Chapitre 4

Méthodes d’optimisation approchées

La plupart des problèmes NP-difficiles étudiés en Optimisation Combinatoire sont simples à formuler (Problème du Voyageur de Commerce, Satisfiabilité,…). Ces problèmes peuvent être résolus, du moins en théorie, de manière triviale, rien qu'en énumérant toutes les solutions possibles et en choisissant la meilleure. Ceci est le principe même de la plupart des méthodes exactes. En pratique, cette approche n'est applicable que pour des instances de problèmes de petite taille.
Du point de vue informatique, le temps de résolution d'un problème NP-difficile est prohibitif. C'est pourquoi les chercheurs se sont intéressés ces dernières années aux méthodes approchées. Au lieu de chercher la solution optimale d'un problème, il est souvent préférable de trouver une solution approximative (une bonne solution dont le coût est proche de celui da solution optimale) calculable en un temps raisonnable.
Le recourt aux méthodes approchées telles que les heuristiques s'avère de plus en plus intéressant. En effet, dans beaucoup de cas pratiques, disposer d'une solution optimale n'est pas primordial et une solution d'assez bonne qualité trouvée en un temps réduit est jugée plus intéressante qu'une solution optimale trouvée en un temps beaucoup plus long.
Avant de présenter quelques méthodes d'optimisation, nous rappelons des définitions relatives aux problèmes d'optimisation.

- **La fonction objectif,** appelée aussi **fonction d'évaluation**, notée généralement ***f,*** sert à évaluer le coût d'une solution d'un problème donné. A titre d'exemple, pour le PCG, ***f*** peut être définie comme le nombre de couleurs utilisées ou bien le nombre de conflits générés.
- Un problème d'optimisation combinatoire consiste à chercher le minimum (resp. le maximum) *s** d'une fonction *f* (fonction objectif) à valeur entière ou réelle, sur un ensemble fini *S*. S étant l'ensemble des solutions réalisables du problème d'optimisation.

$$f(s^*) = \min_{s \in S} \{ f(s) \} \quad (resp. = \max \{ f(s) \})$$

- **Le voisinage de S** qu'on note généralement ***V (S)*** est l'ensemble des solutions générées à partir d'une solution courante S suivant une stratégie donnée.

1/ Méthodes approchées

L'application des méthodes exactes devient rapidement inenvisageable vu le temps d'exécution qui croit exponentiellement avec l'augmentation de la taille du problème. Il est souvent indispensable de trouver des solutions, même approchées, en un temps raisonnable. Les méthodes approchées, répondent à ce besoin en permettant de trouver des solutions acceptables, parfois même optimales, en des temps relativement courts. Les méthodes de résolution approchées suivent trois approches différentes pour la recherche d'une solution: l'approche de construction, l'approche de voisinage et l'approche d'évolution. Ces méthodes sont fondées principalement sur diverses heuristiques, souvent spécifiques à un type de problème.
Les métaheuristiques constituent une autre partie importante et plus générale des méthodes approchées et ouvrent des voies très intéressantes en matière de conception de méthodes de résolution pour l'optimisation combinatoire. Nous présentons dans ce qui suit le principe de ces méthodes en mettant l'accent sur les métaheuristiques.
Une heuristique est une procédure qui exploite au mieux la structure d'un problème à optimiser dans le but de trouver une solution raisonnable (non nécessairement optimale) en un

temps réduit. Chaque heuristique respecte donc les contraintes du problème considéré. Les performances d'une heurisitque sont liées à la qualité de la solution produite ainsi qu'au temps de calcul nécessaire pour l'obtenir. Selon la stratégie de recherche de solution, on distingue deux types d'heuristiques: constructives et de voisinage:

1.1 Les méthodes constructives

Les méthodes constructives partent d'une solution initiale vide S_0, et insèrent à chaque étape k une composante x_k dans la solution partielle courante $S_{k-1=}(x_0, \ldots, x_{k-1})$ tout en respectant les contraintes du problème, pour aboutir enfin à une solution admissible de la forme $S_{=}(x_0, x_1, \ldots, x_n)$.

Les deux avantages de ces méthodes sont leur facilité de mise en œuvre et leur rapidité d'exécution. Par contre, la faible qualité des solutions trouvées est en général leur grand défaut. En effet, ces méthodes ne tiennent pas compte de l'effet du choix de la composante à insérer sur les choix futur et sur la qualité de la solution finale.

Le PCG étant un problème NP-difficile, il n'y a donc pas d'algorithme efficace pour le résoudre en un temps raisonnable pour une taille importante du problème. En revanche, plusieurs heuristiques sont proposées pour trouver des solutions approximatives, réduisant considérablement le temps d'exécution. Nous présentons dans ce qui suit les principales méthodes constructives pour colorier un graphe G:

- **LF (Largest First) :** proposée en 1967 par Welsh et Powell, les sommets du graphe à colorier sont triés par ordre décroissant de leur degré
- **SL (Smallest Last) :** Proposé par Matula et al en 1972, les sommets sont triés selon l'ordre croissant de leur degré.
- **DSatur :** Proposée par Brélaz en 1979, cet algorithme ordonne les sommets de G à colorier par ordre décroissant de leur degré de saturation (ld degré de saturation de *x* est le nombre de couleurs déjà utilisées par les voisins de x), puis selon leur degré en cas d'égalité.
- **RLF (Récursive Largest First) :** Proposé par Leighton en 1979. Les classes de couleurs sont construites les unes après les autres. Apres avoir choisi un sommet de degré maximum, l'ensemble stable courant est complété en rajoutant aussi longtemps que possible un sommet de X-B de degré maximum où B contient les sommets non colorés ne pouvant pas appartenir au stable en cours de construction.

L'utilisation des algorithmes LF et SL se révèle intéressante dans les cas où une solution approximative suffit, mais où le temps de calcul est primordial. Si par contre la qualité de la solution prime sur la rapidité, plusieurs expériences ont montré que Dsatur et RLF sont plus performants que les autres. Nous présentons dans ce qui suit le principe de ces deux dernières méthodes.

Algorithme DSatur : C'est une méthode gloutonne. DSatur est inspirée des méthodes séquentielles pour la résolution du problème de coloration. Son principe consiste à :

- Colorer le sommet de degré maximal avec la couleur 1.
- A chaque itération i, prendre le sommet non coloré de degré de saturation maximal et lui affecter la plus petite couleur possible qui ne crée pas de conflit (dsatur (i)). Le principe de cette méthode est résumé dans la figure 1.

Algorithme Dsatur
Début
Initialisation :
X: Ensemble de sommets du graphe G ;
NC : ensemble de sommets de Graphe G non colorés ;
DS_Max() : //fonction qui retourne le sommet ayant le degré de saturation maximum dans NC ;
NBC le nombre de couleurs ;
Poser NBC=0 ;
Tant Qu'*il existe des sommets non encore colorés,* ***Faire***
x= DS_Max() ;
Colorer v avec la plus petite couleur possible ;
NBC=NBC+1 ;
Fin Tant Que
Fin.

Figure1 : Algorithme Dsatur.

DSatur peut être vue comme un cas particulier de méthodes séquentielles où l'ordre de choix des sommets est tel que les sommets colorés forment à chaque itération un sous-graphe connexe. Bien que DSatur soit très bon en moyenne, sa complexité est $O(n^2)$.

1.2 Les méthodes de voisinage

Une méthode typique de voisinage est un processus itératif fondé sur deux éléments essentiels : un voisinage et une procédure exploitant le voisinage. Avant de présenter quelques algorithmes de cette famille, nous donnons le principe général de ces méthodes.

Stratégie de génération des solutions voisines : C'est une transformation élémentaire appliquée aux éléments d'une solution S pour générer le voisinage V(S) de cette solution. Par exemple, on peut inverser l'ordre de deux villes pour le problème du voyageur de commerce, ou inverser 0 et 1 pour les problèmes dont les solutions sont codées en binaires, ou encore déplacer un sommet d'une partition à une autre (changer la couleur d'un sommet) pour le problème de coloration des graphes**.**

Principe d'une méthode de voisinage

Une méthode de voisinage débute avec une solution initiale (générée aléatoirement ou par l'application d'une heuristique), et réalise ensuite un processus itératif qui consiste à remplacer la configuration courante (solution courante) par l'un de ses voisins en tenant compte de la fonction coût (fig. 2). Ce processus s'arrête et retourne la meilleure configuration trouvée quand la condition d'arrêt est satisfaite. Cette dernière concerne généralement un nombre d'itérations ou un objectif à réaliser.

Un des avantages des méthodes de voisinage réside précisément dans la possibilité de contrôler le temps de calcul. La qualité de la solution trouvée tend à s'améliorer progressivement au cours du temps et l'utilisateur est libre d'arrêter l'exécution au moment qu'il aura choisi.

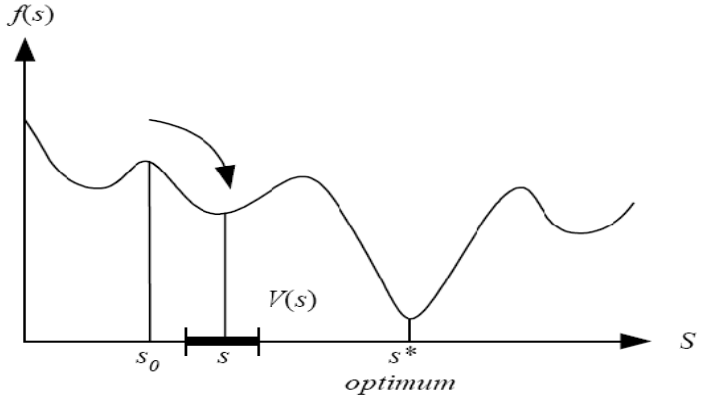

Figure 2 : Processus de recherche dans une méthode de voisinage.

Une méthode de voisinage typique est la Recherche Locale (RL) que nous décrivons ci-dessous :

La **Recherche Locale (RL)**, appelée aussi méthode de descente, est une méthode itérative qui explore l'espace de recherche en partant d'une solution initiale S_0 et en se déplaçant pas à pas d'une solution à une solution voisine dans le but d'améliorer la solution courante (fig. 3). Son principe est résumé de la manière suivante :

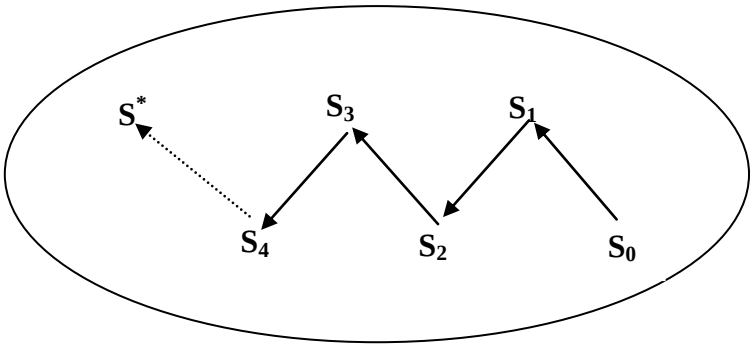

Figure 3 : Exploration par la recherche locale.

La recherche locale reçoit en entrée une solution initiale générée aléatoirement, ou, comme c'est souvent le cas, par une méthode constructive. A chaque itération, la *RL* cherche une solution meilleure que la solution courante dans le voisinage de cette dernière.

Si une telle solution existe alors celle-ci remplacera la solution courante et on passe à une autre itération. Sinon, la *RL* cherche une solution de même coût que la solution courante et relance la recherche à partir de cette nouvelle solution. Cette opération permet une légère diversification de la recherche en ce sens qu'elle permet d'explorer d'autres voies à partir d'autres solutions de même coût que la solution courante. Si toutefois, la recherche stagne après génération de S_{min} solutions de même coût, le processus de recherche s'arrête. La *RL* s'arrête aussi, naturellement, quand une solution optimale a été trouvée.

La figure suivante donne le pseudo code d'un algorithme de RL pour le PCG:

Algorithme Recherche Locale (S, f, S_{min})
Tant Que *(f(S) >0)*
- Modifier S en changeant l'élément le moins récemment modifié et qui améliore f(S), s'il existe un tel élément.
- ***Si*** *un tel élément n'existe pas* ***Alors*** *Modifier S en changeant l'élément le moins récemment modifié et qui ne change pas la valeur de f(S).*
- ***Si*** *après Smin modifications successives, on n'arrive pas à améliorer S, ou s'il n'existe plus de solution meilleure ou de même coût que S dans son voisinage,* ***Alors***
Retourner S
Fin Tant Que
Retourner S
Fin

Figure 4 : Pseudo algorithme de la RL.

Malgré sa grande simplicité et sa rapidité, la recherche locale simple présente l'inconvénient de retourner souvent des solutions de qualité médiocre. En effet, elle se retrouve souvent piégée dans un optimum local.

2/ Les métaheuristiques

Les heuristiques sont souvent dédiées au problème traité. Elles ne peuvent pas être généralisées car chacune se base sur des caractéristiques spécifiques au problème considéré. Les chercheurs se sont donc intéressés à des méthodes qui se basent sur des concepts plus généraux, ne prenant pas forcément en considération la nature du problème. Cela a donné naissance à ce qu'on appelle aujourd'hui les *Métaheuristiques*.

Les métaheuristiques forment une famille de méthodes d'optimisation visant à résoudre des problèmes d'optimisation difficile souvent issus des domaines de la recherche opérationnelle ou de l'intelligence artificielle pour lesquels on ne connaît pas de méthode exacte efficace. Grâce à ces méthodes, on peut proposer aujourd'hui des solutions approchées pour des problèmes d'optimisation classiques de plus grande taille et pour de très nombreuses applications qui étaient impossible d'être traitées auparavant. Les métaheuristiques regroupent essentiellement deux classes de méthodes.
1. Les méthodes de voisinage ou itératives manipulant une seule solution telles que le recuit simulé et la recherche tabou;
2. Les méthodes évolutives qui manipulent une population (de solutions) et se basent en général sur l'évolution naturelle ou sur l'observation des comportements sociaux des individus ou des communautés. On retrouve dans cette classe les algorithmes génétiques, les méthodes basées sur les colonies de fourmis, les essaims d'abeilles et les systèmes immunitaires.

2.1 Les méthaheuristiques manipulant une seule solution

Ce sont des algorithmes qui font évoluer une seule solution sur l'espace de recherche à chaque itération. La notion de voisinage est alors primordiale. Les méthodes les plus connues (et les plus utilisées aussi) dans cette classe, sont la *Recheche Tabou* et le *Recuit Simulé*.

2.1.1 Recherche tabou

La recherche Tabou (*RT*) est une métaheuristique originalement développée par Glover et indépendamment par Hansen en 1986, sous l'appellation de *steepest ascent mildest descent*. Elle est basée sur des idées simples, mais très efficaces. Cette méthode combine une procédure de recherche locale avec un certain nombre de règles et de mécanismes permettant à celle ci de s'échapper des optima locaux, tout en évitant de cycler en 1993. Pour ce faire, la *RT* fait explicitement usage d'un transfert de connaissance entre des étapes successives du processus de recherche pour bien guider celle-ci. Elle a été appliquée avec succès pour résoudre de nombreux problèmes difficiles d'optimisation combinatoire notamment les problèmes de coloration des graphes.

Principe de la recherche tabou

Etant donné une solution initiale S (générée aléatoirement ou par l'application d'une heuristique constructive), la *RT* cherche une meilleure solution S* dans le voisinage V(S), par une stratégie de sélection donnée, en interdisant le retour à S dans les « *t* » prochaines itérations (*t* préalablement définie). En effet, la méthode tabou s'appuie sur un principe qui consiste à garder en mémoire dans une liste appelée *liste Tabou*, les dernières solutions visitées et à interdire le retour vers celles ci pour un nombre fixé d'itérations afin d'éviter le retour vers une solution déjà visitée. Le but de cette stratégie est de donner assez de temps à l'algorithme pour sortir d'un minimum local. La *RT* guide intelligemment sa recherche et ne reste pas bloquée dans un optimum local en acceptant les détériorations dans le voisinage de S. En effet, la meilleure solution de l'itération courante sera prise même si elle est moins bonne que la solution globale. La *RT* a trois caractéristiques fondamentales:

1. A chaque itération, le voisinage **V(s)** de la solution actuelle **s** est examiné, et une solution **s'** y est sélectionnée**,** même si son coût n'améliore pas celui de la solution courante.
2. On s'interdit de revenir sur une solution visitée dans un passé proche grâce à une liste taboue **T** de longueur **NT**, qui stocke les **NT** dernières solutions visitées.
s' est donc cherchée dans **V(s) – T**.
3. la meilleure solution trouvée s* à chaque itération est conservée. Le processus de recherche s'arrête après un nombre maximal **nMax** d'itérations fixé au préalable, ou après un nombre maximal d'itérations sans améliorer la meilleure solution, ou bien quand **V(s) – T = Ø.** Ce dernier cas, très rare, ne peut se produire que sur de très petits problèmes, pour lesquels le voisinage tout entier peut être enfermé dans **T**.

Paramètres de la Recherche Tabou

Le voisinage : A chaque itération, il faut rechercher s' le voisin optimal de s, appelé aussi le successeur de s (celui dont la fonction objectif a un coût minimal (problème de minimisation)). De manière plus formelle: $s' \in V(s) / \forall s_i \in V(s), f(s') \leq f(s_i)$

Pour sélectionner une solution dans le voisinage, il existe plusieurs stratégies :
Best move : Cette stratégie consiste à choisir la meilleure solution dans tout le voisinage
First improve : On choisit la première solution qui améliore la fonction objectif.
Stratégie aléatoire : On choisit aléatoirement une solution parmi celles du voisinage.
Cependant, un problème évident se pose : lorsqu'on choisit le successeur de **s** une configuration **s'** qui dégrade **s**, il y a de fortes chances que **s∈V (s')**, et que **s** soit le successeur de **s' (**voisin optimal). Ce qui induit immédiatement un cyle **s ↔ s'** qui se traduit par une chaîne de **s** vers **s',** puis de **s'** vers **s.** L'algorithme dans ce cas risque de boucler. Pour remédier à ce problème, la *RT* utilise une liste taboue.

- ***La liste Tabou*** **(T):** C'est une mémoire flexible qui garde une trace des dernières opérations passées. On peut y stocker des informations pertinentes à certaines étapes de la recherche pour en profiter ultérieurement. **T** fonctionne donc comme une mémoire à court terme. A chaque itération, la plus ancienne solution de **T** est écrasée par la dernière solution examinée. En pratique, **T** se gère simplement comme une structure de données de type file. Cette liste peut être statique (de taille fixe) ou dynamique (de taille variable), et permet d'empêcher les blocages dans les minima locaux en interdisant de passer à nouveau sur des configurations précédemment visitées (problème de cycle).On peut distinguer deux types de listes tabou:

Liste tabou explicite : elle sauvegarde des solutions complètes.
Liste tabou attributive **:** elle sauvegarde seulement les informations à propos des solutions qui changent à travers la recherche. Pour le PCG par exemple on peut sauvegarder le couple (sommet et sa nouvelle couleur).
La taille de la liste Tabou est une donnée primordiale à déterminer empiriquement et qui varie en fonction du problème à traiter. En effet une liste trop petite peut conduire à un cycle, alors qu'une liste trop grande peut empêcher des transformations intéressantes. Glover en 1993 a constaté qu'une liste taboue de taille $|T| = 7$ à 20 suffit en pratique pour empêcher l'algorithme de boucler en revenant sur une solution déjà visitée. De son coté, Taillard (1990) en appliquant la RT au problème de l'affectation quadratique, a observé qu'en faisant varier la taille de la liste Tabou tout au long du processus de recherche permet d'obtenir de bons résultats.
En théorie, il faut stocker les $|T|$ dernières solutions visitées « complètes ». Les voisinages considérés étant souvent grands, il est clair que le test répétitif de présence dans T est très coûteux, sans parler de la mémoire nécessaire pour stocker les $|T|$ solutions complètes. Le stockage explicite des solutions est donc souvent évité. En général, on sauvegarde dans T les mouvements ayant permis de changer des solutions ou certains attributs des solutions. Ensuite, on interdit les mouvements inverses ou les solutions ayant les mêmes attributs. Une technique encore plus simple pour gérer la liste taboue est d'interdire de repasser par les $|T|$ dernières valeurs de la fonction objectif. Dans ce cas, il suffit de stocker uniquement les coûts des $|T|$ dernières solutions. Les résultats peuvent être satisfaisants, sauf si la fonction objectif prend relativement peu de valeurs différentes.
Un des algorithmes de RT qui ont été appliquée avec succès au PCG est TabuCol établit par Hertz et Werra en 1987. La figure 5 présente le principe de cet algorithme.

Algorithme TabuCol
Début
Générer une solution initiale S_0 par un algorithme constructif . Puis l'insérer dans la liste Tabou ;
Poser $S^*=S_0$ et LT=Ø;
Définir t la taille de LT et initialiser la fonction d'aspiration Asp. ;
Tant qu*'aucun critère d'arrêt n'est atteint faire*
Générer un échantillon de p paramètres de V(S) ;
Sélectionner la meilleure solution S'
($S \in V(S)$ ET $S \notin$ liste tabou);
Si *$F(S) < F(S^*)$* ***Alors***
Poser $S^* \leftarrow S$;
Sinon *f(S)= 0* ***Alors*** *stop;*
Fin Si
Poser S=S' ;
Introduire (v,i)dans LT;Si|LT|>t Alors retirer de LT le couple le plus ancien;
Mettre à jour de la fonction d'aspiration Asp ;
Fin Tant que
Fin

Figure 5 : Algorithme TabuCol.

Amélioration de la Recherche Tabou

- **Critère d'aspiration**

En utilisant une liste Tabou non explicite qui contient des transformations élémentaires, on interdit non seulement de revenir vers des solutions précédentes mais aussi vers un ensemble de solutions dont plusieurs peuvent ne pas avoir été visitées jusqu'ici. Le risque est alors de négliger des solutions potentielles. Il est donc nécessaire de remédier à ce problème et de trouver un moyen de lever l'interdiction d'une transformation élémentaire déjà effectuée. Ceci est réalisable en utilisant ce qu'on appelle le *critère d'aspiration*. Cette correction permet aussi de revenir à une solution déjà visitée et de redémarrer la recherche dans une autre direction. Le critère d'aspiration le plus simple et le plus couramment utilisé consiste à tester si la solution produite, de statut tabou, présente un coût inférieur à celui de la meilleure solution trouvée jusqu'à présent. Si cette situation se produit, le statut tabou de la solution est levé. Ce critère est évidemment très sévère, il n'est donc pas vérifié très souvent, par conséquent, il apporte peu de changements à la méthode. D'autres critères d'aspiration plus complexes peuvent être envisagés. L'inconvénient de recourir trop souvent à l'aspiration est qu'elle peut détruire, dans une certaine mesure, la protection offerte par la liste taboue.

- **Intensification**

L'intensification consiste à approfondir la recherche dans certaines régions du domaine, identifiées comme susceptibles de contenir un optimum global. Cette intensification est appliquée périodiquement, et pour un certain nombre de générations. Pour mieux intensifier la recherche dans une zone bien localisée, plusieurs stratégies sont proposées.
La plus simple consiste à retourner à l'une des meilleures solutions trouvée jusqu'à présent, en réduisant la longueur de la liste taboue pour un nombre limité d'itérations. Dans ce cas, on adapte la procédure de recherche taboue, en élargissant le voisinage de la solution courante tout en diminuant le pas des transformations.

- **Diversification**

La diversification permet à l'algorithme de bien explorer l'espace des solutions, et d'éviter que le processus de recherche soit trop localisé en laissant de grandes régions du domaine totalement inexplorées. La plus simple des stratégies de diversification consiste à interrompre périodiquement l'acheminement normal de la procédure taboue, et à redémarrer la recherche à partir d'une autre solution, choisie aléatoirement.
Une autre stratégie consiste à biaiser la fonction d'évaluation, en introduisant un terme qui pénalise les transformations effectuées fréquemment, afin d'en favoriser de nouvelles. Ce type de stratégies de diversification peut être utilisé de façon continue, sans interrompre la procédure de recherche taboue.
En résumé, la diversification et l'intensification sont des concepts complémentaires, qui enrichissent la méthode de recherche taboue et la rendent plus robuste et plus efficace. Ces deux principes peuvent être combinées: En premier lieu, on permet d'intensifier la recherche dans une région donnée, puis la solution trouvée va servir dans la deuxième étape à diversifier la recherche vers une région non visité (fig. 6).

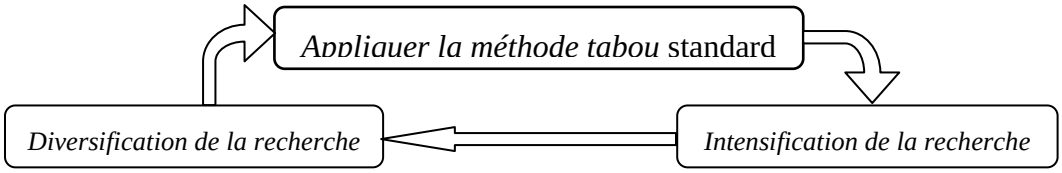

Figure 6 : Stratégies d'utilisation de la diversification et l'intensification.

2.1.2 Le Recuit simulé

Le ***Recuit Simulé*** (*RS*) est une méthode d'optimisation proposée en 1983 par Kirkpatrick pour la résolution d'un problème de placement en VLSI (*Very Large Scale Intégration*). Le RS est inspiré du principe de la thermodynamique. Son processus de recherche est assimilé au processus du recuit physique. Quand un métal est chauffe à une température très élevée, il devient liquide et peut prendre toute forme. Quand la température décroît, le métal se fige peu à peu dans une configuration qui est de plus en plus difficile à déformer (refroidissement). Pour retravailler de nouveau le métal, et lui donner la forme désirée, il faut le réchauffer (recuit).

En optimisation, le processus du recuit simulé est une procédure itérative qui cherche des configurations de coût plus faible tout en acceptant de manière contrôlée des configurations qui dégradent la fonction de coût. A chaque nouvelle itération, un voisin s $\in V(s_i)$ de la configuration courante s est généré de manière aléatoire. Selon les cas, ce voisin sera soit retenu pour remplacer s_i, soit rejeté. Si ce voisin est meilleur que la solution courante; i.e., si $F(s) \leq F(s_i)$, il est systématiquement retenu. Dans le cas contraire, s est accepté avec une probabilité $p(\Delta F, T)$ qui dépend de deux facteurs : un paramètre de contrôle, la température *T*, (une température élevée correspond à une probabilité plus grande d'accepter des dégradations), et l'importance de la dégradation $\Delta F = F(s) - F(s_i)$ (les dégradations plus faibles sont plus facilement acceptées). La température est contrôlée par une fonction décroissante « $e^{(-\Delta F /T)}$ » qui définit un schéma de refroidissement. Les deux paramètres de la méthode définissent la longueur des paliers et la fonction permettant de calculer la suite décroissante des températures. En pratique, l'algorithme s'arrête et retourne la meilleure configuration trouvée lorsqu'aucune configuration voisine n'a été acceptée pendant un certain nombre d'itérations à une température ou lorsque la température atteint un seuil au dessous d'un certain niveau.

Notons que le rôle du paramètre T est déterminant pour le choix de la solution courante. En effet, à haute température ΔF /T tend vers 0, cela implique que $e^{(-\Delta F /T)}$ tend vers 1,donc la plupart des solutions trouvées sont acceptées et ainsi le processus de recherche est pratiquement aléatoire. Cependant, à basse température, ΔF /T est très grand, cela implique que $e^{(-\Delta F /T)}$ tend vers 0, donc la plupart des solutions augmentant la fonction coût sont systématiquement rejetées. La performance du *RS* dépend fortement du schéma de refroidissement utilisé. Plusieurs schémas de refroidissement ont été proposés en pratique et en théorie.

Il existe des schémas qui garantissent la convergence asymptotique du recuit simulé (en un temps infini).

Le *RS* constitue une des plus anciennes méthodes de voisinage. Elle a acquis son succès essentiellement grâce à des résultats pratiques obtenus sur de nombreux problèmes d'optimisations et grâce aussi à la preuve de convergence bien que celle-ci n'ait aucun impact dans la pratique. En ce qui concerne le PCG, le *RS* ne s'avère pas très efficace en comparaison avec d'autres méthodes comme la *RT* surtout pour les graphes de grande taille.

Pour résoudre un problème d'optimisation combinatoire par la méthode du Recuit Simulé, il est nécessaire de bien spécifier les paramètres suivants:

La solution Initiale : La solution initiale est générée soit aléatoirement, ou avec une méthode constructive.

La température initiale: Par analogie avec le recuit physique, Kirpatrick et al (1983), suggèrent que la valeur de la température initiale T_0 soit assez élevée de sorte que la probabilité d'acceptation de mauvaises solutions soit environ 80%. Le changement de la

température est donné par la formule suivante $T_{t+1} = \alpha T_t$, où α est en pratique une constante comprise entre 0.8 et 0.99.

Le nombre d'itérations pour chaque température: Un nombre d'itérations à effectuer à chaque température est un paramètre à fixer au préalable.

Le critère d'arrêt de l'algorithme: Quand il n y a aucune amélioration de la fonction objectif pour une température donnée ou une borne minimale de la température est atteinte, l'algorithme s'arrête.

La stratégie du voisinage: Le voisinage de toute solution doit être défini ainsi que les méthodes le manipulant à savoir, la stratégie de génération de voisinage et celle de sélection d'une solution dans le voisinage.

La figure 7 présente l'algorithme du *RS* appliqué au PCG (2002):

Algorithme Recuit Simulé
Début
Initialisation i = 0; T = Température initiale
générer une coloration initiale s,
poser s^*=s, choisir T (valeur initial du paramètre température) ;
Choisir $\alpha \in (0,1)$ (facteur de diminution de T)
Tant que** critère d'arrêt non satisfait, **faire
***répéter** R fois (R paramètre prédefini)*
choisir aléatoirement s' dans V(s)(une coloration voisine est obtenue en déplaçant un sommet vers une autre partion)
***si** F(s')≤F(s), poser s=s'*
***sinon** (F(s') > F(s))*
- générer aléatoirement un réel r entre 0 et 1
- si $r < e^{F(s)-F(s')/T}$, poser s=s'
Fin si
Fin répéter
poser T=α.T (diminution de la température)
i = i+1
Fin Tant que
Fin

Figure 7 : Algorithme du recuit simulé appliqué au PCG.

2.2 Méthodes manipulant une population

Cette classe de méthodes d'optimisations regroupe toutes les métaheuristiques évolutives. Contrairement aux méthodes, constructives et de voisinage, qui font intervenir une solution unique (partielle ou non). Les méthodes évolutives manipulent un groupe de solutions à chacune des étapes du processus de recherche. L'idée générale consiste à utiliser régulièrement les propriétés collectives d'un ensemble de solutions distinctes, appelé population, dans le but de guider efficacement la recherche vers de bonnes solutions dans l'espace de recherche. Les méthodes évolutives constituent la base d'un vaste champ de la programmation informatique en pleine expansion.

Ces méthodes sont basées sur le principe du processus d'évolution naturelle. Les algorithmes évolutifs doivent leur nom à l'analogie avec les mécanismes d'évolution des espèces vivantes. Un algorithme évolutif typique est composé de trois éléments essentiels:

1. ***Une population*** constituée de plusieurs individus représentant des solutions potentielles du problème donné.
2. ***Une fonction 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.

Du point de vue opérationnel, un algorithme évolutif typique débute avec une population initiale souvent générée aléatoirement et répète ensuite un cycle d'évolution composé de trois étapes séquentielles:

1. Mesurer l'adaptation (la qualité) de chaque individu de la population par la fonction d'évaluation.
2. Sélectionner une partie des individus (les parents).
3. Produire de nouveaux individus (les enfants) par des recombinaisons d'individus sélectionnés.

Selon l'analogie avec l'évolution naturelle, la qualité des individus de la population devrait tendre à s'améliorer au fur et à mesure du processus. Ce processus se termine quand une condition d'arrêt est vérifiée, qui consiste souvent en un nombre maximum de cycles (générations) ou un maximum de temps atteint.

Un algorithme évolutif comporte un ensemble d'opérateurs qui consiste le plus souvent en la sélection, la mutation et éventuellement le croisement.

a/ La sélection consiste à choisir les paires d'individus qui vont participer à la reproduction de la population future. La fonction de sélection calcule une probabilité de sélection pour chaque individu, en fonction de sa fitness (qualité) par rapport à la qualité de tous les autres individus dans la population.

b/ Le croisement est le principal opérateur agissant sur la population de parents. Il est appliqué avec une certaine probabilité, appelée taux de croisement Pc (typiquement proche de l'unité). Le croisement consiste en général, à choisir deux individus représentés par leurs chaînes de gènes, tirés au hasard dans la population courante, et à définir aléatoirement un ou plusieurs points de croisement. Les nouvelles chaînes sont alors créées en inter-changeant les différentes parties de chaque chaîne.

c/ La mutation protège les algorithmes évolutifs des pertes prématurées d'informations pertinentes. Elle permet d'introduire une certaine information dans la population, qui a pu être perdue lors de l'opération de croisement. Elle participe ainsi au maintien de la diversité, utile à une bonne exploration du domaine de recherche.

2.2.1 Les Algorithmes génétiques

Les Algorithmes Génétiques (AG) sont des méthodes qui s'inspirent des mécanismes de sélection naturelle et des phénomènes d'évolution génétiques. Ils ont été introduits par John Holland pour la première fois au début des années soixante à l'Université du Michigan en 1975. Les algorithmes génétiques sont très performants dans leurs principes de recherche et d'amélioration. De plus, ils ne sont pas limités par des hypothèses sur le domaine d'exploration de l'espace de recherche et la problématique. En outre, comme toute autre méthode évolutive, les AG risquent moins d'être piégés dans des minima locaux du fait qu'ils manipulent en parallèle un ensemble de solutions possibles d'un problème donné. Ils utilisent un ensemble d'opérateurs génétiques (croisement, mutation, et remplacement). Ces algorithmes utilisent un vocabulaire similaire à celui de la génétique et de la biologie. Nous introduisons ci-dessous quelques définitions relatives aux AG.

1. L'individu (chromosome)

Dans les AG, l'individu est une solution du problème codée. Les algorithmes génétiques s'appuient fortement sur un codage universel sous forme de chaînes 0/1 de longueur fixe, mais peut aussi utiliser un autre codage selon la nature du problème à optimiser. L'ensemble des individus forme une population ou génération. Learman et Ngouenet (2000) conseillent de

prendre comme taille de la population la valeur correspondante à la longueur du codage des individus. En effet, si la taille de la population est très grande, l'évaluation de tous les individus de la population peut s'avérer trop longue. Par contre, si, elle est très petite, l'algorithme peut converger trop rapidement vers un optimum local.
Une codification possible d'une solution d'un problème de coloration de graphe dans le cas de l'approche d'assignation, est donnée dans la figure 8. Un chromosome est constitué de n cases. La i ème case contient comme information le numéro de la couleur affectée au i ème sommet.

Figure 8 : représentation chromosomique d'une solution d'un PCG (Approche d'assignation).

Une deuxième codification possible, appliquée au problème de k-coloration (k fixe) par Galinier et Hao (1999), considère que chaque chromosome est composé de k cases contenant chacune une liste de sommets formant un stable (Approches par partition). La figure suivante (fig.9) illustre un chromosome utilisant cette codification (on considère la même coloration que celle de l'exemple précédent):

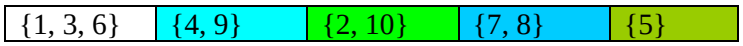

Figure 1 : Représentation chromosomique d'une solution d'un PCG (Approche de partition).

2. La fonction d'évaluation (fitness)

Elle attribue à chaque individu un coût qui permet de le juger apte ou non à se reproduire.

3. Les opérateurs génétiques

Les opérateurs génétiques définissent la manière avec laquelle, les individus se combinent pendant la phase de reproduction, pour générer de nouvelles solutions.

- **L'opérateur de croisement (Crossover) *:*** Son principe est de prendre deux individus aléatoirement et de couper chacun d'eux en un ou plusieurs points, ensuite, inter changer les gènes situés entres les points considérés. Cet opérateur est appliqué avec une probabilité ***Pc***. Plus ce taux est élevé, plus la population subit des changements importants. Il existe plusieurs types de croisement :

  ***i. Le croisement à un point* :** Une position de coupure $k \in \{1,..., n\}$ dans un individu (parent) est généré aléatoirement. L'information mémorisée dans les composantes i> k des deux parents est interchangée.

  ii. Le croisement à deux points (bipoints) : Dans ce cas, on génère deux solutions enfants en échangeant l'information des parents située entre deux points de coupure sélectionnés aléatoirement.

  iii. Le croisement à N points : C'est une généralisation du croisement bi-points. Les individus sont coupés en N points; si une partie est échangée entre deux individus, la partie suivante ne le sera pas et ainsi de suite.

  iv. Le croisement uniforme : Dans ce cas un masque de croisement (de taille $\leq n$) est généré aléatoirement, où un bit du masque qui vaut 1 à la $N^{ième}$ position désigne le bit à échanger entre le couple d'individus à la même position. La figure suivante (fig.10) illustre ces trois types de croisement.

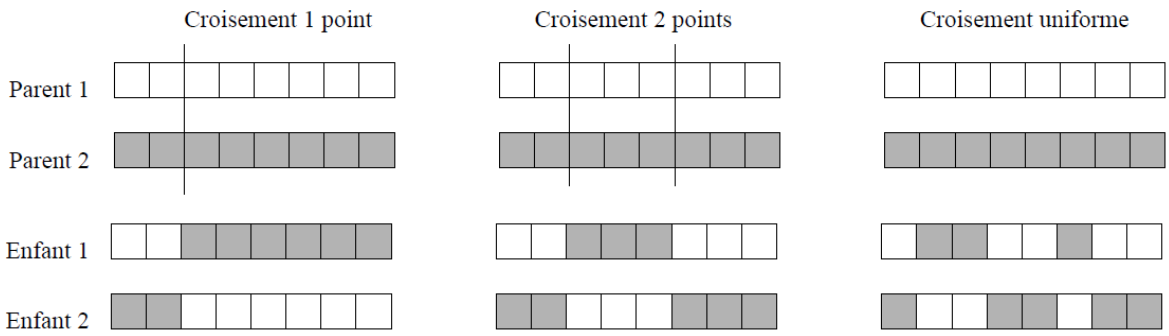

Figure 2 : Illustration des trois principaux types de croisement.

- **L'opérateur de mutation :** Il consiste à choisir un ou deux bits aléatoirement, puis les inverser. L'opérateur de mutation s'applique avec une certaine probabilité *Pm*, appelée taux de mutation souvent relatif à la taille de la population.

D'après plusieurs études expérimentales, *Pm* est compris entre 0.05 et 0.10. Ce faible taux de mutation est dû au fait que la mutation est considérée comme un mécanisme d'adaptation secondaire pour les algorithmes génétiques.

- **La sélection :** Elle sélectionne certaines solutions pour former la population intermédiaire afin de lui appliquer les opérateurs de croisement et de mutation. Chaque élément de la population est sélectionné avec une probabilité selon sa valeur d'adaptation. Ainsi les individus ayant une grande valeur d'adaptation, ont plus de chance d'êtres sélectionnés pour la génération suivante. Il existe plusieurs stratégies de sélection comme: La sélection par la roue de loterie biaisée, ordonnancement (Ranking), la sélection par tournoi ou la sélection par la roue de loterie inverse.
- **L'opérateur de remplacement :** Après le croisement et la mutation, de nouveaux individus sont introduits dans la population, ce qui augmente la taille de celle-ci. L'opérateur de remplacement est alors appliqué pour garder une taille constante de la population. Il existe différentes stratégies de remplacement:
  - Les parents sont remplacés par les enfants,
  - les enfants remplacent les individus les moins forts de la population courante,
  - la nouvelle génération est constituée des meilleurs individus de la population,
  - à chaque génération les deux meilleurs individus entre les enfants et les parents sont gardés.

Critère d'arrêt : Le processus de recherche dans un algorithme génétique s'arrête, si l'un des deux critères d'arrêt suivants est vérifiés : un nombre d'itérations prédéfini est atteint, ou bien lorsque la population cesse d'évoluer ou évolue très lentement. Le pseudo code d'un algorithme génétique standard est donné dans la figure suivante.

```
Algorithme AG
  Début
    Génération d'une population initiale
    Tant que critère d'arrêt non atteint faire
        Sélection
        Croisement
        Mutation
        Mise à jour de la population
    Fin Tant que
  Fin
```

Figure 3 : Algorithme Génétique.

Les algorithmes génétiques ont relativement peu de fondements théoriques. Il n'existe aucune garantie que la méthode converge vers la solution optimale. En effet, il est difficile de faire l'analyse d'un algorithme génétique dans sa globalité, mais en revanche, on peut s'intéresser à l'un de ses opérateurs séparément
Le principal avantage des algorithmes génétiques est leur rapidité. Cependant, le fait qu'ils se basent sur des opérateurs génétiques n'utilisant aucune connaissance du problème traité, a pour effet de limiter considérablement leur efficacité. Il est possible de remédier à cet inconvénient en adaptant les opérateurs de croisement et de mutation au problème traité. Ainsi, ces opérateurs ne seront plus aléatoires, mais basés sur des connaissances spécifiques au problème. Le processus de recherche est alors mieux guidé, et donc plus efficace.
En résumé, les algorithmes génétiques sont des techniques en pleine mutation, et sont toujours en pleine expansion. Ils offrent une alternative intéressante aux techniques classiques de la résolution des problèmes d'optimisation. Leur indépendance vis-à-vis du problème à résoudre facilite leur application sur un large éventail de problèmes.

2.2.2 La Recherche Dispersée

La Recherche Dispersée (RD) a été proposée par Glover en 1997, dans le cadre de la résolution des programmes mathématiques en nombres entiers. La RD s'inspire du principe des algorithmes génétiques et les combine à plusieurs techniques de la Recherche Tabou. Cependant, cette technique n'a pas connu un succès comparable à celui des algorithmes génétiques. Ceci peut s'expliquer par le fait que la méthode n'a pas été présentée sous des traits aussi aguichants. Pourtant, la recherche dispersée est assez similaire aux algorithmes génétiques, du moins si l'on considère les implantations les plus récentes qui ont démontré de nombreux avantages pratiques de cette approche pour résoudre divers problèmes d'optimisation. Il n'est donc pas exagéré de prétendre que cette technique était très moderne pour son temps, même si elle n'a pas été appréciée à sa juste valeur.
Tout comme les algorithmes génétiques, la RD est basée sur une population de solutions qui évolue dans le temps à l'aide à la fois d'un opérateur de sélection, de combinaison linéaire de solutions (pour créer une nouvelle solution provisoire), d'un opérateur de projection permettant de rendre la solution provisoire admissible et d'opérateurs d'élimination. Ainsi, on peut voir la recherche dispersée comme un algorithme génétique présentant les particularités suivantes :

1. L'opérateur de sélection peut élire plus de deux solutions,
2. l'opérateur de croisement est remplacé par une combinaison linéaire,
3. l'opérateur de mutation est remplacé par un opérateur de réparation ou de projection qui ramène la solution nouvellement créée dans l'espace des solutions admissibles ou bien par un opérateur d'amélioration.

Ces particularités peuvent également être vues comme des généralisations des algorithmes génétiques qui ont été proposées et exploitées ultérieurement par divers auteurs.
La RD se compose de deux phases : une phase d'initialisation et une phase évolutive :

- **Phase d'initialisation**

Elle est elle-même composée de plusieurs étapes, au bout desquelles est générée une population élite dite aussi, ensemble de référence.

1. Générer un ensemble de solutions initiales
2. Faire appel au générateur de diversification pour produire une collection de solutions diversifiées, en utilisant une solution arbitraire de la population comme solution initiale.
3. Répéter l'exécution des deux étapes précédentes jusqu'à atteindre un nombre voulu (prédéfini) de solutions qui constitueront la population initiale.

4. Mettre à jour la population en choisissant un sous-ensemble de cette population pour former la population élite, et un autre sous-ensemble pour former la population diversifiée.

- **Phase d'évolution**

Elle permet de faire évoluer la population produite de la phase précédente dans le but d'obtenir de meilleurs individus. Cette phase est aussi composée de plusieurs étapes :

5. produire des sous-ensembles de solutions déjà existantes, lesquelles seront destinées à créer les solutions combinées.
6. combiner des solutions générées par l'étape précédente pour produire une ou plusieurs solutions. La figure 12 illustre un exemple de combinaison où l'ensemble de la population originale est représenté par les points A, B et C. Après une combinaison linéaire des solutions A et B, le point 1 est crée. En effet, plusieurs solutions dans le segment de la ligne définie par les points A et B ont été crées. Cependant, seulement la solution 1 est sélectionnée. De la même façon, la combinaison des solutions de la population (les originales et les nouvelles) crée les solutions 2, 3 et 4.
7. Pour chaque solution combinée, appliquer une méthode d'amélioration pour transformer une solution de faible valeur d'adaptation en une ou plusieurs solution(s) plus intéressantes(s). Puis mettre à jour la population.
8. Si aucune solution n'est produite à l'étape 6, la population est réinitialisée avec la population élite, et on reprend à partir de l'étape 2.
9. Répéter l'exécution des étapes 5-8 jusqu'à satisfaire un critère d'arrêt.

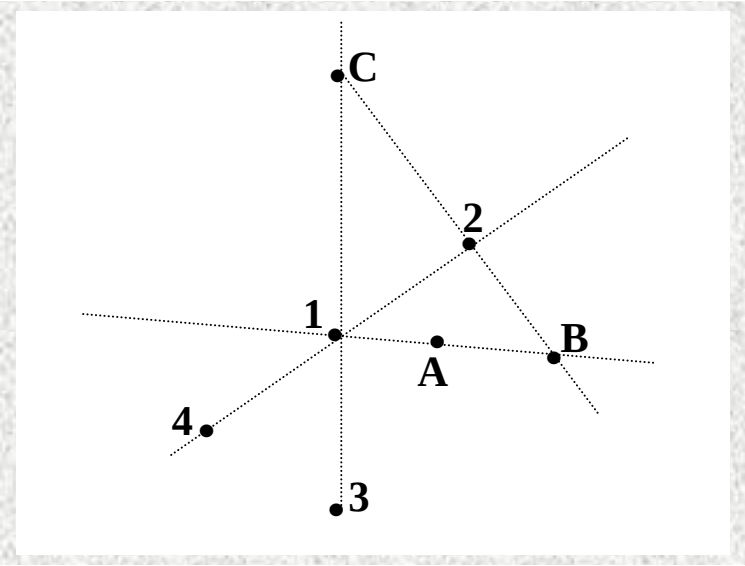

Figure 4 : Exemple de combinaison dans la RD.

2.3 Approches hybrides

Une autre façon d'améliorer les performances d'une approche de résolution d'un problème d'optimisation, ou de combler certaines de ses lacunes consiste à la combiner avec une autre méthode. Plusieurs recherches ont été faites dans le but de combiner les différents avantages des algorithmes de l'optimisation combinatoire. En effet, la mise en commun des principes de plusieurs métaheuristiques ou de méthodes exactes a permis d'élaborer de méthodes d'optimisation plus performantes, appelées méthodes hybrides. Le but d'une méthode hybride est donc de faire coopérer différentes approches afin d'être efficace pour un plus large panel d'instances.

Il existe plusieurs formes d'hybridation. On peut exécuter en parallèle ou de manière séquentielle, diverses métaheuristiques, voire même plusieurs fois la même métaheuristique mais avec divers paramètres. Les méthodes communiquent entre elles périodiquement pour échanger des informations sur leurs résultats partiels. Une autre forme d'hybridation consiste à combiner les métaheuristiques avec des méthodes exactes. Une métaheuristique (généralement une méthode de recherche locale) peut par exemple fournir des bornes à une méthode de type Branch and Bound pour améliorer les solutions partielles à certains points de choix de l'arbre de recherche, ou pour explorer un ensemble de solutions proches d'une solution obtenue de manière gloutonne dans l'arbre de recherche. D'autre part, une méthode exacte peut donner lieu à une technique efficace pour la détermination du meilleur voisin d'une solution. Cette dernière forme d'hybridation s'exprime souvent par une recherche locale qui utilise des mécanismes complets pour rendre les voisinages étendus plus intéressants. Dans ce qui suit, nous présentons quelques formes d'hybridation les plus utilisées.

2.3.1 Hybridation algorithme génétique – recherche locale

Un cas particulier de l'hybridation entre deux méthodes consiste à combiner un algorithme génétique et une méthode de recherche locale. En effet, les méthodes de recherche locale remplacent la configuration courante par une autre configuration voisine. L'opérateur de mutation des algorithmes évolutionnaires effectue lui aussi des modifications légères sur la configuration sélectionnée. Le principe utilisé est le suivant : à chaque génération, un nombre de croisements (opérateur de croisement spécifique au problème, ou aléatoire) est effectuée. Pour chaque nouvelle configuration (individu) ainsi générée, un opérateur de recherche locale est appliqué pour améliorer sa qualité. Finalement, un mécanisme de sélection décide si le nouvel individu amélioré doit être introduit dans la population. Un cas particulier de ce type d'hybridation est donné dans le paragraphe suivant.

2.3.2 Hybridation algorithme génétique – RT : Algorithme génétique hybride

Cette hybridation est basée sur les principes de fonctionnement de la recherche taboue et de l'algorithme génétique. L'algorithme dispose d'un principe de codage des individus, d'un mécanisme de génération de la population initiale et d'opérateurs permettant de diversifier la population au cours des générations et d'explorer l'espace de recherche. La RT est utilisée pour améliorer un nouvel individu généré à chaque itération. L'algorithme est décrit comme suit :
Après la phase de croisement, l'opérateur de mutation est appliqué à une solution choisie aléatoirement (parmi les solutions générées par le croisement) avec une probabilité *pm* pour obtenir un nouvel individu.
Une variante de la procédure de RT est ensuite lancée, comme opérateur d'amélioration locale, à l'autre enfant généré par l'opérateur de croisement non choisi par l'étape précédente pour la mutation. Un nouvel individu est ainsi obtenu.
Une technique d'élitisme est utilisée afin d'assurer que le meilleur individu actuel est toujours gardé dans la mémoire de l'algorithme. Autrement dit, un individu de la population courante, dont la valeur d'adaptation est la meilleure, va sûrement être choisi comme un individu de la

prochaine génération. Un algorithme génétique hybride pour le problème de coloration des graphes « HCA » est proposé par Galinier et Hao en 1999. HCA a donné de très bons résultats et est donc considéré comme un algorithme de référence pour la résolution du PCG.

2.3.3 Hybridation recherche aléatoire – méthodes de voisinage (GRASP)

La méthode GRASP (*Greedy Randomized Adaptive Search Procedure*) est une procédure itérative introduite par Feo et Resende en 1989. GRASP est une méthode hybride qui combine les avantages des heuristiques gloutonnes, de la recherche aléatoire et des méthodes de voisinage. Un algorithme GRASP répète un processus composé de deux étapes, la construction d'une solution suivie par une descente pour améliorer la solution construite.
Durant la première étape, une solution est itérativement construite, à chaque itération un élément est ajouté dans la solution partielle courante. Pour déterminer l'élément qui sera ajouté, une liste des meilleurs candidats obtenus avec une fonction gloutonne est utilisée. Un élément est pris aléatoirement dans cette liste. Cette dernière est dynamiquement mise à jour après chaque itération de construction.
Cette étape de construction continue jusqu'à ce qu'une solution complète soit obtenue. Enfin, une descente est appliquée à partir de cette solution dans le but de l'améliorer.
Une procédure GRASP répète ces deux étapes et retourne à la fin la meilleure solution trouvée. Les deux paramètres de cette méthode sont donc la longueur de la liste de candidats et le nombre d'itérations autorisées. Cette méthode a été utilisée pour bon nombre de problèmes et a donné de bons résultats.

2.3.4 Hybridation méthodes exactes – recherche locale

Les méthodes exactes sont souvent utilisées pour obtenir des solutions optimales et effectuer des preuves d'optimalité. De plus, il a été observé, qu'en raison de leur caractère systématique d'exploration de l'espace de recherche, les solutions intermédiaires qu'elles produisent sont souvent de piètre qualité.
D'autre part, grâce à leur façon opportuniste d'explorer l'espace de recherche, les méthodes approchées, basées essentiellement sur la recherche locale, sont supposées produire de bonnes solutions en un temps de calcul raisonnable.
Malheureusement, de telles méthodes ne sont pas toujours capables de sortir facilement d'optima locaux et peuvent perdre beaucoup de temps dans l'exploration de voisinages inintéressants. Un tel comportement est prohibitif, puisque la qualité des solutions doit s'améliorer graduellement, et le plus rapidement possible, au cours du temps. De plus, la recherche locale nécessite une expertise conséquente afin d'ajuster les paramètres initiaux. En effet, son efficacité dépend beaucoup de ces réglages, qui eux même dépendent de l'instance du problème à résoudre.

Conclusion

Nous avons présenté dans ce chapitre les différentes approches de résolution des problèmes d'optimisation combinatoire. Nous nous sommes étalés plus particulièrement sur les métaheuristiques qui sont des méthodes approchées destinées aux problèmes NP-difficiles. Elles ont prouvé leur efficacité face à de nombreux problèmes d'optimisation combinatoires.
Les métaheuristiques regroupent essentiellement deux classes de méthodes : Les méthodes de voisinage ou itératives manipulant une seule solution et les méthodes évolutives qui manipulent une population. On retrouve dans cette dernière classe toutes les méthodes qui se basent en général sur l'évolution naturelle ou sur l'observation des comportements sociaux des individus ou des communautés.
L'idée générale des métaheuristiques évolutives consiste à utiliser régulièrement les propriétés collectives d'un ensemble de solutions distinctes, appelé population, dans le but de guider efficacement la recherche vers de bonnes solutions dans l'espace de recherche. Les méthodes évolutives constituent la base d'un vaste champ de la programmation informatique en plein expansion.