ et $E$ (les enfants), obtenus par un échange des morceaux de chromosomes : $D$ aura pour chromosome la concaténation de $C_A^1$ et de $C_B^2$, et $E$ aura pour chromosome celle de $C_B^1$ et de $C_A^2$.
Ce type de croisement fait que les couples dé gènes n'ont pas tous la même probabilité de rester ensemble (ainsi les extrémités d'un chromosome seront-elles systématiquement séparées), ce qui n'est pas toujours souhaitable. On peut alors, modifier ce croisement en un ≪ croisement à deux points ≫ (et plus généralement à $k$ points)...

# La mutation

Le dernier opérateur, la mutation, est conçu pour apporter une certaine diversité dans la population et empêcher que celle-ci ne converge trop vite vers un même individu, ou vers un petit groupe d'individus, ce qui rendrait inopérant le croisement (si du moins celui-ci conserve les caractéristiques des procréateurs). Il agit aléatoirement sur le codage d'un individu, en remplaçant un ou plusieurs des symboles du codage par autant d'autres symboles de l'alphabet. Par exemple dans le cas d'un entier codé en binaire, la mutation peut être de changer un 1 en 0 ou un 0 en 1. Pour des problèmes de position, la mutation consistera à modifier localement la solution associée à l'individu à muter par une opération appropriée.
Par exemple, pour le voyageur de commerce, on pourra considérer comme mutation une transformation du type 2-opt (c'est-à-dire un échange en croix). Plus généralement, on pourra s'inspirer des transformations élémentaires que sont les voisinages d'une solution réalisable. Attention, pour ne pas trop perturber la composition de la population et ne pas transformer les algorithmes génétiques en une errance aléatoire, la mutation doit avoir une faible probabilité $p$,d'être appliquée. On pourra, selon les cas, envisager au plus une mutation par chromosome ou au contraire envisager éventuellement une mutation par gène. La valeur de probabilité devra être choisie en conséquence (plus faible dans le second cas que dans le premier).

# Autres opérateurs

D'autres opérateurs sont possibles. Par exemple, celui qui consiste à améliorer séparément les individus de la population, souvent à l'aide d'une méthode d'amélioration itérative (mais on pourrait évidemment envisager de lui substituer une méthode plus sophistiquée, comme le recuit simulé ou la méthode Tabou). Dans leur conception originelle, les algorithmes génétiques n'intègrent pas cette idée qui modifie sensiblement la philosophie de la méthode : le croisement peut alors paraître réduit à un moyen d'échapper à un minimum local que risquerait d'atteindre la méthode d'amélioration itérative sans pouvoir en sortir. On peut éventuellement interpréter cette phase d'amélioration individuelle comme une sorte d'apprentissage permettant à l'individu de mieux s'adapter à l'environnement (que traduit la fonction à optimiser).