République Algérienne Démocratique et Populaire
وزارة التعـلـيـم العـالـــي والبـحـــث العـلـمـــي
Ministère de l'Enseignement Supérieur et de la Recherche Scientifique
UNIVERSITE DES SCIENCES ET DE LA TECHNOLOGIE d'ORAN Mohamed Boudiaf

Faculté des Sciences
Département de L'Inforamatique
Spécialité : Informatique *Option : MOEPS*

MEMOIRE
Présenté par

Mr. OULD-SAADI Youcef

Pour l'obtention du diplôme de Magister en Informatique
Thème

Conception d'une Métaheuristique Réactive

Devant la commission d'examen composée de :

Président	Mr BENYETTOU Abdelkader	Professeur	USTO
Rapporteur	Mr SADOUNI Kaddour	M.conf. A	USTO
Examinateur	Mr BELKADI Khaled	M.conf. A	USTO
Examinateur	Mr RAHAL SidAhmed	M.conf. A	USTO
Invité	Mr ZENNAKI Mahmoud	M.A.A	USTO

Année universitaire : 2011-2012

Remerciement

Je tiens à remercier mon encadreur Mr *SADOUNI Kaddour*, mon co-encadreur *Mr ZENNAKI Mahmoud* pour le soutien qu'ils m'ont apporté et la confiance qu'ils m'ont témoigné, ce qui m'a été d'une grande aide dans la réalisation de ce travail.

Je les remercie aussi pour leur grande disponibilité et leurs précieux conseils, recevez ici le témoignage de ma sincère reconnaissance.

Je remercie également Mr *BENYETTOU Abdelkader*, de m'avoir fait honneur de présider mon jury, j'espère qu'il trouvera ici l'expression de ma profonde gratitude.

Je remercie Mr *BELKADI Khaled*, Mr *RAHAL Sid Ahmed*, d'avoir accepté d'examiner mon mémoire et pour l'intérêt qu'ils ont bien voulu porter à ce travail.

Dédicace

Je dédie ce modeste travail

A mes très chers parents en témoignage de ma profonde affection.

A mes très chères sœurs et frères.

A tous mes amis ainsi que toutes les personnes que j'ai connues, qui m'ont aidé, soutenu et encouragé, et qu'une simple dédicace ne peut contenir, mais leur noms resterons toujours gravés dans mon cœur.

SOMMAIRE

LISTE DES TABLEAUX

LISTE DES FIGURES

LISTE DES ALGORITHMES

INTRODUCTION GENERALE

Les problèmes d'optimisation sont omniprésents, aussi bien dans le monde académique que dans l'univers industriel. Beaucoup d'entreprises et d'organisations font face à différents types de défis, tels que : minimiser un coût de production, rationaliser l'utilisation de ressources, améliorer les performances d'un système de production, ou fournir une aide à la décision, etc.

Ces taches complexes sont souvent difficiles à résoudre par des méthodes dites exactes, qui ont l'avantage de garantir l'optimalité des solutions, mais qui prennent un temps de calcul exorbitant, à cause de l'explosion combinatoire.

C'est dans le but de réduire cette explosion, que des méthodes dites approchées (ou métaheuristique) ont été conçu. Elles ne garantissent pas l'optimalité, mais offrent des solutions presque optimales en un temps raisonnable. Cependant elles peuvent être appliquées à un large ensemble de problèmes, à cause de leur adaptabilité et leur indépendance de la structure du problème traité.

Les recherches actuelles, autour des métaheuristiques, ont évolué vers de nouveaux enjeux tels que : robustesse, intelligence, efficacité et auto adaptation.

Cependant, le choix des paramètres d'une métaheuristique est très critique ; il affecte, d'une manière considérable, ses performances et son comportement. Généralement, les concepteurs des métaheuristiques font recours à de séries d'expérimentations, afin de sélectionner les bons paramètres (paramétrage hors ligne), ce qui est coûteux en temps d'exécution et nécessite une connaissance approfondie de la stratégie de recherche utilisée ; ce qui n'est pas le cas pour un utilisateur final.

Dans ce mémoire, le travail présenté est consacré à l'étude et l'analyse de différentes stratégies, faisant face à ce type de difficultés ; car il est possible de collecter des informations pertinentes, pendant le processus de résolution, afin de sélectionner les paramètres convenables et par conséquent, guider plus efficacement, l'exploration de l'espace de recherche. Des techniques, consistant à combiner des heuristiques de résolution avec des méthodes de fouille de données et d'apprentissage, et qui ont pour objectif, l'extraction d'informations implicites, auparavant inconnues, mais potentiellement utiles, ceci à partir de grands volumes de donnée. Cet objectif est identique à celui utilisé par des chercheurs en optimisation combinatoire, et qui sont confrontés au déploiement de méthodes d'exploration dans de très vastes espaces de recherche.

Notre objectif principal, est de concevoir une technique, rendant la stratégie de recherche, plus informée, et plus « auto consciente » du problème. Bien que l'apprentissage automatique semble une bonne piste, il demeure un certain nombre de défis à relever. Le plus important de ces défis, concerne le niveau opérationnel et la complexité de calcul : il est essentiel, que toute information supplémentaire, intégrée dans un processus de résolution, n'induise pas d'importants coûts, supplémentaires, de calcul.

Une métaheuristique réactive intègre des techniques d'apprentissage artificiel à son mécanisme de recherche afin de résoudre des problèmes d'optimisation complexes. La fouille dans l'historique de recherche ainsi que les connaissances accumulées durant l'exécution permet à l'algorithme d'être auto-adaptatif.

En ce qui concerne le plan de ce mémoire, il est composé de six chapitres : Le premier est divisé en deux parties ; dont une est consacrée à l'étude théorique des notions de base et définitions des problèmes d'optimisation combinatoire et l'autre partie, consacrée à la présentation des stratégies de paramétrage utilisées dans les métaheuristiques ainsi que la motivation vers les approches de paramétrage en ligne.

Dans le deuxième chapitre, nous introduisons le principe de la réactivité sur la fonction objectif.

Le troisième chapitre est consacré à la réactivité sur le voisinage, nous présentons une description détaillée des méthodes utilisées.

Le quatrième chapitre est dédié à la Recherche Tabou et la Réactivité ; nous présentons plusieurs stratégies d'exploration de l'espace de recherche, qui servent à sélectionner la taille de la liste tabou d'une manière réactive.

Dans le cinquième chapitre, Nous présentons les principes du Recuit simulé, nous présentons aussi quelques techniques qui permettent de rendre réactif certain de ses paramètres.

Nous terminerons notre mémoire par un sixième chapitre, dans lequel nous intégrons à l'algorithme de recherche tabou, certains principes, vus dans les chapitres précédents ; ce procédé permet d'améliorer les performances de l'algorithme, en l'aidant à mieux s'adapter au problème traité.

CHAPITRE I : Concepts de bases et motivation

1. Introduction

L'optimisation combinatoire occupe une place très importante en recherche opérationnelle et en informatique. Les problèmes d'optimisation sont souvent des problèmes NP-difficiles et donc ne possèdent pas des méthodes exactes permettant de les résoudre en un temps polynomial. La plupart des métaheuristiques, bien qu'elles sont très efficaces et utiles pour de nombreux problèmes pratiques, elles ne garantissent pas l'optimalité des solutions et sont très sensibles au choix de leurs paramètres.

Nous introduisons, dans ce premier chapitre, le contexte et la motivation des approches heuristiques, en optimisation combinatoire. Nous portons une attention particulière aux méthodes de paramétrage des métaheuristiques, à savoir les méthodes hors ligne et en ligne.

2. Optimisation combinatoire

Les problèmes d'optimisation combinatoire sont définis par plusieurs manières [1].

2.1. Problème d'optimisation

Un problème d'optimisation se définit comme la recherche du minimum ou du maximum d'une fonction donnée. Mathématiquement, dans le cas d'une minimisation, un problème d'optimisation se présentera sous la forme suivante :

Minimiser $f(s)$ (fonction à optimiser)

avec $g(s) \leq 0$ (m contraintes d'inégalités)

et $h(s) = 0$ (p contraintes d'égalités)

En d'autres termes, résoudre un problème d'optimisation $P(S, f)$ revient à déterminer une solution $s^* \in S$ minimisant ou maximisant la fonction f avec S, 1'ensemble des solutions, ou 1'espace de recherche ; donc $f : S \rightarrow Y$ une application, ou une fonction d'évaluation qui a chaque configuration s associe une valeur $f(s) \in Y$.

Il est possible de passer d'un problème de maximisation à un problème de minimisation grâce a la propriété suivante :

$$max_{s \in S} f(s) = min_{s \in S}(-f(s))$$

Généralement, une solution $s \in S$ est un vecteur d'un espace à N dimensions.

2.2. Problème d'optimisation continue

Dans le cas de variables réelles, on a: $S \subseteq \mathbb{R}^N$. On parle alors de problème d'optimisation en variables continues.

Un problème d'optimisation continue *(PO)* peut être formulé de la façon suivante :

$$(PO)\ max_{s \in \mathbb{R}^N}\ f\ (s)$$

2.3. Problème d'optimisation combinatoire

Un problème d'optimisation combinatoire est un problème d'optimisation dans lequel l'espace de recherche S est dénombrable

Un problème d'optimisation combinatoire *(POC)* peut être formulé ainsi :

$$(POC)\ max_{s \in Z^N}\ f\ (s)$$

Où une solution s est un vecteur composé de N valeurs entières, soit : $S \subseteq Z^N$.

La principale différence entre problème d'optimisation continue et problème d'optimisation combinatoire repose sur l'utilisation de variable discrète. Dans les deux catégories de problèmes, une solution $s \in S$ est une instanciation des variable $x_i \in S$, où i est l'indice de la variable dans [1 ; N], et *X* est le vecteur de dimensions *N* correspondant à la solution, et $f(s)$ est son évaluation. Résoudre ces problèmes revient à trouver une solution *optimale appelée* aussi *optimum global*.

3. Les méthodes de résolution

Il existe un grand nombre de problèmes d'optimisation. Les problèmes les plus classiques de programmation linéaire utilisent une fonction objectif ainsi que des contraintes linéaires. Un problème d'optimisation exprimé sous la forme d'un programme linéaire chercher à maximiser une fonction de *N* variables x_i en respectant m contraintes linéaires. Il peut s'écrire de la façon suivante [2]:

$$(PL)\begin{cases} \max f(x_1, x_2, \dots, x_N) = c_0 + c_1x_1 + \cdots + c_Nx_N \\ a_{i1}x_1 + a_{i2}x_2 + \cdots + a_{iN}x_N \ \leq b_i \ \ i = 1, \dots, m \\ x_j \in \mathbb{R} \qquad\qquad j = 1, \dots, p \\ x_j \in \mathbb{R}^+ \qquad\qquad j = p+1, \dots, q \\ x_j \in \mathbb{N} \qquad\qquad j = q+1, \dots, N \end{cases}$$

Avec c_i les coefficients de la fonction à maximiser, a_{ij} et b_i les coefficients des contraintes et $\mathbb{R}^+$ 1'ensemble des réels positifs ou nuls.

Les problèmes de programmation linéaire peuvent utiliser des variables réelles, des variables réelles positives ou nulles et des variables entières. Dans ce cas, ils sont appelés programmes linéaires en variables mixtes. Si toutes les variables du problème sont entières on parle alors de programmes linéaires en variables entières ou encore de programme linéaire en *0-1* si les variables sont de type booléen. Ce type de problème est résolvable de manière exacte via différentes méthodes comme la méthode du simplexe par exemple pour les problèmes en variables continues. La principale caractéristique d'une méthode exacte est la garantie de trouver un optimum global. Beaucoup d'études ont été menées afin de modéliser sous la forme d'un problème de programmation linéaire les problèmes d'optimisation combinatoire, utilisés généralement. Certains problèmes combinatoires ont pu être modélisés sous forme linéaire cependant la plupart d'entre eux ne le sont pas.

3.1. Problèmes difficiles et algorithmes heuristiques

Une instance d'un problème d'optimisation combinatoire est définie par un ensemble de solutions candidates et une fonction objectif. Résoudre un tel problème (plus précisément, une telle instance du problème) consiste à trouver une solution qui optimise ou minimise la fonction objectif donnée. Dans la pratique, certains problèmes combinatoires sont considérés comme des problèmes difficiles, ou non traitables.

En termes de complexité, ils sont généralement des problèmes NP-complets ou NP-durs (NP-hard) [3,4] : tout algorithme exact de résolution peut nécessiter un temps de calcul prohibitif, qui croit de manière exponentielle avec la taille de l'instance traitée.

Les problèmes combinatoires difficiles ont fait l'objet de plus de 50 ans de recherche intense dans plusieurs communautés liées aux algorithmes, incluant: l'algorithmique et la théorie de complexité, l'optimisation, la recherche opérationnelle et l'intelligence artificielle. Des progrès importants ont été réalisés, il existe à ce jour trois approches génériques [5,6] pour tenter de résoudre les problèmes difficiles, elles se résument par les approches suivantes:

- Étudier des classes d'instances qui admettent des algorithmes exacts efficaces
- Développer des algorithmes d'approximation de temps polynomial ;
- Développer des algorithmes heuristiques, métaheuristiques ou stochastiques.

Concernant la première approche, il est important de mentionner qu'un algorithme théoriquement exponentiel peut être remarquablement efficace dans la pratique. L'algorithme du simplexe est un exemple représentatif pour cette situation favorable : bien qu'il ait une complexité exponentielle dans le pire des cas, cet algorithme est couramment employé pour résoudre efficacement un très grand nombre de programmes linéaires. Il arrive que des algorithmes exponentiels résolvent facilement même des instances de grande taille pour certains problèmes pratiques.

Les problèmes difficiles peuvent être également abordés par des algorithmes polynomiaux d'approximation qui garantissent une solution avec une qualité bornée par un certain seuil par rapport à la qualité optimale. Bien qu'il y ait des problèmes NP-durs pour lesquels des algorithmes d'approximation aient été prouvés, la plupart des problèmes NP-durs n'admettent pas d'algorithme d'approximation polynomial. En effet, pour la plupart des problèmes les plus difficiles, une solution presque optimale ne peut pas être systématiquement garantie en un temps polynomial.

Une alternative qui a fait ses preuves en pratique pour de nombreux problèmes difficiles de grande taille est l'utilisation d'algorithmes de recherche heuristique. Ces algorithmes consomment des ressources limitées de temps et ils peuvent produire des solutions de bonne qualité. Les heuristiques permettent des gains très importants au niveau pratique, non seulement pour résoudre des problèmes NP-durs classiques, mais aussi pour de nombreux problèmes réels difficiles. Dans la pratique, il n'est pas toujours nécessaire de trouver une solution optimale et des solutions acceptables peuvent être souvent trouvées par recherche heuristique. De plus, les solutions optimales fournies par des algorithmes exacts peuvent être souvent atteintes plus rapidement par des heuristiques.

Le terme métaheuristique, représente une classe d'algorithmes heuristiques génériques, qui peuvent être appliqués à tout problème d'optimisation [5,6]; la seule condition nécessaire est de définir une représentation des configurations, ainsi qu'une fonction objectif. Certaines métaheuristiques exigent aussi une relation de voisinage (les recherches locales) ou un opérateur de mutation (les algorithmes évolutionnistes). Une métaheuristique n'est pas concernée par leur mise en œuvre précise ; elle s'intéresse uniquement au (méta) stratégies principales.

3.2. Caractéristique principale des métaheuristiques

Les propriétés fondamentales des métaheuristiques sont résumés des les points suivantes [6].

- Les métaheuristiques sont des stratégies qui permettent de guider la recherche d'une solution optimale
- Le but visé par les métaheuristiques est d'explorer l'espace de recherche efficacement afin de déterminer des solutions (presque) optimales.
- Les techniques qui constituent des algorithmes de type métaheuristique vont de la simple procédure de recherche locale à des processus d'apprentissage complexes.
- Les métaheuristiques sont en général non-déterministes et ne donnent aucune garantie d'optimalité
- Les métaheuristiques peuvent contenir des mécanismes qui permettent d'éviter d'être bloqué dans des régions de l'espace de recherche.
- Les concepts de base des métaheuristiques peuvent être décrits de manière abstraite, sans faire appel à un problème spécifique.
- Les métaheuristiques peuvent faire appel à des heuristiques qui tiennent compte de la spécificité du problème traité, mais ces heuristiques sont contrôlées par une stratégie de niveau supérieur.
- Les métaheuristiques peuvent faire usage de l'expérience accumulée durant la recherche de l'optimum, pour mieux guider la suite du processus de recherche.

Une autre caractéristique des métaheuristiques est largement utilisée pour effectuer une classification. Elle fait référence au nombre de solutions employées par la méthode. On distingue les méthodes utilisant une population de solutions et des méthodes utilisant une seul solution appelées recherche locale.

Dans le premiers cas, les méthodes sont basées sur la notion de population. Elles manipulent un ensemble de solutions en parallèle lors de chaque itération. Les algorithmes évolutionnaires et les algorithmes de colonies de fourmis sont des exemples de méthodes à base de population. Dans le second cas, les méthodes de recherche locale [7] sont basées sur la notion de trajectoire. La notion de voisinage est alors primordiale. Il existe un grand nombre de méthodes de recherche locale. Les plus connus sont probablement la Recherche Tabou, le recuit simulé.

Le travail présenté dans ce mémoire entre clairement dans le cadre des méthodes de recherche locale pour traiter des problèmes d'optimisation combinatoire.

3.3. Techniques d'apprentissage en optimisation combinatoire

Les métaheuristiques ont eu un succès impressionnant pour plusieurs problèmes. Une possibilité pour mieux comprendre le comportement d'un algorithme heuristique consiste à analyser son évolution avec des techniques de statistique, d'apprentissage et de fouille de données. Ces techniques permettent d'extraire des informations globales à partir des grands volumes de données traitées au cours de temps par le processus de recherche.

L'espace de recherche représente un exemple classique de grand volume de données, et ainsi, ce type de méthode statistique pourrait fournir des informations exploitables, qui peuvent être utilisées pour rendre la stratégie de recherche mieux-informée. Certes, beaucoup de métaheuristiques introduisaient déjà une forme d'apprentissage implicite. Toutefois, les méthodes de fouille de données nous permettent d'analyser des données plus complexes, pour aller au-delà des méthodes actuelles, citée dans la littérature.

- **Analyse de l'espace de recherche**

Des informations fondamentales peuvent être extraites par des analyses statistiques des espaces de recherche. En fait, toute heuristique efficace tient compte, explicitement ou implicitement, des propriétés structurelles de l'espace de recherche. Il existe plusieurs approches pour analyser l'espace et elles peuvent se focaliser sur les aspects suivants, sans se limiter à ceux-ci [5]:

- la forme de la surface de recherche (convexe ou non convexe, information primordiale dans l'optimisation numérique).
- des indicateurs de rugosité et de difficulté.
- les similarités et les variables partagées par les optima locaux.
- le nombre, la qualité et la distribution spatiale des optima locaux.

- **Apprentissage de nouvelles fonctions d'évaluation**

Elle consiste en une direction active de recherche, consacrée à des modifications de la fonction objectif qui permettent de mieux guider l'heuristique à travers l'espace des solutions. Pour toute configuration donnée, une meilleure fonction d'évaluation devrait mieux estimer le

potentiel pour conduire à une solution. Cela est particulièrement utile si de nombreuses configurations partagent la même valeur de la fonction objectif ; s'il y a une façon de discriminer ces configurations, il y aurait moins d'optima locaux pour le processus de recherche. Bien que la fonction d'évaluation soit souvent conçue par le chercheur, elle peut être aussi ajustée par le processus de recherche en fonction des informations acquises. L'algorithme *Guided Local Search* [Chapitre 2] est un exemple représentatif issu de cette direction de recherche ; il propose de modifier la fonction objectif lorsque le processus de recherche est bloqué dans un optimum local.

4. Sélection d'algorithme et de paramètres

4.1. Le choix de l'heuristique

Le premier problème pratique qui se pose à un utilisateur confronté à une application concrète est d'effectuer un choix parmi les différentes métaheuristiques disponibles. Ce choix est d'autant plus difficile qu'il n'existe pas de comparaison systématique et fiable des différentes métaheuristiques. Cependant, il est possible de caractériser les métaheuristiques selon quelques critères généraux, de manière à faciliter ce choix.

Les critères de comparaisons retenus sont les suivants :

- facilité d'adaptation au problème,
- possibilité d'intégrer des connaissances spécifiques au problème,
- qualité des meilleures solutions trouvées,
- rapidité de calcul, pour trouver une telle solution.

La connaissance d'éléments de théorie connus en algorithmique, ou dans le domaine des méthodes dites exactes, permet d'améliorer les performances, selon les caractéristiques du problème à résoudre.

4.2. Le choix des paramètres

Chaque métaheuristique nécessite un réglage de ses paramètres, ce réglage va permettre de donner de la souplesse et de la robustesse à l'algorithme, cependant, il requit beaucoup d'attention et s'avère une tache difficile et compliquée.

Les paramètres d'une métaheuristique ont une influence majeure sur le rendement et l'efficacité de la recherche. Il n'est pas évident de définir a priori les valeurs qui doivent être utilisées.

Les valeurs optimales des paramètres dépendent essentiellement du problème et des instances traitées, ainsi que du temps, que l'utilisateur veut dépenser pour la résolution du problème.

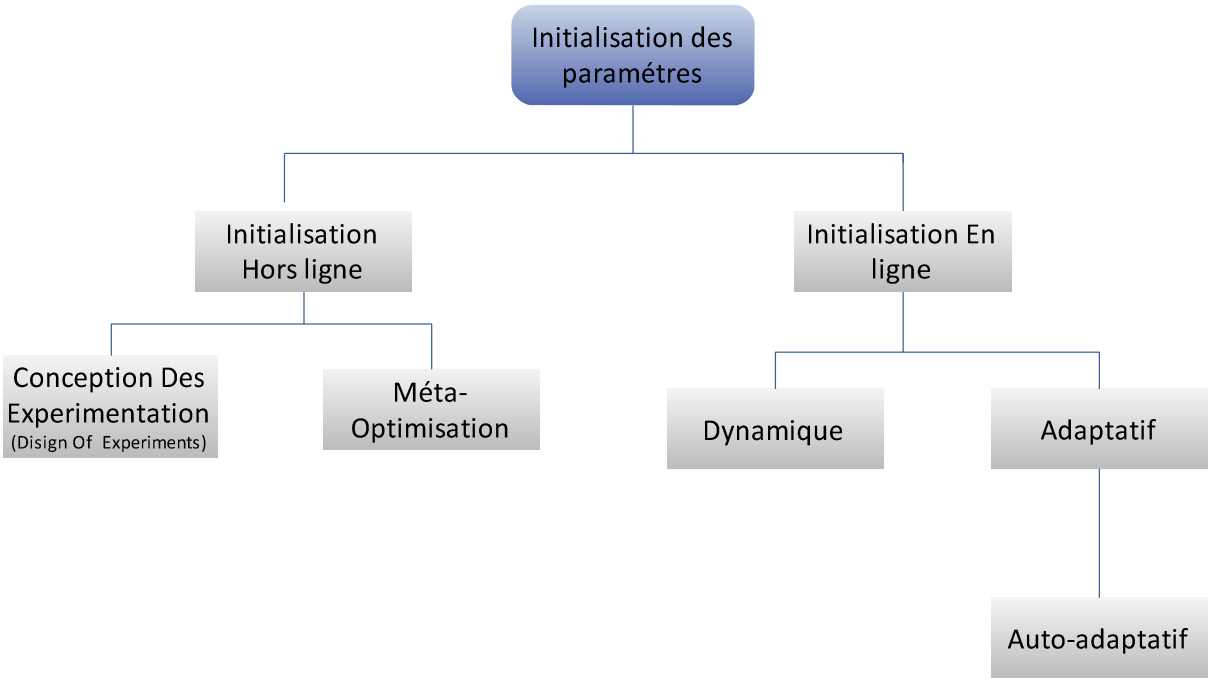

Figure 1 : Stratégies d'initialisation des paramètres.

Il n'existe pas de méthode standard consistant à trouver des paramètres idéaux pour n'importe quelle métaheuristique

Les approches de paramétrage en optimisation peuvent être classifiées selon que la méthode soit "en ligne" ou "hors ligne" (**Figure 1**). Par exemple, la plupart des analyses de l'espace de recherche font partie des méthodes "hors ligne" : ces analyses sont réalisées dans une étape de pré-optimisation, avant de commencer la phase principale d'optimisation.

Bien qu'une analyse préalable puisse fournir des directives essentielles pour le choix et la construction d'un algorithme approprié, elle n'est pas employée pour prendre des décisions

pendant le processus d'optimisation, ex. pour réagir à des informations acquises durant l'exploration.

Dans l'approche hors ligne, les valeurs des différents paramètres sont fixées avant l'exécution. Dans l'approche en ligne, les paramètres sont contrôlés et modifiés dynamiquement, ou ils sont adaptés pendant le processus d'optimisation.

5. Le paramétrage hors ligne

Comme nous l'avons mentionné précédemment, les métaheuristiques ont un inconvénient majeur; ils nécessitent une phase de paramétrage, qui n'est pas faciles à réalisée. Il ne s'agit pas seulement des paramètres à valeurs numériques, mais aussi d'une structure complexe et relative, du problème traité.

Généralement, les concepteurs des métaheuristiques ajustent un paramètre à la fois, et sa valeur optimale est déterminée de manière expérimentale. Dans ce cas, aucune interaction entre les paramètres n'est étudiée. Cette stratégie d'optimisation séquentielle (c'est à dire, un paramétrage un par un) ne garantie pas de trouver le paramètre optimal.

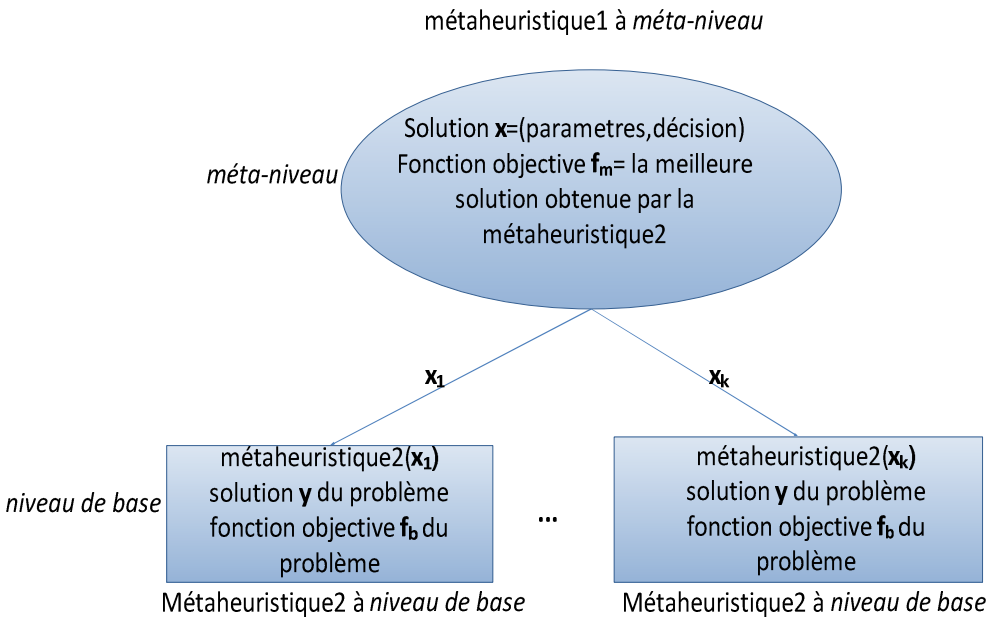

Figure 2 : Méta-Optimisation utilisant une métaheuristique.

Pour surmonter ce problème, l'approche de conception expérimentale (Plan d'expérience) est utilisée [8]. Avant d'utiliser une approche de conception expérimentale, les concepts suivants doivent être définis:

- Etablir des Facteurs (appelée aussi variable de conception, variable d'entrée ou variable prédictive) qui représentent les paramètres qui varient durant les expériences.
- Les niveaux qui représentent les différentes valeurs des paramètres, qui peuvent être quantitatifs (ex, la probabilité de mutation) ou qualitatifs (ex, le voisinage).

Prenons $\boldsymbol{n}$ facteurs dont $\boldsymbol{k}$ le nombre des niveaux de chaque facteur, un design factoriel complet a besoin de $\boldsymbol{n^k}$ expériences. Ensuite, le meilleur des niveaux est sélectionné pour chaque facteur. Le principal inconvénient de cette approche est son coût de calcul exorbitant surtout quand le nombre de paramètres (facteurs) et leurs valeurs de domaine sont importants, un très grand nombre d'expériences doit être réalisées [9]. Cependant, un petit nombre d'expériences peut être effectuée en utilisant la méthode *Latin hypercube designs* [10], *la conception séquentielle*, ou la *conception fractionnaire* (fractional design) [11].

D'autres approches utilisées dans le domaine d'apprentissage automatique tel que les *Racing Algorithms* [12], peuvent être aussi considérées.

Dans l'approche *hors ligne*, la recherche des paramètres d'une métaheuristique pour un problème donné peut être formulée comme un problème d'optimisation.

Cette méta-optimisation peut être effectuée par n'importe quelle heuristique (méta), conduisant à une approche méta-métaheuristique. La méta-optimisation peut être ainsi, considéré comme un schéma hybride dans la conception des métaheuristiques (**Figure 2**).

Cette approche est composée en deux niveaux: le *méta-niveau* (meta-level) et le *niveau de base* (base level).

Au *méta-niveau*, une métaheuristique opère sur les solutions (ou populations) qui représente les paramètres de la métaheuristique d'optimisation. Une solution x au *méta-niveau* représente tous les paramètres que l'utilisateur veut optimiser: tels que la taille de la liste tabou pour la Recherche Tabou, la variation de la température dans le recuit simulé, la probabilité de mutation et de croisement pour les algorithmes génétiques. Le type de stratégie de sélection dans les algorithmes évolutionnaires, le type de voisinage pour une recherche locale, et ainsi de suite.

Au *méta-niveau*, La fonction objectif $\mathbf{f_m}$ associée à une solution x est généralement la meilleure solution trouvée (ou tout autre indicateur de performance) par la métaheuristique utilisant les paramètres spécifiés de la solution **x**. Ainsi, pour chaque solution x du *méta-niveau* correspondra une métaheuristique indépendante du *niveau de base*.

La métaheuristique du niveau de base opère sur les solutions (ou populations) du problème d'optimisation à résoudre. La fonction objective $\mathbf{f_b}$ utilisée par la métaheuristique du niveau de base est associé au problème cible. La formule suivante montre la relation entre les deux niveaux:

$$\boldsymbol{f_m(x) = f_b(Meta(x))}$$

Où $\boldsymbol{Meta(x)}$ représente la meilleure solution retournée par la métaheuristique utilisant le paramètre $\boldsymbol{x}$.

L'inconvénient majeur des approches hors-ligne est leur coût important de calcul, surtout s'ils doivent être utilisés pour chaque instance saisie du problème. En effet, les valeurs optimales des paramètres dépendent largement du problème traité, et de ses diverses instances.

Pour améliorer l'efficacité et la robustesse des approches Hors-ligne, ils doivent être appliqués à n'importe quelle instance d'un problème donné. Une autre alternative consiste à utiliser une approche parallèle *Multi Start* qui utilise différents configurations des paramètres en parallèle [13].

Un autre inconvénient important des stratégies *hors ligne* est que l'efficacité d'un paramètre peut changer durant la recherche, les valeurs optimales d'un paramètre varient d'un moment à l'autre, et les paramètres qui sont optimales pour une séquence de recherche ne les sont plus dans une autre séquence. Par conséquent, les approches *en ligne,* qui adaptent les valeurs des paramètres lors de la recherche, doivent être conçues.

6. Le paramétrage en ligne

Aujourd'hui, une technique assez commune consiste à utiliser des mécanismes réactifs pour adapter automatiquement les valeurs des paramètres durant le processus de recherche. Le terme mécanisme réactif fait référence à une boucle rétroactive qui modifie un paramètre selon l'état actuel de la recherche, d'une manière *en ligne*.

Cependant, d'autres algorithmes peuvent avoir des réactions sur le voisinage, sur la fonction d'évaluation, ou sur le programme de recuit [14] (Nous abordons ces principes en détails dans les prochains chapitres).

Le paramétrage automatique a des implications sur la direction de recherche, un changement de paramètre peut déclencher une transformation complète d'algorithme. Avec un tel changement, on peut transformer un Recuit Simulé en Descente Pure; les opérateurs d'algorithmes génétiques peuvent être aussi choisis de cette façon [15].

Tout apprentissage *en ligne* est limité par de fortes contraintes de complexité, parce qu'il ne doit pas introduire un ralentissement important dans le processus de recherche. Les heuristiques sont habituellement des algorithmes rapides, qui doivent visiter de nombreuses configurations dans un immense espace de recherche.

Les Approches en ligne peuvent être classées comme suit:

- **Approche Dynamique :** Dans cette approche, le changement de la valeur des paramètres est effectué sans tenir compte de la progression de la recherche. Une modification aléatoire ou déterministe de la valeur des paramètres est effectuée.

- **Approche Adaptative (réactive) :** L'approche adaptative permet de changer les valeurs des paramètres en fonction de la progression de la recherche. Ceci est réalisé en utilisant une mémoire pour sauvegarder la trajectoire de la recherche. Les paramètres sont auto-adaptatifs, et sont sujettes à des changements aussi-bien que les solutions du problème.

Le schéma ci-après illustre ce procédé de fonctionnement.

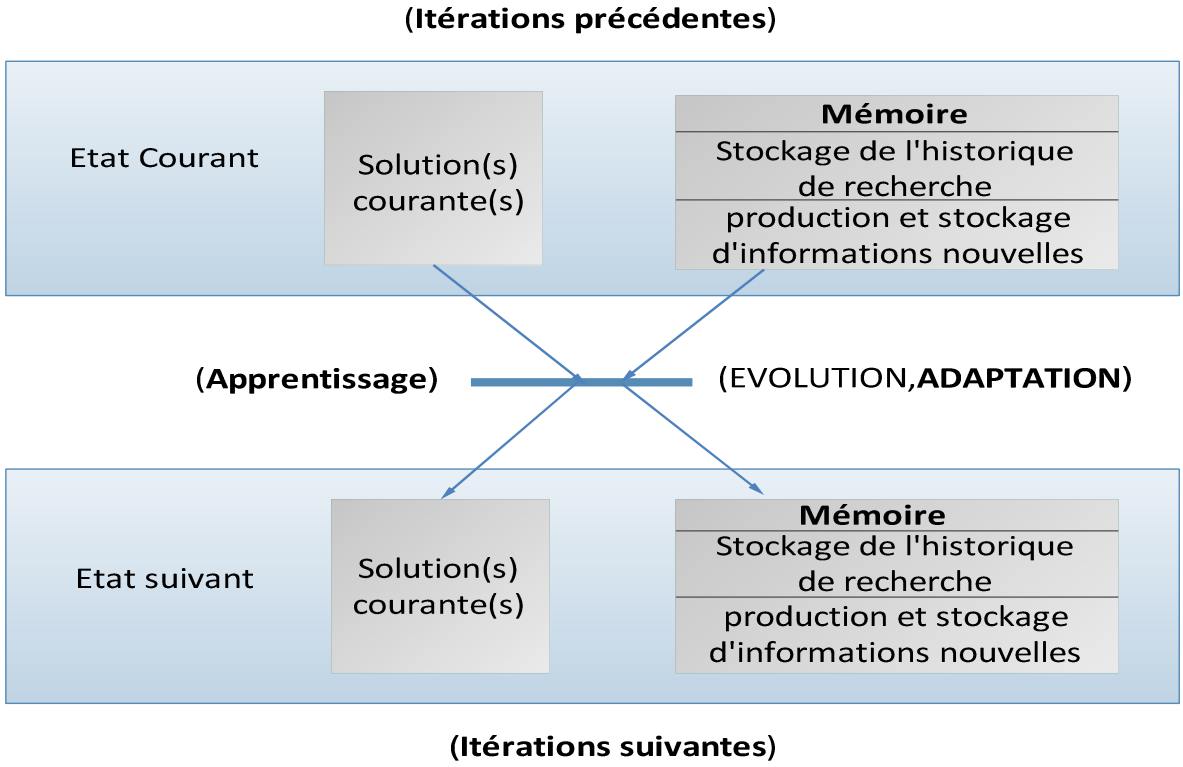

Figure 3 : Schéma de fonctionnement d'une approche adaptative.

7. Paramétrage en ligne vs hors ligne

Le tableau suivant présente un bref comparatif entre le paramétrage hors ligne et le paramétrage en ligne :

Tableau 1 : comparaison entre une démarche en ligne et hors ligne

	Avantage :	**inconvénient**
Paramétrage hors ligne	• L'approche n'intègre pas un mécanisme de sauvegarde de l'espace de recherche qui peut alourdir la métaheuristique.	• Gaspillage de temps lors de la phase d'initialisation des paramètres. • Les paramètres ne sont pas adaptatifs. • Besoin de connaissance et de maitrise pour l'utilisateur finale.
Paramétrage en ligne	• Paramètres auto- adaptatifs • Gain de temps (pas de pré-initialisation des paramètres) • Qualité de solution meilleure.	• alourdir la méthode (sauvegarde de l'espace de recherche).

8. Conclusion

Dans ce chapitre, nous avons dans un premier temps défini des notions de base sur les problèmes d'optimisation combinatoire et les métaheuristiques. Ensuite Nous avons présenté les deux approches utilisées pour le paramétrage des métaheuristiques ; l'approche hors ligne et l'approche en ligne. Cette dernière peut, potentiellement améliorer le comportement des métaheuristiques en introduisant un apprentissage automatique et un mécanisme adaptatif à l'algorithme.

CHAPITRE II : Réactivité sur la fonction objectif

1. Introduction

La fonction objectif est utilisé en optimisation pour désigner une fonction qui sert de critère pour déterminer la meilleure solution à un problème d'optimisation. C'est un des concepts les plus fondamentaux dans l'élaboration d'une métaheuristique.

Généralement, la fonction objectif est définie d'une manière statique (choisie en préalable et qui reste figée durant tout le processus de recherche).

Nous présentons dans ce chapitre le principe de la réactivité sur la fonction objectif ; l'idée est d'utiliser une technique de recherche locale dans laquelle la fonction objectif varie durant le processus de recherche, le but étant de rendre les minima locaux déjà visités moins attractifs.

2. La fonction objectif

La fonction objectif f (ou la fonction de coût) formule l'objectif à atteindre. Elle associe à chaque point de l'espace de recherche une valeur qui décrit la qualité de la solution, $f : S \rightarrow R$. Le but du problème d'optimisation est alors de minimiser ou de maximiser cette fonction.

La fonction objectif représente un élément important dans la conception d'une métaheuristique. Elle va guider la recherche vers les bonnes solutions de l'espace de recherche. Si la fonction objectif est mal conçue, elle pourrait conduire à des solutions non acceptables quel que soit la métaheuristique utilisée.

Dans ce qui suit, nous présentons une méthode qui adapte la fonction objectif durant le processus de recherche afin de guider la stratégie de recherche vers des endroits plus appropriés de l'espace de recherche.

3. La Recherche Locale Guidée

La Recherche Locale Guidée (RLG) est une stratégie de recherche intelligente pour des problèmes d'optimisation combinatoire. Une caractéristique principale de cette approche est l'utilisation itérative de la recherche locale [16], l'information est recueillie et exploitée à partir de sources diverses afin de guider la recherche locale dans des régions prometteuses de l'espace de recherche.

La RLG modifie et transforme dynamiquement la fonction objectif selon les caractéristiques des optimums locaux obtenus (**Figure 4**), elle permet la modification de la structure du paysage exploré pour échapper d'un optimum local.

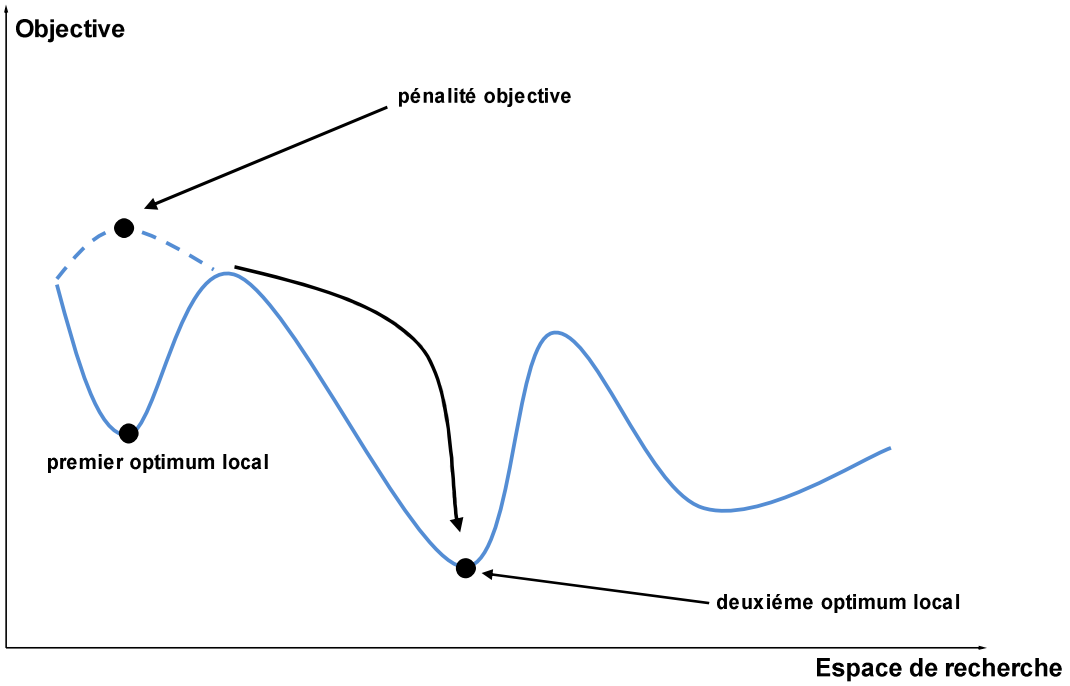

Figure 4 : Recherche Locale Guidée pénalisant les solutions trouvées pour échapper d'un optimum local. La fonction objectif est modifiée selon les caractéristiques des solutions trouvées.

Considérant qu'un ensemble de m caractéristiques d'une solution sont définies, noté $CR_I(i=1,\ldots,m)$. Un coût c_i est associé à chaque caractéristique. Lorsque la recherche est piégée dans un optimum local, l'algorithme pénalise certaines solutions en fonction de quelques caractéristiques choisies. Chaque fonction i est associée à une pénalité p_i qui représente l'importance d'une caractéristique. La fonction objectif f associée à une solution S est alors pénalisée comme suit:

$$f'(s) = f(s) + \lambda \sum_{i=1}^{m} p_i I_i(s)$$

Où λ représente le poids associé à des pénalisé différentes et $I_i(s)$ un indicateur qui détermine si la caractéristique CR_I est présente dans la solution S:

$$I_i(s) = \begin{cases} \mathbf{1}\ si\ la\ \text{caractéristique}\ \boldsymbol{CR}_i \in \mathbf{S} \\ \\ \mathbf{0}\ si\ non \end{cases}$$

Toutes les pénalités sont initialisées à 0.

4. Identification des caractéristiques appropriées

L'application de la Recherche Locale Guidée à un problème d'optimisation donné nécessite l'identification des caractéristiques appropriées de ses solutions. Le choix des caractéristiques dépend principalement du problème d'optimisation traité [13] :

4.1. Problèmes de routage

Pour des problèmes de routage tels que le *problème de voyageur de commerce*, ou le *problème de tournées de véhicules*, une caractéristique peut être associée par la présence d'une arête ou arc *(a, b)* dans la solution, le coût correspondrait à sa distance (ou temps de déplacement) associée à la solution.

4.2. Les problèmes d'affectation

Dans ce genre de problèmes, telles que le *problème d'affectation généralisée* et *le problème de positionnement*, une solution représente l'association d'un nombre d'objets à un ensemble de locations. Une caractéristique peut être représentée par la paire *(i, k)*, où $\boldsymbol{i}$ représente un objet et $\boldsymbol{k}$ son emplacement spécifique, tel que :

$$I_{ik}(s) = \begin{cases} \mathbf{1} \quad si\ \boldsymbol{s(i) = k} \\ \\ \mathbf{0} \quad si\ non \end{cases}$$

Le coût $\boldsymbol{c_{ik}}$ correspondrait à l'affectation de l'objet $\boldsymbol{i}$ à l'emplacement $\boldsymbol{k}$.

4.3. Problèmes de satis faisabilité

Dans ce genre de problèmes, tels que le *MAX-SAT*, l'objectif est généralement liée au nombre de contraintes violées. Une caractéristique peut être associée à chaque contrainte, tandis que le coût serait lié à la violation de la contrainte.

La question principale qui se pose, est la façon dont les caractéristiques sont sélectionnées et pénalisées. Le but est de pénaliser les caractéristiques qui sont présentes dans les optimums locaux obtenus, ou les caractéristiques qui sont défavorables.

Étant donné un optimum local s^*, une utilité u_i est alors associée à chaque caractéristique i comme suit :

$$u_i(s^*) = I_i(s^*)\frac{c_i}{1+p_i}$$

Où c_i représente le coût de la caractéristique i.

Par exemple, si une caractéristique donnée n'est pas présente dans un optimum local s^* alors l'utilité de la caractéristique serait égale à 0, si non l'utilité serait proportionnelle au coût c_i et inversement proportionnelle à la pénalité p_i.

Ensuite, la caractéristique associée à l'utilité la plus grande sera pénalisée.

D'une part, la RLG va intensifier la recherche dans des régions prometteuses, définies par des caractéristiques à bas coûts. D'autre part, elle diversifiera la recherche en pénalisant les caractéristiques des optimums locaux pour les éviter.

Le pseudo code suivant explique le principe de la RLG :

Algorithme 1 : principe d'une Recherche Locale Guidée

Entrée: λ, *caractéristique I, coût c.*
$s=s_0$; /*générer la solution initiale*/
$p_i=0$; /*initialisation des pénalités*/
Répéter
  $i=1$;
  Pour chaque caractéristique i de s^* **Faire**
    $u_i = \frac{c_i}{1+p_i}$; /*calculer l'utilité*/
    $u_j=max_{i=1,..m}(u_i)$; /*sélectionner le maximum des utilités*/
    $p_j= p_{j+1}$; /*modifier La fonction objectif en pénalisant la caractéristique j*/
  Fin Pour
Jusqu'à ce que le critère d'arrêt soit satisfait
Sortie: meilleure solution trouvée.

Les paramètres d'entrée de l'algorithme RLG sont représentés par, le choix des caractéristiques, leurs coûts et le paramètre λ. Les caractéristiques et les coûts sont dépendants du problème d'optimisation traité. La littérature suggère que les performances de

l'algorithme n'est pas très sensible à la valeur de **λ** [17]. Une grande valeur de **λ** encourage la diversification, tandis qu'une petite valeur, intensifie la recherche autour de l'optimum local. Une stratégie commune pour initialiser **λ** consiste à diviser la valeur de la fonction objectif de l'optimum local, par le nombre moyen des caractéristiques qui y présentent.

5. Recherche Locale Guidée pour le Problème de Voyageur de Commerce

Une solution du PVC est générée par la permutation d'un ensemble de villes, dans une solution S. La fonction objectif initiale, consiste à minimiser la distance totale:

$$f(s) = \sum_{i=1}^{n} d_{i,s(i)}$$

Où n représente le nombre de villes.

Comme nous l'avons présenté avant, une caractéristique du *PVC,* peut être représentée par une arête, l'indication de la présence d'une caractéristique peut être représentée comme suit:

$$I_{(i,j)}(s) = \begin{cases} 1 & si\ arête(i,j) \in s \\ 0 & si\ non \end{cases}$$

Le coût associé à une arête (i, j) est la distance d_{ij}. Les pénalités sont représentées par une matrice $P_{n,n}$, où p_{ij} représente la pénalité de l'arête (i, j). Tous les éléments de la matrice P sont initialisés à 0.

La modification de la fonction objectif du problème est formulé comme suit [13,18] :

$$f'(s) = \sum_{i=1}^{n} d_{i,s(i)} + \lambda\, p_{i,s(i)}$$

La fonction d'utilité est définie comme suit:

$$u_i(s^*, (i,j)) = I_{(i,j)}(s)\frac{d_{ij}}{1 + p_{ij}}$$

6. Conclusion

Nous avons présenté dans ce chapitre une technique qui permet d'adapter un des paramètres les plus importants d'une métaheuristique, en l'occurrence, la fonction objectif.

La technique de la *Recherche Locale Guidée* permet de varier la fonction objectif durant le processus de recherche, afin de rendre les minima locaux déjà visités moins attractifs et par conséquence avoir davantage de chance de trouver l'optimal.

CHAPITRE III : Réactivité le voisinage

1. Introduction

La notion de voisinage est sans doute le principe général le plus utilisé pour la conception d'une métaheuristique. Pour les problèmes combinatoires, le voisinage a un impact important sur son comportement. Généralement les concepteurs des métaheuristiques utilisent un voisinage fixe, que se soit de taille ou de structure, ce qui est considéré, malgré sa simplicité, comme un inconvénient, puisque un voisinage qui est bon pour un endroit de l'espace de recherche ne l'est forcement pas, pour un autre endroit.

Nous introduisons dans ce chapitre, les techniques et les méthodes qui permettent à une métaheuristique basée sur la *recherche locale,* d'adapter son voisinage durant la recherche.

2. Notions de voisinage

Les méthodes de voisinage sont fondées sur la notion de *voisinage*. Nous allons donc introduire d'abord cette notion fondamentale, ainsi que d'autres notions qui lui sont associées.

La définition de la structure de voisinage est une étape clef, dans la mise au point d'un algorithme de recherche locale, puisqu'elle permet de définir l'ensemble des solutions qu'il est possible d'atteindre, à partir d'une solution donnée, par une série de transformations. En particulier, la définition du voisinage permet d'identifier toutes les paires de solutions voisines.

2.1. Définition du Voisinage

Le voisinage de s est un sous-ensemble de configurations de S, directement atteignables à partir d'une transformation donnée de s. Il est noté $V(s)$ et une solution $s' \in V(s)$ est dite voisine de s.

Cette notion de voisinage structure 1'espace de recherche dans le sens ou elle permet de définir des sous-ensembles de solutions. En particulier, à partir d'une solution donnée, on peut établir plusieurs structures de voisinage selon la transformation que 1'on s'autorise, celle-ci étant définie comme une application $V:S \rightarrow P(S)$. Chaque structure de voisinage fournit un voisinage, ou un ensemble de solutions, précis via la transformation définie.

Pour caractériser une structure de voisinage dans le contexte de l'optimisation, elle peut être définie en fonction de différentes propriétés. Il en existe trois principales [6]:

1. La *complexité spatiale* qui détermine la relation entre la taille du voisinage d'une solution et la taille de 1'espace de recherche du problème.

2. La *complexité temporelle* qui correspond au temps nécessaire pour 1'évaluation des solutions voisines en fonction de la taille du voisinage.

3. La *portée* qui permet de mesurer le degré de transformation entre la solution courante et les solutions voisines. Trois degrés de portée sont proposés dont les qualifications générales sont assez imprécises mais qu'il convient d'apprécier pour chaque problème. Le premier degré de portée est la portée locale qui est caractérisée par l'introduction de peu de modifications dans la solution courante. Le second degré est nomme régional car une plus grande quantité de modifications est introduite dans les solutions voisines par rapport à la solution courante. Enfin, les structures de voisinage de portée globale transforment beaucoup les caractéristiques de la solution initiale.

La portée permet de différencier deux structures de voisinage en fonction de 1'é1oignement de 1'ensemble des solutions accessibles par rapport à la solution initiale.

2.2. Portée d'une structure de voisinage

La portée d'une structure de voisinage ***V(s)*** est la distance *d* maximale des solutions ***s' ∈V (s)*** par rapport à la solution ***s***.

Deux structures de voisinage ***V1*** et ***V2*** sont de portées identiques si et seulement si :

$$d(s, V_1(s)) = d(s, V_2(s)), \forall s$$

Avec,

$$d(s, V_k(s)) = max_{s' \in Vk(s)}(d(s, s'))$$

et ***d(s, s')*** la distance entre les solutions **s** et **s'**. Elles sont de portées différentes dans le cas contraire.

2.3. Optimum local

Une solution s E S est un optimum local si et seulement si il n'existe pas de solution ***s' ∈V (s)*** dont l'évaluation est de meilleure qualité que ***s***, soit :

$$\forall s' \in V(s) = \begin{cases} f(s) \leq f(s') \text{ dans le cas d'un problème de minimisation} \\ f(s) \geq f(s') \text{ dans le cas d'un problème de maximisation} \end{cases}$$

avec ***V(s)*** 1'ensemble des solutions voisines de ***s***.

La définition d'un optimum local est liée à la structure de voisinage. En d'autres termes, un optimum local pour une structure de voisinage donnée ne l'est pas forcement pour une autre structure de voisinage.

2.4. Optimum global

Une solution ***s*** qui vérifie la propriété précédente pour toutes les structures de voisinage du problème est appelée optimum global.

Beaucoup d'algorithmes de recherche spécifiques à un problème (heuristiques) ou généraux (métaheuristiques) utilisent une seule structure de voisinage. L'intérêt est la simplicité de mise en œuvre et la meilleure compréhension de ce que fait l'algorithme. Cependant, l'utilisation de plusieurs structures de voisinage présente un intérêt certain : une solution s reconnue comme un optimum local pour une structure de voisinage donnée peut ne pas être un optimum local pour une autre structure de voisinage. Il peut donc être intéressant d'utiliser des algorithmes combinant plusieurs structures de voisinage pour la recherche dans les problèmes disposant de nombreux minimums locaux.

3. Voisinage à taille variable

Lors de la conception d'une métaheuristique, un compromis doit être fait entre la taille (ou diamètre) du voisinage à utiliser et sa complexité de calcul, pour l'explorer. Concevoir un voisinage de grande taille peut améliorer considérablement la qualité des solutions obtenues, mais nécessite un temps de calcul supplémentaire pour le générer et l'évaluer.

La taille d'un voisinage pour une solution ***s*** est le nombre de ses solutions voisines. La plupart des métaheuristiques utilisent un voisinage de taille réduite, qui a souvent une complexité polynomiale. Certains voisinages de grande taille peuvent avoir une complexité polynomiale d'ordre élevée ou exponentielle.

Un voisinage a une complexité exponentielle, si sa taille augmente exponentiellement avec la taille du problème. Par conséquence, la complexité de la recherche sera beaucoup plus élevée et nécessite de concevoir des procédures efficaces, pour son exploration.

Une approche efficace consiste à réguler la taille de voisinage durant la recherche [14,19], en fonction de la qualité de solutions obtenues, afin de sélectionner la taille adéquate du voisinage (**Figure** 5).

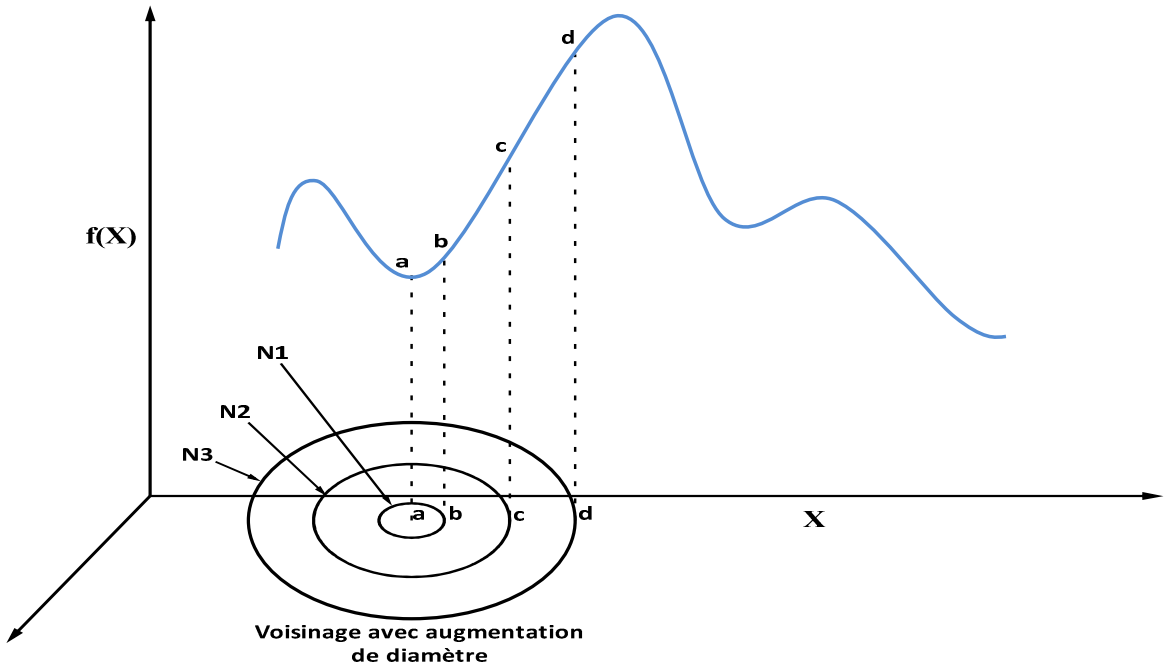

Figure 5 : Voisinage variable avec plusieurs diamètres.

4. Recherche à voisinage variable

La recherche à voisinage variable ou (VNS pour Variable Neighborhood Search) est une méthode proposée par P.Hansen et N. Mladenovic [20, 21]. L'idée de base consiste, à explorer successivement un ensemble de voisinages préalablement définis, afin de trouver de meilleures solutions.

La RVV explore systématiquement, d'une manière aléatoire, un ensemble de voisinage différents, pour obtenir plusieurs optimums locaux et d'échapper à des *bassins d'attraction*. La RVV exploite, également le fait que l'utilisation de différents voisinages, peut générer plusieurs optimums locaux et que l'optimum global, n'est que l'optimum local d'un

voisinage particulier (**Figure 6**). En effet, l'utilisation de différentes structures de voisinage génère différents paysages de l'espace de recherche.

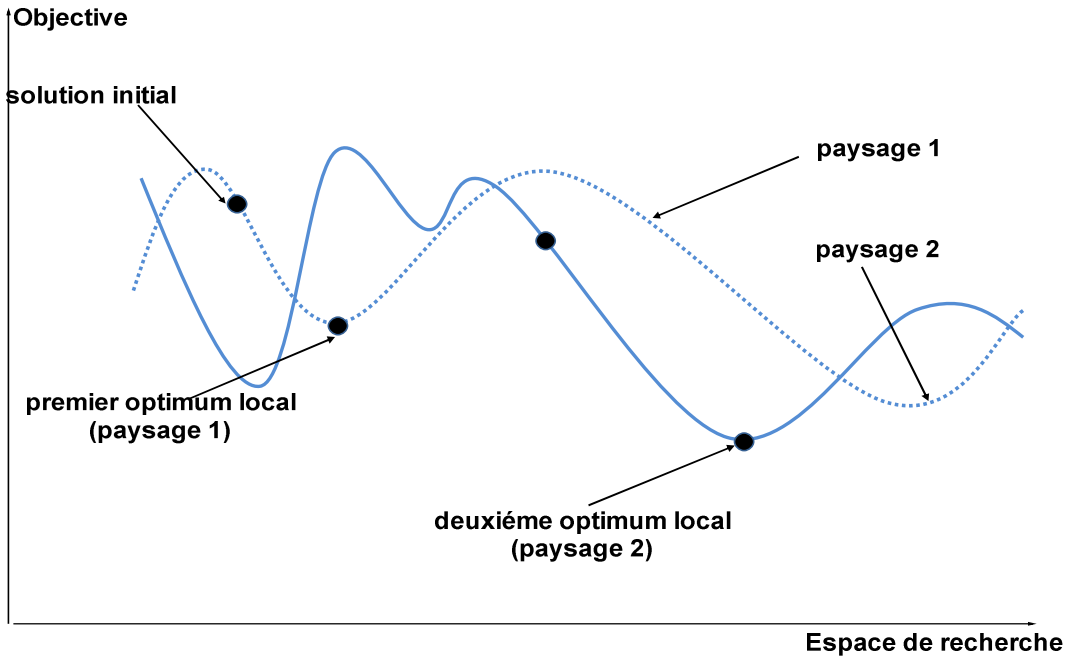

Figure 6 : *Voisinage Variable utilisant* deux structures de voisinage

4.1.Descente à Voisinage Variable

La Descente à Voisinage Variable est une version déterministe du *RVV* [13].

La *DVV* utilise des voisinages successifs, en descente vers un optimum local. On doit d'abord définir un ensemble de structures de voisinage noté N_L $(L = 1,..., L_{max})$.

Soit N_1 le premier voisinage à utiliser et $\boldsymbol{x}$ la solution initiale. Si une amélioration de la solution $\boldsymbol{x}$ n'est pas possible dans son voisinage courant $N_i(x)$, alors la structure de voisinage changera de N_i vers N_{i+1}. Par contre, si une amélioration de la solution courante $\boldsymbol{x}$ est enregistrée, on retournerait à la structure initiale $N_1(x)$, afin de relancer la recherche (**Figure 7**).

Cette stratégie serait plus efficace si les différents voisinages utilisées sont complémentaires, dans le sens où un optimum local pour un voisinage N_i ne le serait pas pour un autre voisinage N_j.

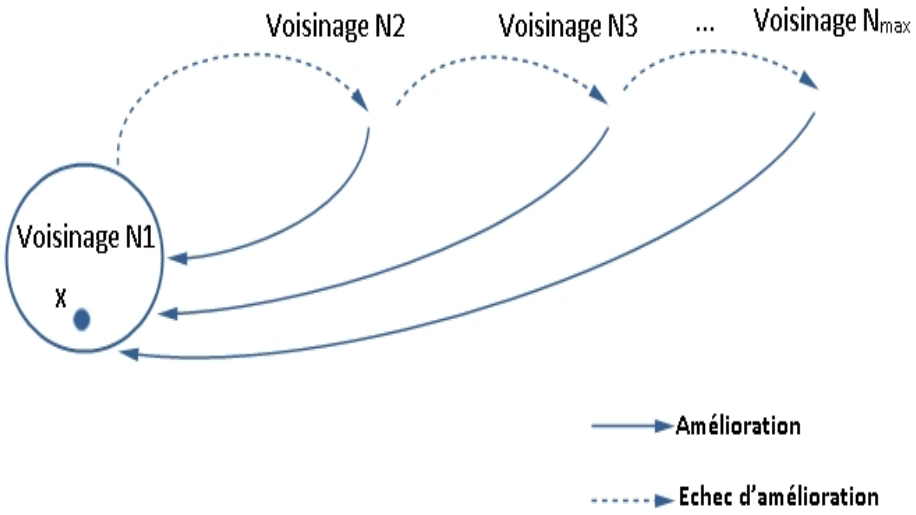

Figure 7 : Le principe de la *Descente à Voisinage Variable*.

Le pseudo code suivant éclaire le principe de *DVV*:

Algorithme 2 : Principe d'une *Descente à Voisinage Variable*.

Entrée: un ensemble de structures de voisinage N_i pour $i = 1,..., i_{max}$.
$x=x_0$; /*générer la solution initiale*/
$i=1$;
Répéter
  Trouver le meilleur voisin x' de x dans $N_i(x)$;
  SI $f(x')<f(x)$ **alors** $x=x'$; $i=1$;
  Sinon
  $i=i+1$;
Jusqu'à ce que $i \leq i_{max}$
Sortie: meilleure solution trouvée.

La conception de la *DVV* est principalement liée à la structure des voisinages utilisés et de l'ordre de leur sélection. La complexité de l'exploration et de l'évaluation des voisinages doivent être prise en compte. Plus le voisinage est large, plus il prend du temps pour son évaluation. Quant à l'ordre de sélection de voisinage à utiliser, la stratégie la plus commune, est de les classer suivant l'ordre croissant de leur complexité (Ex : la taille de voisinage $|N_i(x)|$).

4.2. Recherche à voisinage variable générale

RVVG est une méthode stochastique [13, 20, 21, 22], un ensemble de structures de voisinage N_k $(k = 1,..., N)$ doit être préalablement définis. Ensuite à chaque itération la méthode passera par trois étapes: choix d'une solution (shaking phase), recherche locale, changement de voisinage.

A chaque itération, une solution x' est générée aléatoirement du voisinage courant N_k, ensuite une recherche locale est appliquée à la solution x' pour générer une solution x'', une solution courante n'est pas remplacée par un nouveau optimum local x'' sauf s'il est meilleur $(f(x'') < f(x))$. La même recherche locale est donc relancée à partir de la solution x'' dans le voisinage N_1. Si aucune amélioration n'est trouvée $(f(x'') >= f(x))$, la méthode sélectionnerait le voisinage suivant N_{k+1}, génère aléatoirement une nouvelle solution dans ce voisinage, et tente de l'améliorer.

Le pseudo code suivant éclaire le principe de *RVVG*:

Algorithme 3 : Principe de *RVVG*.

Entrée: un ensemble de structures de voisinage N_i pour $i = 1,..., i_{max}$.
  $x=x_0$; /*générer la solution initiale*/
Répéter
  $i=1$;
  Répéter
    Sélectionner une solution aléatoire x' dans $N_i(x)$;
    x''=recherche locale(x');
    If $f(x'')<f(x)$ **alors**
      $x=x''$;
      Continuer la recherché avec N_1 ;
      $i=1$;
    Sinon
      $i=i+1$;
  Jusqu'à ce que $i \leq i_{max}$
Jusqu'à ce que le critère d'arrêt soit satisfait
Sortie: meilleure solution trouvée.

Afin d'améliorer la recherche locale pour le *RVVG*, nous pouvons la remplacer par *DVV*.

Le pseudo code suivant explique le principe :

Algorithme 4 : Principe de *RVVG amélioré*

Entée:
un ensemble de structures de voisinage N_k pour $i = 1,..., k_{max}$. /*pour Shaking phase*/
un ensemble de structures de voisinage N_i pour $i = 1,..., i_{max}$. /*pour recherche locale*/
$=x_0$; /*générer la solution initiale*/
Répéter
  Pour $k=1$ à k_{max};
    Shaking **:** Sélectionner une solution aléatoire x' dans $N_k(x)$;
    Recherche locale de la Descente à Voisinage Variable
    Pour $i=1$ à i_{max};
      Chercher le meilleur voisin x'' de x' dans $N_i(x')$
      x''=recherche locale(x');
      $i=1$;
      Si $f(x'')<f(x)$ **alors**
      $x'=x''$;
      Continuer la recherché avec N_1 ;
      $i=1$;
      Sinon
       $i=i+1$;
      Si l'optimum local est meilleur que x **alors**
       $x=x''$;
      Continuer la recherché avec N_1 (k=1);
     Sinon
     $k=k+1$;
Jusqu'à ce que le critère d'arrêt soit satisfait
Sortie: meilleure solution trouvée.

La conception d'une méthode VVRG est principalement liée à la structure des voisinages utilisées et du choix de la solution initial (shaking phase). Généralement, des voisinages imbriqués sont utilisés, où chaque voisinage $N_k(x)$ contient le précédent $N_{k-1}(x)$:

$$N_1(x) \subset N_2(x) \subset \cdots \subset N_k(x), \forall x \in S$$

Afin d'améliorer la méthode et combiner entre l'intensification et la diversification, un compromis doit être trouvé. En effet, une concentration sur la recherche locale va générer des meilleurs optimums locaux (plus d'intensification), tandis qu'une concentration sur le choix de la solution initiale (shaking phase) mènera à des régions potentiellement favorites dans l'espace de recherche (plus de diversification).

5. Conclusion

La notion de voisinage est sans doute un des principes le plus utilisé lors conception de l'élaboration d'une métaheuristique. La définition de la structure de voisinage est une étape clef, dans la mise au point d'un algorithme de recherche locale.

Dans ce chapitre, nous avons présenté des méthodes et des techniques qui permettant de rendre le voisinage réactif et adaptatif, ces techniques utilise méthodiquement plusieurs types de voisinages en fonction de l'espace de recherche visité, ce mécanisme permet de diversifier l'exploration de l'espace des solutions afin d'accéder à un plus grand nombre de régions intéressantes.

CHAPITRE IV : Recherche Tabou et Réactivité

1. Introduction

La recherche tabou est une métaheuristique de recherche locale très populaire, elle a connu un large succès pour résoudre divers problèmes d'optimisation combinatoires.

Nous présentons dans ce chapitre les principes de cette métaheuristique, nous présentons aussi quelques techniques qui permettent de rendre adaptatif et réactif, l'un de ses paramètres le plus, à savoir la taille de la liste tabou.

2. Recherche Tabou

2.1. Principe

La méthode Tabou est une technique de recherche dont les principes ont été proposés pour la première fois par Fred Glover dans les années 80 [23], puis elle est devenue très classique en optimisation combinatoire. Elle se distingue des méthodes de recherche locale simples par le recours à un historique des solutions visitées, qui aide de s'extraire d'un minimum local, mais, pour éviter d'y retomber périodiquement, certaines solutions sont rendues taboues. On parle alors de restrictions taboues. Ces restrictions peuvent exceptionnellement être levées grâce aux critères d'aspiration. Pour améliorer encore plus la puissance de la recherche, des techniques d'intensification et de diversification sont intégrées. L'intensification mémorise des propriétés favorables des meilleures configurations afin de les utiliser ultérieurement. Ainsi, elle permet d'explorer des régions de l'espace de recherche proches de celles caractérisant des configurations à priori de bonne qualité visitées antérieurement. La diversification cherche à diriger la recherche vers des configurations inexplorées. Elle permet alors de générer des configurations qui diffèrent de manière significative de celles rencontrées auparavant.

Si on nomme *NT(s)* toutes les solutions qui ne sont pas tabou ainsi que celles qui le sont mais dont le statut tabou est levé en raison des critères d'aspiration, l'algorithme général peut se représenter avec le pseudo-code suivant :

Algorithme 5 : Principe de *La Recherche Tabou*.

NT (s) = { $s' \in N(s)$ tel que $s' \notin T$ ou $f(s') < f(s^*)$ }

Procédure *methode_Tabou* (solution initiale *s*)

Poser T ← ∅ et $s^* \leftarrow s$;

Répéter

Choisir *s'* qui minimise *f(s')* dans $N_T(s)$

Si $f(s') < f(s^*)$ **alors** poser $s^* \leftarrow s'$;

Poser $s \leftarrow s'$ et mettre à jour T ;
Jusqu'à ce que le critère de terminaison soit satisfait
Fin

Après avoir introduit le principe de base de la recherche tabou, nous allons discuter les différents types de mémoire utilisées dans la *recherche tabou* (**Tableau1**):

Type de mémoire	**rôle**	**forme**
court terme	Eviter des cycles	Liste tabou
moyen terme	intensification	Mémoire récente
long terme	diversification	Mémoire de fréquence

Tableau 2 : les types de mémoires pour *recherche tabou*.

2.2. La mémoire à court terme

Le rôle de la mémoire à court terme est de stocker l'histoire des mouvements récents pour prévenir l'apparition des cycles [13,23]. La représentation simple consiste à enregistrer toutes les solutions visitées durant la recherche. Cette représentation assure le manque de cycles, mais elle est rarement utilisée, car elle produit une grande complexité du stockage de données et de temps de calcul. La première amélioration est de réduire la complexité de l'algorithme et de limiter la taille de la liste tabou. Si la liste contient les **k** dernies mouvements utilisés, la recherche tabou empêche un cycle de taille **k**. L'utilisation de table de hachage peut également réduire la complexité de l'algorithme et la manipulation de la liste tabou.

La liste tabou est composée des mouvements qui seront interdits dans les prochaines itérations. Elle est relative à la structure du voisinage de problème traité. Si un mouvement **m** est appliqué sur une solution $\mathbf{S_i}$ pour générer une solution $\mathbf{S_j}$ ($\mathbf{S_j=S_i \oplus m}$), alors le mouvement **m** est stocké dans la liste, et serait interdit pour un nombre d'itérations données, relatif à la taille de la liste.

La taille de la liste tabou est un paramètre critique, qui a un impact important sur les performances de la recherche tabou. A chaque itération, le dernier mouvement est ajouté dans la liste, alors que le mouvement le plus ancien est retiré. Plus la taille de la liste tabou est petite, plus est importante la probabilité d'apparition des cycles. La taille élevée de la liste taboue encourage la diversification de la recherche, puisque que le nombre des mouvements interdits est important. Un compromis doit être fait pour trouver la taille convenable au problème traité.

La taille de la liste tabou peut prendre différentes formes:

- ***Statique:*** En général, la taille de la liste tabou est fixe. Elle dépend de la taille de l'instance du problème traité et en particulier la taille du voisinage. Il n'ya pas de taille optimale de la liste tabou pour tous les problèmes, ou toutes les instances d'un problème donné. Par ailleurs, la valeur optimale peut varier pendant la progression de la recherche. Une taille variable de la liste tabou peut surmonter ce problème.
- ***Dynamique:*** La taille de la liste tabou peut changer pendant la recherche sans tenir compte des informations de l'espace de recherche.
- ***Adaptative (réactive):*** la taille de la liste tabou est modifié en fonction de la progression de la recherche.

La liste tabou pourrait être aussi considérée comme un moyen pour l'intensification, si sa taille est réduite, ou pour la diversification, si taille est importante.

2.3. La mémoire à moyen terme

Le rôle de l'intensification est d'exploiter les informations des meilleures solutions trouvées (solutions élites), afin de guider la recherche dans les régions les plus prometteuses de l'espace de recherche [10,17]. Ces informations seront stockées dans une mémoire à moyen terme. L'idée consiste à extraire les caractéristiques des meilleures solutions trouvées et d'intensifier la recherche autour des solutions qui partage ces mêmes caractéristiques.

La représentation principale utilisée pour *la mémoire à moyen* terme est la *mémoire de récence*. Tout d'abord, les composants associés à une solution doivent être définis, ce qui est une tâche relative au problème traité. Ensuite La *mémoire de récence* mémorise pour chaque composant, le nombre d'itérations successives dans l'espace des solutions visitées. Enfin le processus d'intensification est lancé après une période définie, ou après un certain nombre d'itérations sans amélioration.

L'intensification de la recherche dans une région donnée de l'espace de recherche n'est pas toujours utile. L'efficacité de l'intensification dépend de l'ensemble de l'espace de recherche. Si l'espace de recherche est composé de nombreux bassins d'attraction, alors une recherche tabou simple, sans un mécanisme d'intensification, serait plus efficace pour la recherche dans chaque bassin d'attraction ; intensifier la recherche dans chaque bassin est inutile.

2.4. La mémoire à long terme

Le principe de la mémoire à longue terme est introduit pour encourager la diversification de la recherche [13,23]. Son rôle est de forcer la recherche d'inspecter des régions inexplorées de l'espace de recherche.

La représentation principale utilisée pour la mémoire à long terme est la mémoire de fréquence.

Comme dans la mémoire à moyen terme, les composants associés à une solution doivent être d'abord définie. La mémoire de fréquence mémorise pour chaque composant, le nombre de fois qu'il est présent pour toutes les solutions visitées.

Le processus de diversification peut être appliquée périodiquement, ou déclenché après un nombre d'itérations sans amélioration. On distingue trois stratégies de diversification:

- ***Diversification relancé:*** Cette stratégie consiste à appliquer des modifications sur la solution courante ou sur la meilleure solution obtenue, puis relancer une nouvelle fois la recherche.
- ***La diversification continue:*** pour encourager la diversification, un mécanisme qui consiste d'introduire la fréquence d'occurrence des mouvements pour l'évaluation des solutions courantes. Les mouvements les plus fréquemment appliqués seront pénalisés.
- ***L'oscillation stratégique :*** Introduite par Glover en 1989, l'oscillation stratégique consiste à modifier la fonction d'objectif d'une manière qu'elle accepte des solutions intermédiaires, qui sont irréalisables. Cette stratégie guidera la recherche vers des solutions irréalisables et ensuite revenir vers des solutions réalisable.

Comme l'intensification, la diversification n'est pas toujours utile. Elle dépend de la structure du problème d'optimisation traité et de l'ensemble de l'espace de recherche. Si toutes les bonnes solutions sont localisées dans une même région de l'espace de recherche, la diversification vers d'autres régions de l'espace serait inutile.

Les temps de recherche affectés à la diversification et l'intensification des composants de la recherche tabou, doivent être soigneusement réglés en fonction des caractéristiques de la structure du paysage associés au problème.

La recherche tabou a été appliquée avec succès à de nombreux problèmes d'optimisation. La liste tabou, la mémoire à moyen terme, et la mémoire à long terme doivent être conçues en fonction des caractéristiques du problème d'optimisation à résoudre. Ce n'est pas une tâche facile pour certains problèmes d'optimisation. Par ailleurs, la recherche tabou peut être très sensible à certains paramètres, tels que la taille de la liste tabou.

3. Reactivité sur la liste tabou

La résolution des problèmes d'optimisation combinatoire par la *Recherche Tabou* posent certains problèmes, qui méritent d'avoir beaucoup de réflexion [14] :

- La détermination de la taille appropriée de la *liste tabou* pour les différentes instances traitées.
- La robustesse de la méthode pour un large éventail de problèmes d'optimisation.
- L'adoption et la sélection des stratégies avec un minimum de complexité pour l'exploration et le traitement de l'historique de recherche.

Nous examinons dans ce qui suit ces trois problèmes en introduisant un mécanisme réactif à la méthode tabou.

3.1. Auto-sélection de la taille de la liste tabou

Dans la méthode *tabou réactive* [14,24] la taille de la *liste tabou* ***T*** (le nombre des mouvements interdits) est déterminée grâce à un feedback de l'historique de recherche. ***T*** est fixé à un au début, ***T*** n'augmente seulement que, quand il est évident qu'une diversification est nécessaire, dans le cas contraire ***T*** est diminué. Précisément : une diversification est nécessaire lorsque des répétitions des configurations précédemment visitées soient signalées. Toutes les configurations visitées pendant la recherche sont stockées en mémoire. La méthode vérifie à chaque itération si la configuration courante se trouve dans la mémoire et elle réagit en conséquence (***T*** est augmenté si une configuration est répétée, diminué si aucune répétitions n'est signalé pendant une période suffisamment longue).

La taille ***T*** n'est pas fixée pendant la recherche, mais elle est déterminée de façon dynamique en fonction de la structure de l'espace de recherche. Ceci est particulièrement pertinent pour des problèmes non homogènes où les propriétés de l'espace de recherche

varient considérablement dans des différentes régions (dans ces cas, une taille ***T*** fixe serait inapproprié, du coup la méthode serait plus adapté à un large éventail de problèmes).

3.2. Diversification radical et bassin d'attractions

Le mécanisme utilisé dans la méthode tabou qui consiste d'introduire une liste des mouvements interdits, pour échapper des optimums locaux, n'est pas suffisant pour éviter des cycles longs [14,24]. Même si des cycles de longueur variable sont évités avec le mécanisme réactif, ce dernier ne suffit, malheureusement pas, de garantir que la trajectoire de recherche ne soit pas confinée dans une région limitée de l'espace de recherche. La présence des bassins d'attractions seraient possible (la trajectoire est confinée dans une portion limitée de l'espace de recherche, bien que des cycles ne sont pas détectés).

Pour augmenter la robustesse de la méthode une diversification seconde plus radicale est nécessaire. Les diversifications radicales seront déclenchées quand plusieurs configurations se répètent assez souvent, un mécanisme simple de diversification consiste à appliquer certain mouvement aléatoire sur la configuration courante (un mécanisme qui peut déplacer la trajectoire de la recherche loin de la région de recherche courante).

Avec des diversifications aléatoires, nous pouvons facilement obtenir une convergence asymptotique de la Recherche Tabou Réactive (si la probabilité pour atteindre un point après une diversification est différente de zéro pour tous les points, éventuellement tous les points pourraient être visités, y compris les optimums globaux).

3.3. Stockage et utilisation de l'espace de recherche

L'élément fondamental de la méthode tabou est l'utilisation flexible de la mémoire qui incarne la création et l'exploitation de la structure de données pour tirer profit de l'histoire de recherche.

Dans cette section, nous présentons quelques structures ainsi que des stratégies pour stocker, explorer et récupérer rapidement l'information de l'historique de l'espace de recherche **(Figure 8).**

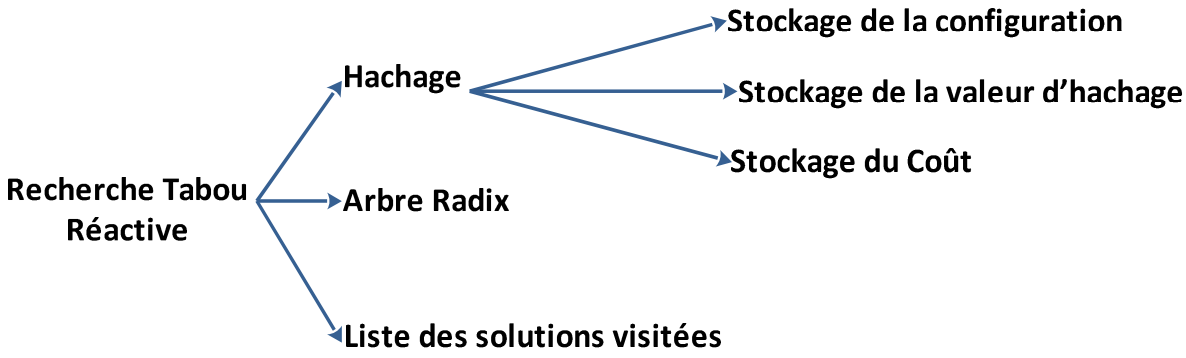

Figure 8 : différentes structures de données possibles pour la Recherche Tabou Réactive.

Les informations qu'on doit mémoriser sont par exemple, les configurations, les valeurs de la fonction objective et les itérations correspondantes. Le choix des stratégies dépend principalement de l'espace-temps, la complexité et de la quantité de données stockées à chaque itération.

3.3.1. La Méthode d'Elimination Inverse

La Méthode d'Elimination Inverse (MEI) [23,25] est une stratégie dynamique qui détermine l'état des mouvements s'ils sont interdis sans faire recours à la liste tabou, mais en tenant compte des relations logiques existantes entre les séquences stockées, ces relations permettent de déterminer à l'avance si une configuration courante peut produire un cycle. Ainsi, elle va générer des restrictions sur des mouvements, pour empêcher la recherche de revenir à des solutions précédemment visitées.

La méthode d'élimination inverse fonctionne comme suit. À chaque itération, une liste ***(L)*** qui contient des informations sur l'historique de recherche, est balayée pour déterminer tous les mouvements qui doivent avoir un statut tabou, pour éviter à la recherche de mener ver des solutions déjà visitées.

En balayant ***(L)***, un ensemble appelé séquence d'annulation résiduelle ***(SAR)*** est construit pour déterminer la différence entre la solution courante et toutes les solutions incluses dans la liste ***(L)***.

Algorithme 6 : principe de la méthode MEI
Entrée: λ, *caractéristique I, mouvements a.*
$s=s_0$; /*générer la solution initiale*/
Pour (touts *a*) **Faire**

StatusTabou(a) ←*inactive* ;
Fin Pour
$SAR \leftarrow \varnothing$
Répéter
Si($\overline{L(i)} \in SAR$)
$SAR \leftarrow SAR \setminus \overline{L(i)}$;
Si non
$SAR \leftarrow SAR \cup \overline{L(i)}$;
Fin Si
Si (|*SAR*|≤ *t*)
Pour (touts $a \in SAR$) **Faire**
StatusTabou(a) ←*inactive ;*
Fin Pour
Jusqu'à ce que le critère d'arrêt soit satisfait
Sortie: Meilleur solution trouvée.

A partir d'un ***SAR*** vide, et à chaque étape de balayage de ***(L)***, deux cas peuvent se produire : Soit le complément d'un attribut traité n'est pas au ***SAR*** et donc la différence entre la solution courante et les solutions précédemment visités augmente, ou l'attribut annule son complément dans ***SAR***, dans ce cas, la solution courante est plus proche de l'une des solutions visités dans l'historique de recherche contenus dans ***(L)***. Si ***SAR*** contient ***t*** attributs, le statut tabou du complément de ces attributs devient actif pour éviter la réapparition d'une solution déjà explorée. Bien qu'une valeur de ***t*** égal à un, soit suffisante pour éviter de revisiter une solution, la valeur ***t*** pourrait être augmenté pour encourager la diversification.

Soit (L) une liste qui contient touts les mouvements : ajouter (a) ou enlever ($\overline{a}$) un attribut.

Les attributs peuvent être considérés comme des éléments pour un *problème de sac à dos*, des nœuds ou arêtes pour un graphe, etc.

Soit (1, 2, 5, 6) une solution initiale et ***(L)*** $(\overline{1}, \overline{6}, 7,3,1, \overline{5}, 6,4, \overline{6}, 5)$ après 10 itérations, l'étape de balayage détermine qu'elles sont les attributs qui doivent avoir un statut tabou actif pour empêcher la recherche de revenir vers des solutions visités durant les 10 premiers itérations.

Le ***Tableau 2*** illustre le mécanisme de *La Méthode d'Elimination Inverse*, en montrant la composition du ***SAR*** à chaque étape de balayage (le paramètre ***t*** est pris à 1).

Itération	*SAR*	Tabou-Actif
1	5	$\overline{5}$
2	$\overline{6}, 5$	-
3	$4, \overline{6}, 5$	-
4	$4,5$	-
5	4	$\overline{4}$
6	$1,4$	-
7	$3,1,4$	-
8	$7,3,1,4$	-
9	$\overline{6}, 7,3,1,4$	-
10	$\overline{6}, 7,3,4$	-

Tableau 3: illustration du concept de *MEI.*

L'effort requis par *la Méthode d'Elimination Inverse,* pour attribuer un statut tabou aux attributs appropries, augmente clairement avec l'augmentation du nombre d'itérations, par conséquent, les principaux recherches dans ce domaine sont consacrées à l'élaboration des mécanismes visant à réduire le nombre de calculs associés à cette tache. Un autre concept important, qui doit être pris en considération, est l'ajustement du paramètre ***t***, pour équilibrer entre l'intensification et la diversification.

3.3.2. La technique de hachage

La technique de hachage est standard dans l'informatique [26,27], c'est une structure de données qui permet une association clé-élément ; On accède à chaque élément de la table via sa clé. Il s'agit d'un tableau ne comportant pas d'ordre (un tableau est indexé par des valeurs de la fonction objectif). L'accès à un élément se fait en transformant la clé, en une valeur de hachage (ou simplement hachage) par l'intermédiaire d'une fonction de hachage. Le hachage est un nombre, qui permet la localisation des éléments dans le tableau, typiquement le hachage est l'index de l'élément dans le tableau **(Figure9).**

Le fait de créer un hash, à partir d'une clé, peut engendrer un problème de « collision », c'est-à dire qu'à partir de deux clés différentes, la fonction de hachage pourrait renvoyer la même valeur de hash, et par conséquent donner accès à la même position dans le « tableau ». Pour minimiser les risques de collisions, il faut donc choisir soigneusement sa fonction de hachage.

Les tables de hachage permettent un accès en ***O(1)*** en moyenne, quel que soit le nombre d'éléments dans la table. Toutefois, le temps d'accès dans le pire des cas, peut être de ***O(n)*** tel que ***n*** est le nombre de collisions pour une entrée. Les tables de hachage sont surtout utiles, lorsque le nombre d'entrées est très important.

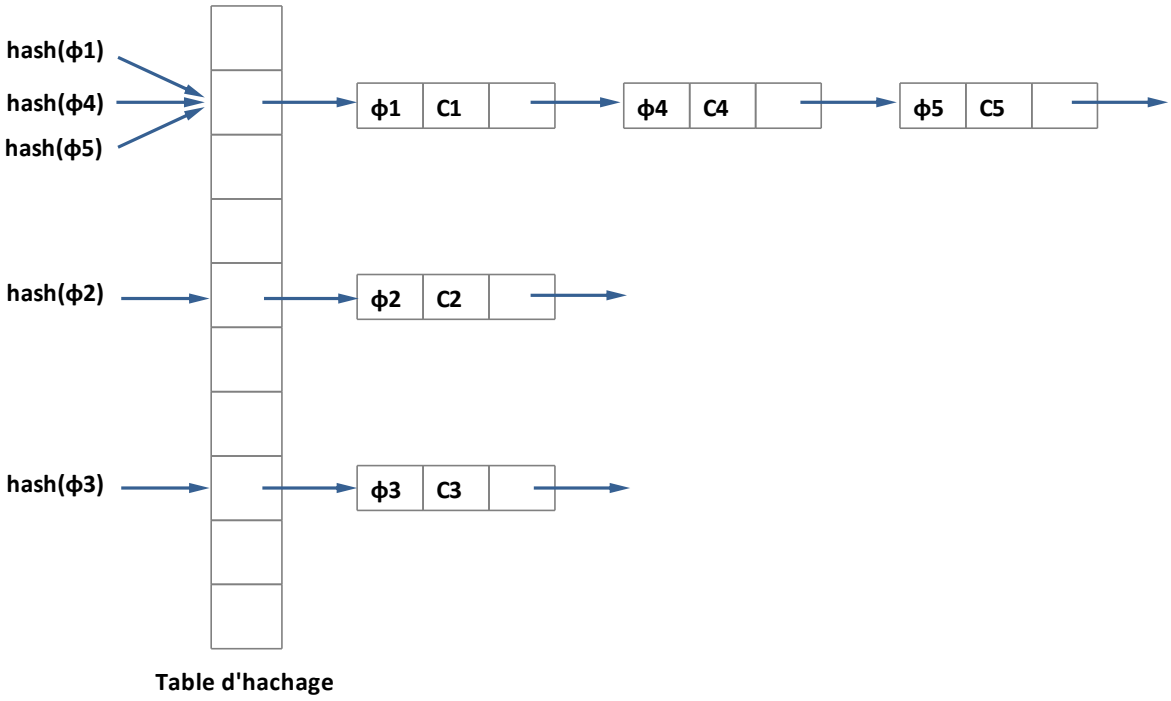

Figure 9 : structure de Hachage pour la *recherche tabou réactive*.

a. Choix de la fonction de hachage

Une bonne fonction de hachage est importante pour les performances. Les collisions étant en général, résolues par des méthodes de recherche linéaire, une mauvaise fonction de hachage engendre beaucoup de collisions, par conséquence elle va fortement dégrader la rapidité de la recherche. D'autre part, il est préférable que la fonction de hachage ne soit pas de complexité élevée.

b. Résolution des collisions

Lorsque deux clés ont la même valeur de hachage, ces clés ne peuvent être stockées à la même position, on doit alors employer une stratégie de résolution des collisions.

De nombreuses stratégies de résolution des collisions, existent mais la plus connue et utilisée est le chainage.

Le chainage est une méthode simple. Chaque case de la table est en fait une liste chaînée des clés qui ont la même valeur de hachage. Une fois la case trouvée, la recherche est alors linéaire. Dans le pire des cas, où la fonction de hachage renvoie toujours la même valeur de hachage, quelle que soit la clé, la table de hachage devient alors une liste chaînée, et le temps de recherche ***O(n)***.

3.3.3. La technique de hachage combiné à l'arbre rouge et noir

Puisque la *Recherche Tabou* est basé sur la recherche locale, la configuration $X^{(t+1)}$ ne diffère de la configuration $X^{(t)}$ que par l'addition ou la soustraction, d'un indice unique (pour le PVC ; un changement de deux ville est procédé, mais tout l'itinéraire est gardé). Il est donc préférable de concevoir une technique plus efficace, qui sert à stocker les états des configurations, d'une manière enchaîné (relative aux configurations passées) que de mémoriser les états des configurations d'une manière arbitraire. L'intégration de la technique de l'arbre rouge et noir qui est une technique avancée, qui semble alors, une bonne piste à suivre.

a. Arbre rouge et noir

Un *arbre rouge et noir* [28,29] est un arbre binaire de recherche où chaque nœud est de couleur rouge ou noire de telle sorte que

1. La racine est noire.
2. Le parent d'un nœud rouge est noir.
3. Le chemin de chaque feuille à la racine contient le même nombre de nœuds noirs.

Ces contraintes impliquent une propriété importante des arbres bicolores : le chemin le plus long possible d'une racine à une feuille ne peut être deux fois plus long que le plus petit possible. On a ainsi un arbre presque équilibré. Comme les opérations d'insertion, de recherche et de suppression requièrent, dans le pire des cas, un temps proportionnel à la taille de l'arbre, les arbres *rouge et noir* restent efficaces, contrairement aux arbres binaires, de recherche ordinaire.

Pour comprendre comment ces contraintes garantissent la propriété ci-dessus, il suffit de s'apercevoir qu'aucun chemin ne peut avoir deux nœuds rouges consécutifs à cause de la

propriété 2. Le plus petit chemin théorique, de la racine à une feuille, ne contient alors que des nœuds noirs, tandis que le plus grand, s'alterne entre les nœuds rouges et noirs. Et comme chacun de ces chemins contient le même nombre de nœuds noirs, le plus grand chemin ne peut être deux fois plus grand que le plus petit (d'après la propriété 3).

La propriété 1 n'est pas nécessaire. Les seuls cas où la racine pourrait devenir rouge étant les deux cas où sa couleur n'a pas d'importance : Soit la racine est le seul nœud, soit elle possède deux fils noirs. Cette propriété est ajoutée uniquement pour visualiser plus rapidement l'isomorphisme avec les arbres 1-2-3 : Chaque nœud noir et ses éventuels fils rouges représentent un nœud d'arbre 1-2-3.

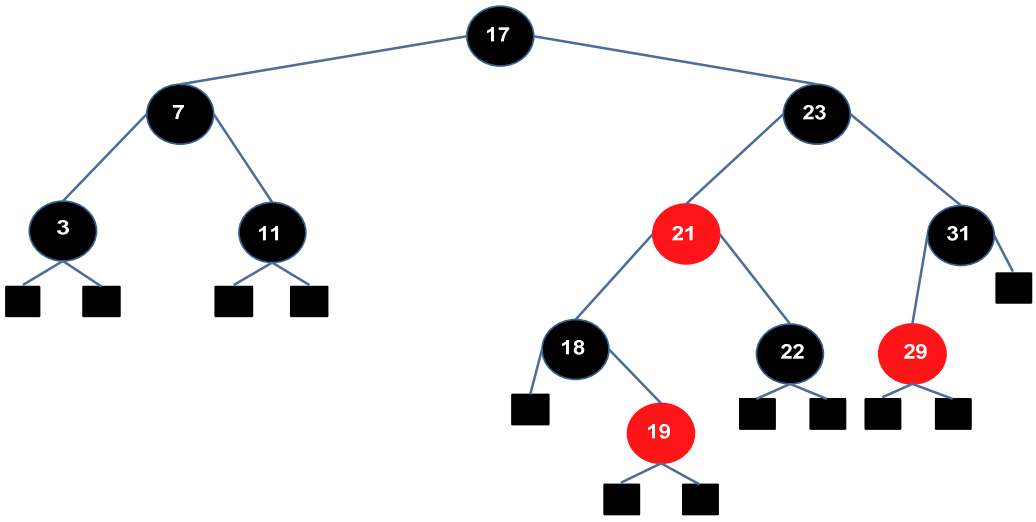

Figure 10 : exemple d'un arbre rouge et noir

b. Fonctionnement

La méthode permet d'économiser un espace mémoire considérable en proposant un espace linéaire. L'approche évite de copier le chemin d'accès complet à chaque fois qu'une mise à jour est produite. Pour ce but, chaque nœud contient un pointeur supplémentaire avec un indicateur de temps (en plus les pointeurs ver les nœuds gauches et droits). Si le pointeur supplémentaire est disponible lors d'un d'ajout d'un pointeur vers un nœud, alors il serait utilisé et le temps d'accès est enregistré. Si le pointeur supplémentaire est déjà utilisé, le nœud serait copié, fixant les pointeurs initiaux gauches et droits de la copie à leurs dernières valeurs. En plus, le pointeur vers la copie est stocké dans le dernier parent du nœud copié. Si le parent a déjà utilisé le pointeur supplémentaire, le nœud parent, lui aussi, serait copié. Ainsi

l'opération se prolifère dans les nœuds précédents jusqu'à ce que la racine soit copiée où un nœud avec pointeur supplémentaire libre est rencontré.

Chercher une donnée dans un temps ***t*** est facile: commençant par une racine appropriée, si le pointeur supplémentaire d'un nœud est utilisé, le pointeur qui suit ce nœud est déterminé en examinant l'indicateur de temps du pointeur supplémentaire et se pointe ver celui-ci si et seulement si, l'indicateur de temps n'est pas plus grand que ***t***. Sinon, si le pointeur supplémentaire n'est pas utilisé, les pointeurs gauche et droite seraient considérés.

La direction des pointeurs (gauche et droite) ne doivent pas être stockés: en basant sur les propriétés de l'arbre de recherche, ils peuvent être obtenus en comparant les indices des nœuds enfants avec celle du nœud courant. En plus, les couleurs ne sont nécessaires que, pour la version de l'arbre la plus récente.

Dans le pire des cas, La complexité de l'insertion, et de la suppression, reste ***O(log L)***, mais le résultat le plus important, est que le coût de l'espace est réduit à ***O(1)*** lors d'une modification.

c. Hachage et arbre rouge et noir

Dans le contexte d'une métaheuristique basant sur l'historique des configurations visitées. Nous nous intéressons à vérifier à chaque itération, si une configuration a déjà été rencontré dans l'histoire de la recherche stockées.

Le moyen le plus pratique est de concevoir une structure de données, qui combine la technique *de hachage* et l'arbre *rouge et noir* [14,24]. Chaque configuration ***X*** est associée à un index (obtenue par une fonction de hachage) dans la table de hachage, avec une complexité d'accès de ***O(1)*** (**Figure 11**). Les collisions sont résolues par une liste chainée : chaque entrée de table contient une liste chainée qui pointe vers la racine de l'arbre rouge-noir et les données nécessaires à la recherche (temps, le nombre de répétitions).

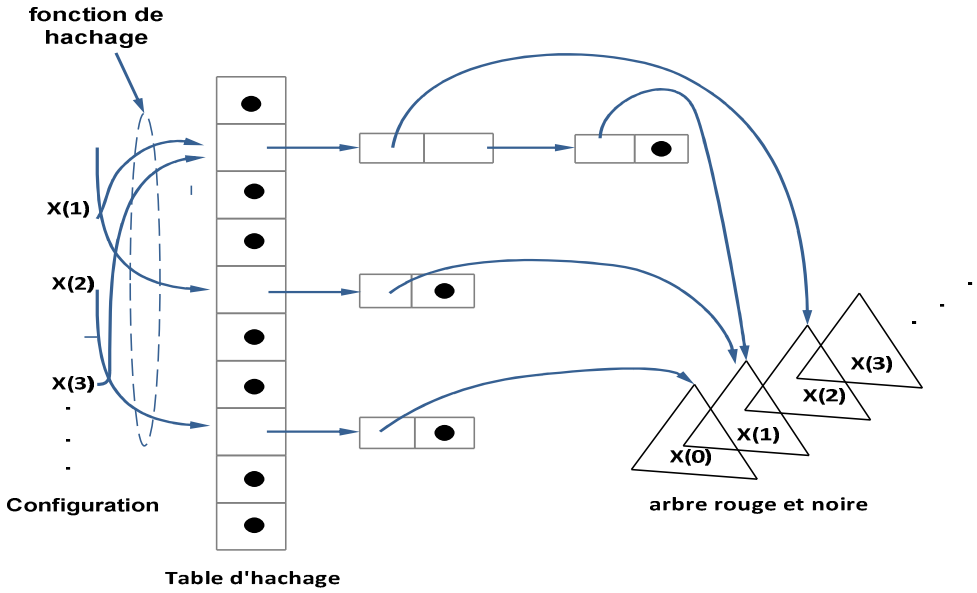

Figure 11 : structure de données avec hachage combinée à l'arbre rouge et noir : un pointeur ver la racine d'un arbre rouge et noire d'une configuration $X^{(t)}$ est stocke dans une liste chainée

Dès qu'une configuration $X^{(t)}$ est générée, l'arbre rouge-noir correspondant est modifié par des opérations d'insertions et de suppressions. La recherche se fait comme suit : la valeur de l'index d'entrée de table est calculée à partir d'une fonction de hachage appliquée à la configuration $X^{(t)}$. Pour chaque élément de la liste chaînée, l'ensemble des arbres rouge et noir est balayé puis comparé avec la configuration $X^{(t)}$. Si une équivalence est signalée, un pointeur vers l'élément de la liste liée est retourné. Sinon, après que la liste entière est balayée sans succès, un pointeur NULL est retourné.

Dans le cas ou la configuration $X^{(t)}$ n'est pas trouvée, un nouvel élément est créé dans la table de hachage avec un pointeur vers la racine de l'arbre rouge et noir appropriée. Sinon, seulement les propriétés de la configuration $X^{(t)}$ trouvées sont modifiées (temps de la dernière visite, compteur de répétition).

En se basant sur ce qui est cité au dessus, et en supposant que la taille de la table de hachage est égale au nombre d'itérations exécutées, il est facile de conclure que chaque itération de la recherche tabou réactive nécessite en moyen un temps ***O(L)*** et un espace réduit ***O(1)*** pour le stockage et la récupération des configurations dans l'espace de recherche passé, afin de détecter les mouvements interdits.

La combinaison de table de hachage et l'arbre rouge et noir nécessite ***O(1)*** d'espace. Le temps pour une modification d'une configuration courante $X^{(t)}$ est ***O(log L)*** dans le pire

des cas, le temps moyen pour la recherche et la mise à jour dans la table de hachage est ***O(1)***. Le temps de recherche d'une configuration $X^{(t)}$ dans la liste chainé de l'arbre rouge et noir est, dans le pire des cas, ***O(L)***. Parce que ***Ω(L)*** est le temps nécessaire, lors de l'évaluation de voisinage pour calculer les valeurs objectif.

4. Conclusion

La recherche tabou est une des métaheuristique les plus connue pour résoudre divers problèmes d'optimisation combinatoires.

Nous avons présenté dans ce chapitre, le principe de la réactivité sur la liste tabou, les techniques ; la méthode d'Elimination Inverse, la technique de Hachage et la technique de hachage combinée à l'arbre rouge-noir permettent d'accélérer la recherche dans l'espace des solutions stockés, afin rendre la taille de la liste tabou réactive et plus adaptés aux problèmes traités.

CHAPITRE V : Recuit simulé et Réactivité

1. Introduction

Le recuit simulé (simulated annealing) est une métaheuristique de recherche locale très populaire, elle est très efficace et utile pour de nombreux problèmes pratiques d'optimisation combinatoires. Le recuit simulé est souvent présenté comme la plus ancienne des métaheuristiques, c'est la première à mettre spécifiquement en œuvre une stratégie d'évitement des minima locaux.

Nous présentons dans ce chapitre les principes de cette métaheuristique, nous présentons aussi quelques techniques qui permettent régler son mécanisme de refroidissement, ainsi que son état d'équilibre d'une manière adaptative et réactive.

2. Principe de recuit simulé

L'origine de la méthode du recuit simulé provient de la métallurgie, où, pour atteindre les états de basse énergie d'un solide, on chauffe celui-ci jusqu'à des températures élevées, avant de le laisser refroidir lentement. Ce processus est appelé le recuit.

Le recuit simulé a été développé simultanément par Kirkpatrick et al. [30] et Cerny [31]. Le recuit simulé repose sur l'algorithme de Metropolis [32] décrit par l'Algorithme 7. Cette procédure permet de sortir des minima locaux avec une probabilité élevée si la température T est élevée et, quand l'algorithme atteint de très basses températures, de conserver les états les plus probables.

Algorithme 7 : Principe de Metropolis

Initialiser un point de départ x_0 et une température T
Pour $i = 1$ à n **faire**
  Tant que x_i n'est pas accepté faire
    Si $f(x_i) <= f(x_i\text{-}1)$: accepter x_i
    Si $f(x_i) > f(x_i\text{-}1)$: accepter x_i avec la probabilité $e^{-\frac{f(s')-f(s)}{T}}$
  Fin Tant que
Fin Pour

L'algorithme de Metropolis permet d'échantillonner la fonction objectif par le biais d'une distribution de Boltzmann de paramètre T. Le point crucial de l'algorithme est donc la loi de décroissance de la température.

Le recuit simulé est un algorithme stochastique qui permet, dans certaines conditions, la dégradation de la solution. L'objectif est d'échapper à des optima locaux.

Le recuit simulé est un algorithme sans mémoire dans le sens où l'algorithme n'utilise pas les informations recueillies lors de la recherche.

À partir d'une solution initiale, l'algorithme se déroule en plusieurs itérations. A chaque itération, un voisin aléatoire est généré. Les mouvements qui améliorent la fonction objectif sont toujours acceptées. Autrement, le voisin est choisi avec une probabilité donnée qui dépend de la température actuelle et la quantité de dégradation ΔE de la fonction objectif. ΔE représente la différence de la valeur objectif (énergie) entre la solution courante et la solution voisine générée, avec la progression de l'algorithme, la probabilité que de tels mouvements (les mouvements qui n'améliorent pas la fonction objectif) ne sont acceptés diminue (**Figure 12**). Cette probabilité suit, en général, la distribution de Boltzmann:

$$P(\Delta E, T) = e^{-\frac{f(s')-f(s)}{T}}$$

La fonction utilise un paramètre de contrôle, appelée température, afin de déterminer la probabilité d'accepter des solutions non améliorante. À un certain niveau de température, de nombreux essais sont explorées. Une fois l'état d'équilibre est atteint, la température est progressivement diminuée conformément à un calendrier de refroidissement de telle sorte que des solutions non améliorante sont acceptées à l'issue de la recherche.

L'algorithme de recuit simulé est résumé par l'Algorithme 8.

Algorithme 8 : Principe de recuit simulé

Déterminer une configuration aléatoire *S*
Choix des mécanismes de perturbation d'une configuration
Initialiser la température *T*
Tant que la condition d'arrêt n'est pas atteinte faire
  Tant que l'équilibre n'est pas atteint faire
    Tirer une nouvelle conguration *S'*
    Appliquer la règle de Metropolis
    Si *f(S') < f(S)*
    $S_{min} = S'$
    $f_{min} = f(S')$
    Fin Si
  Fin Tant que
  Décroître la température
Fin Tant que

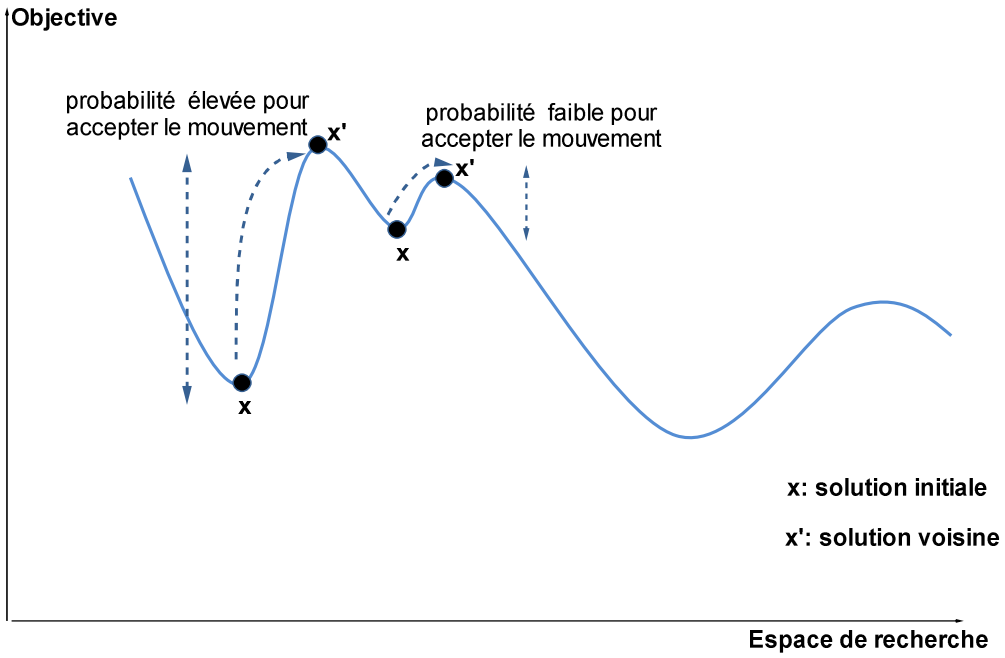

Figure 12 : Recuit simulé échappant à des optima locaux. Plus la température est élevée, plus est importante la probabilité pour accepter les mauvais mouvements.les meilleurs mouvements sont toujours acceptés.

L'algorithme de recuit simulé est devenu rapidement populaire, du fait de sa facilité d'adaptation à un grand nombre de problèmes et son efficacité. En revanche, cet algorithme présente l'inconvénient de disposer d'un nombre élevé de paramètres (température initiale, règle de décroissance de la température, durée des paliers de température, etc.) qui rendent les réglages de l'algorithme assez empiriques.

3. L'État d'équilibre et réactivité

Pour atteindre un état d'équilibre à chaque température, un certain nombre de transitions suffisantes (mouvement) doit être appliqué. La théorie suggère que le nombre d'itérations à chaque température peut être exponentiel à la taille du problème traité, ce qui est difficile à appliquer dans la pratique. Le nombre d'itérations doit être réglé en fonction de la taille de l'instance de problème et en particulier proportionnelle à la taille du voisinage $| N(s) |$.

En général, le nombre de transitions visité est choisi d'une manière statique : le nombre de transitions est déterminé avant le démarrage de la recherche. Par exemple, une proportion y d'un voisinage $N(s)$ est explorée. Par conséquent, le nombre de voisins générés à partir d'une solution s est $y \cdot | N(s) |$. Plus est important le rapport y, plus est le coût de calcul et meilleurs sont les résultats.

Dans la méthode adaptative, Le nombre de voisins générés dépendent des caractéristiques de la recherche. Par exemple, il n'est pas nécessaire pour atteindre l'état d'équilibre à chaque température.

Le recuit simulé non équilibré est proposé par Cardoso et Salcedo [33], cette méthode stipule que la phase de refroidissement peut être déclenchée dès qu'une solution voisine améliorante est générée. Cette fonction peut entraîner la réduction du temps de calcul sans compromettre la qualité des solutions obtenues.

Une autre approche adaptative, proposé par A.Torn et S.Viitanen [34], peut être utilisé, elle utilise à la fois les mauvaises et les meilleures solutions contenues dans la boucle interne de l'algorithme. Soit f_l (respectivement f_h) désigne la plus petite valeur de la fonction objectif dans la boucle courante interne, le nombre suivant de transitions L est défini comme suit:

$$L = L_B + \lfloor L_B \,.\, F_{-} \rfloor$$

Avec : $F_{-} = 1 - e^{\frac{-(fh-fl)}{fh}}$, et L_B est la valeur initiale du nombre de transitions.

4. Refroidissement et réactivité

Dans l'algorithme de Recuit simulé, la température diminue d'une manière progressive, tel que :

$$T_i > 0, \forall_i \text{ et } \lim_{i\to\infty} T_i = 0$$

Il y a toujours un compromis entre la qualité des solutions obtenues et la vitesse de refroidissement. Si la température diminue lentement, les solutions obtenues sont meilleures, mais avec un temps de calcul plus important.

Généralement le mécanisme de refroidissement est réglé d'une manière statique (linéaire, géométrique ou logarithmique), dans le sens où il est défini complètement a priori, Dans ce cas, le mécanisme de refroidissement est "aveugle" aux caractéristiques du paysage de la recherche.

Dans le mécanisme de refroidissement adaptatif, le taux de diminution de la température est dynamique et dépend des informations obtenus durant la recherche [35]. Le mécanisme de refroidissement dynamique peut être utilisé dans le cas où un petit nombre

d'itérations sont effectuées à des températures élevées, ou un grand nombre d'itérations à des températures basses.

5. Conclusion

Le recuit simulé est connus pour son efficacité de résoudre beaucoup de problèmes d'optimisation combinatoires. Néanmoins cet algorithme présente quelques inconvénients, tel que la difficulté de déterminer la température initiale, Nous avons présenté dans ce chapitre une technique qui permet de régler ce problème, en rendant son mécanisme de refroidissement, réactif. Le nombre de transissions générées à chaque itération peut être aussi déterminé d'une manière adaptative, sans qu'il nécessite d'atteindre l'état d'équilibre, ce qui améliore considérablement la vitesse de l'algorithme.

CHAPITRE VI : Mise en œuvre

1. Introduction

Dans ce chapitre nous mettons en œuvre les principes que nous avons vus dans les chapitres précédents. Dans un premier temps, nous commençons par implémenter une méthode *tabou classique* avec une liste fixe, puis nous introduisons ensuite un apprentissage intelligent qui aide l'algorithme à s'auto-adapter avec des paramètres appropriés afin de donner des résultats plus efficaces (réactivité sur la liste) ou améliorer le temps d'exécution (réactivité sur le voisinage).

Nous proposons de tester notre algorithme sur le fameux Problème du Voyageur de Commerce (PVC) qui consiste en : étant donné un ensemble de villes séparées par des distances données, de trouver un chemin de longueur totale minimale qui passe exactement une fois par chaque point et revienne au point de départ (**Figure 13**). Il s'agit d'un problème d'optimisation pour lequel on ne connait pas d'algorithme permettant de trouver une solution exacte en un temps polynomial. Si nous essayons de résoudre le problème avec une méthode exacte nous serions contraint d'analyser un espace de solutions de l'ordre de ***O(n!)***.

Etant donné que le problème PVC est bien étudié et que plusieurs travaux sont menés sur ce sujet [36], notre but n'est pas de se focaliser sur le PVC mais plutôt de proposer une méthode tabou plus perfectionnée et plus intelligente indépendamment du problème traité.

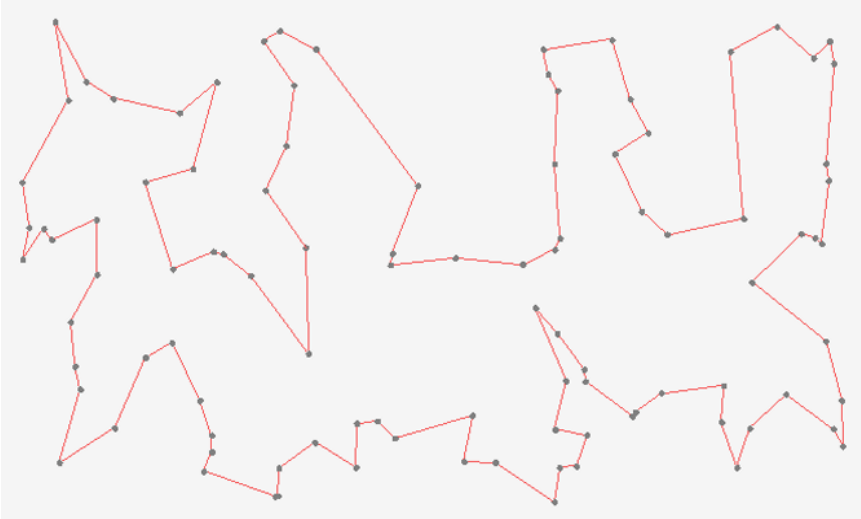

Figure 13 : Solution Optimale d'un problème de voyageur de commerce pour une instance de 100 villes.

2. Environnement de travail

Notre algorithme est codé en java sdk2, avec Netbeans comme un environnement de développement.

Les tests de notre méthode sont menés sur un processeur CoreI5.

Pour que les observations soient plus pratiques, nous testons notre algorithme sur des instances de PVC bien étudié (*Benchmark* [37]*)* dont nous connaissons l'optimum global.

Nous avons gardé les mêmes solutions de départ pour chaque Benchmark afin de bien remarquer le comportement des stratégies utilisées, le nombre de villes est intégré dans le nom du benchmark.

La qualité des solutions est représentée par l'écart entre la solution de départ et l'optimum obtenu.

3. Point de départ

Nous expliquons d'abord les spécifications de base de la *recherche tabou classique* que nous avons implémentée et que nous avons utilisée comme Framework pour la suite de notre travail.

3.1. Les spécifications de base

Représentation de la solution : une solution réalisable est représentée comme une séquence de nœuds, chaque nœud apparait une seule fois, le premier nœud visité ainsi que le dernier sont fixés à 1.

Solution initiale : deux algorithmes de recherche d'une solution sont utilisés :

- ***Solution initiale aléatoire* :** il s'agit de prendre le nœud « 1 » comme départ et sélectionner ensuite les autres nœuds d'une manière aléatoire sans aucune contrainte, elle fournit des solutions de qualité dégradée.
- ***Solution initiale par heuristique gloutonne* :** il s'agit d'utiliser une heuristique gloutonne dite du premier voisin d'abord, qui consiste à prendre le premier nœud et trouver son voisin non visité le plus proche, cette étape est répétée tant qui il y a des nœuds non visités.

L'effet du choix de la solution initiale est bien aperçu sur les instances de grande taille (dépassant 100 villes), ou l'algorithme met beaucoup de temps avant qu'il converge vers des solutions de bonne qualité, dans de tels cas, le choix de la solution initiale par heuristique est préférée, mais il s'avère parfois pénalisant [13].

Voisinage : un voisinage d'une solution courante est définie par l'ensemble de tous les échanges possibles entre deux nœuds, si nous avons un plan de « n » villes en fixant le premier et le dernier nœud à « 1 » ; alors nous aurons un voisinage de taille : [(n-1)*(n-2)/2].

Ce choix de voisinage est appelé voisinage complet, il conduit à un balayage complet des solutions voisines. Il garantit une bonne qualité de solutions, mais s'avère exorbitant en matière de temps d'exécution pour des instances de grande taille à qui on préfère généralement un voisinage partiel.

Dans la section *Réactivité sur le voisinage***,** nous allons montrer comment un changement dynamique et intelligent sur le voisinage peut réduire considérablement le temps d'exécution sans toutefois altérer la qualité des solutions.

Fonction objective : à partir d'une matrice de distances, le coût d'une solution est évalué par la somme de toutes les distances séparant les nœuds.

Puisque le coût d'une solution est relatif au coût de la solution voisine précédente, nous allons garder l'ancienne valeur pour éviter de recalculer le coût de nouveau et le modifier en fonctions des mouvements sélectionnés.

Mouvement : un mouvement est une transition de la meilleure solution *s''* du voisinage précédent à la meilleure solution *s'* du voisinage actuel. Le coût de chaque solution est calculé par la fonction objectif, selon laquelle on peut savoir si le mouvement est améliorant ou pas, le type de mouvement que nous utilisons dans notre algorithme est un changement entre deux villes à chaque itération.

Liste Tabou: pour éviter que l'algorithme cycle dans un ensemble de solutions, les mouvements de nœud récemment effectués sont stockés dans une liste tabou, qui interdise leur répétitions dans un laps de temps limité, après un certain temps (relatif à la taille de la liste) le mouvement sera à nouveau autorisé.

Le choix de la taille de la liste est très critique, il risque de mener à des cycles dans le cas des petites tailles ou à des diversifications non souhaitées pour des grandes tailles.

Dans la section **Réactivité sur la Liste Tabou,** nous allons montrer comment l'apport de la réactivité et de l'intelligence sur le la taille de la liste peut affecter d'une manière positive la qualité des solutions et tout cela sans l'intervention de l'utilisateur.

Critère d'aspiration : Dans certains cas, les interdictions occasionnées par la liste taboue sont jugées trop radicales. En effet, on risque d'interdire certains mouvements particulièrement utiles. Alors pour assouplir le mécanisme de la liste tabou nous allons faire recours à un critère d'aspiration selon lequel un mouvement bien qu'il est tabou, peut quand même être accepté.

Le critère que nous choisissons dans notre cas est d'autoriser un mouvement tabou s'il conduit à une valeur objective meilleure que la meilleure solution déjà trouvée.

Critère d'arrêt : l'algorithme s'arrête quand le nombre d'itération s'achève.

3.2. Résultat

Le tableau suivant présente des résultats numériques de notre algorithme de base ; la recherche tabou classique :

Tableau 4 : Résultats obtenus par la recherche tabou classique pour 100000 itérations.

Benchmark	Taille (liste tabou)	Voisinage	Temps d'exec.	Solution de départ	Optimum (Amélioration de la fonction objective %)
berlin52	20	Fixe Complet	18s	22205.617	8892.509(149%)
		Fixe 1/2	10s		12584.137(76%)
	100	Fixe Complet	22s		8957.772(147%)
		Fixe 1/2	13s		9878.258(124%)
st70	30	Fixe Complet	34s	3410.556	744.346(358%)
		Fixe 1/2	20s		995.080(242%)
	100	Fixe Complet	36s		751.883(353%)
		Fixe 1/2	24s		900.45(278%)
KroA100	50	Fixe Complet	74s	191393.738	27359.561(600%),
		Fixe 1/2	40s		40739.124(369%)
	100	Fixe Complet	74s		25173.627(660%)
		Fixe 1/2	45s		37620.665(408%)

Bien que les résultats obtenu par la *recherche tabou classique* sont acceptables, nous remarquons que les paramètres (taille de liste et voisinage) ont une influence majeur sur la méthode.

La taille de voisinage à un impact visible sur la qualité des solutions obtenues, ainsi que le temps d'exécution, s'il est choisi complet, la qualité serait meilleure au détriment du temps d'exécution, nous devrions alors trouver une solution qui nous permet d'accélérer le temps d'exécution sans perdre la qualité des solutions obtenues.

Nous essayons aussi de rendre la liste tabou adaptatif selon la trajectoire de recherche obtenue, la liste serait courte si besoin d'une intensification, large, si besoin d'une diversification.

4. Réactivité sur le voisinage

Le choix de la taille de voisinage a un impact majeur sur les performances de la méthode *tabou*, puisqu'il influence sur la qualité des solutions et le temps d'exécution. D'une manière générale un voisinage ***N(S)*** est défini comme un ensemble de solutions qui peuvent être obtenues par l'application d'une transition sur une solution courante.

Dans notre cas, le voisinage de *PVC* est l'ensemble des solutions qu'il est possible de construire en permutant deux villes dans une solution courante.

La taille de voisinage peut alterner tout en étant large (assurer la qualité au détriment du temps d'exécution) ou étant court (favoriser le facteur du temps au détriment de la qualité).

4.1. Les différents types de voisinage

Dans ce qui suit, nous discutons les différents types de voisinage et leur impact sur la performance de la stratégie de recherche :

Voisinage Complet : Dans ce type de voisinage, toutes les solutions voisines sont générées et évaluées à chaque itération, pour un problème de n variables (villes dans notre cas), une ville est échangée avec l'ensemble de toutes les villes existantes, la taille du voisinage est évaluée à : $|N(s)| = n(n-1)/2$.

Le voisinage complet garantie une qualité de solution meilleure, mais au fur et à mesure que la taille du problème augmente le voisinage augmente d'une manière

importante, par conséquent son évaluation deviendrait une tache trop gourmande en temps UC et qui peut mener à une convergence trop lente.

Voisinage Partiel : Dans ce cas, le voisinage est généré par l'échange d'une ville avec un ensemble « n' » d'autres villes. La taille de voisinage est évalué à :

$$|N(s)| = n'(n - n') + \sum_{i=1}^{n'}(\mathrm{i} - 1)$$

Le voisinage partiel peut réduire le temps d'exécution d'une manière considérable mais ne garantie, malheureusement pas, une bonne qualité de solution.

Le Premier Voisin Améliorant : Dans ce cas, un voisinage complet ou partial est généré, mais le processus de son évaluation s'arrête au premier voisin améliorant, si un tel voisin n'est pas trouvé alors tout le voisinage serait évalué et le meilleur serait sélectionné.

La taille du voisinage dans ce cas n'est pas constante et peut s'alterner entre un minimum "$(n-1)$" et un maximum "$n(n-1)/2$".

Voisinage Aléatoire : Dans ce type de voisinage, un ensemble de « n' » villes est sélectionné d'une manière aléatoire. La taille de voisinage est similaire à la taille de voisinage partiel.

Voisinage Adjacent : c'est un cas spécial de Voisinage Partiel où seulement la ville adjacente est sélectionnée. C'est le voisinage le plus court en matière de taille, et il est évalué à : $|N(s)| = (n-1)$.

4.2. Voisinage Réactif

Cette technique a pour objectif de combiner entre l'avantage du Voisinage Complet (qualité des solutions) et celui du Voisinage Partiel (temps d'exécution) (**Figure 14**).

L'idée de base est de commencer avec un Voisinage Adjacent (Minimum) et de lancer la recherche. Si des améliorations sont trouvées, des diminutions seront effectuées.

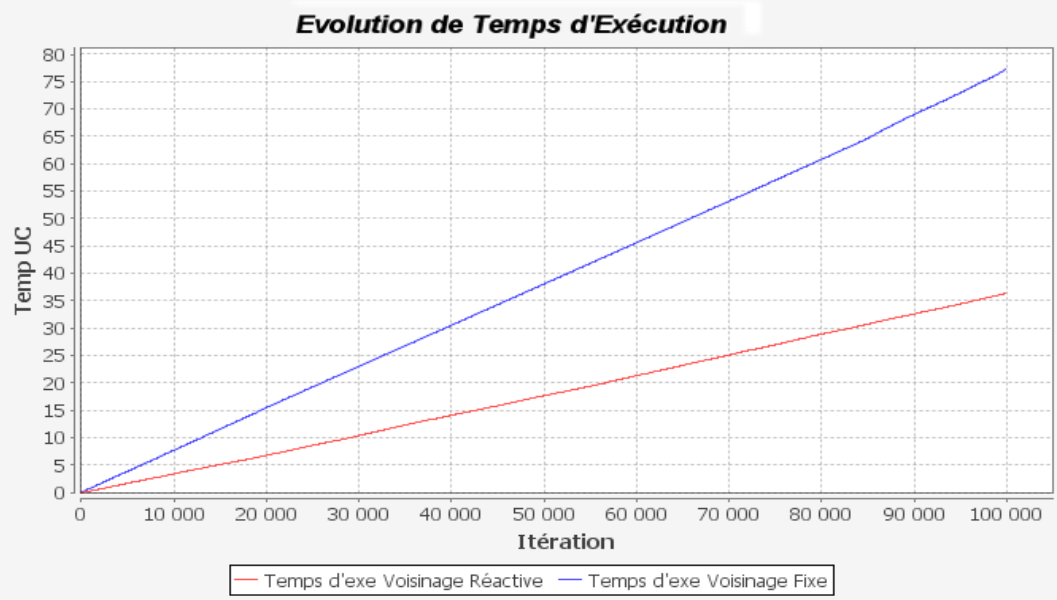

Figure 14: Comparaison entre le Temps d'exécution d'un Voisinage Réactif et un Voisinage fixe pour une instance de 100 villes (le voisinage réactif permet de gagner plus de la moitié du temps d'exécution, sans toutefois perdre de la qualité de la solution).

Dans le cas où il n'y aurait plus d'amélioration, alors nous opérons une augmentation graduelle du voisinage jusqu'à ce qu'il atteigne la taille maximale (celle du voisinage Complet). Ceci peut être expliqué par le fait que s'il n'y a pas d'amélioration, c'est parce que le voisinage est pauvre en matière de bonnes solutions, alors nous augmentons la taille afin de l'enrichir. Par contre, s'il y a des améliorations, c'est que le voisinage est riche, alors nous diminuons la taille de voisinage en espérant toujours que le sous-voisinage résultant serait aussi bon. La **Figure 15** illustre clairement ce principe.

Le pseudo-code du Voisinage Réactif peut se représenter comme suit :

Algorithme 9 : principe du Voisinage Réactif

Procédure *Voisinage_Reactif* (N_1,.., N_{Kmax})
  Poser K ← 1; (K taille du Voisinage)
  Répéter
    Choisir *s'* qui minimise *f(s')* dans $N_K(s)$;
    Si *f(s')* < $f(s^{-1})$ alors (s^{-1} Solution précédente)
      K ← Min (Augmenter(K), K_{max}) ;
    Sinon
      K ← Max (Diminuer(K), 1) ;
  Jusqu'à ce que le critère de terminaison soit satisfait
Fin

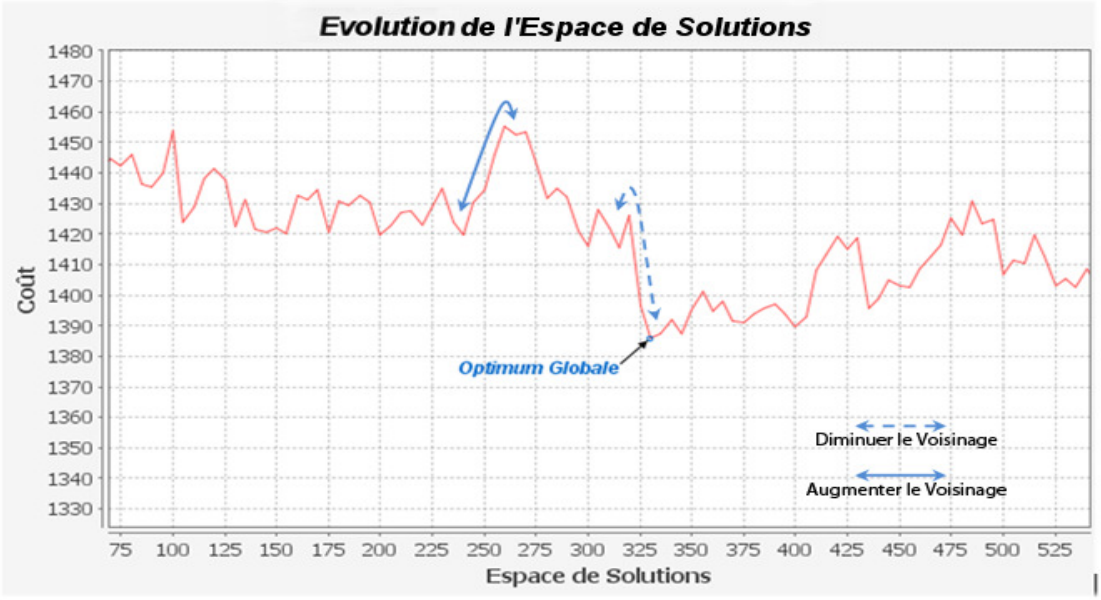

Figure 15 : Principe du voisinage réactif sur un espace de solutions pour une instance de 100 villes.

4.3. Pas d'augmentation et pas de diminution

Afin d'améliorer au maximum le temps d'exécution, le pas de diminution sera toujours plus important que le pas d'augmentation. (Le principe d'une diminution rapide et une augmentation prudente). Dans notre cas, le pas de diminution est fixé à « 1/6 » et le pas d'augmentation à « 1/12 » (**Figure 16**), valeurs prises à partir d'une série d'expérimentations.

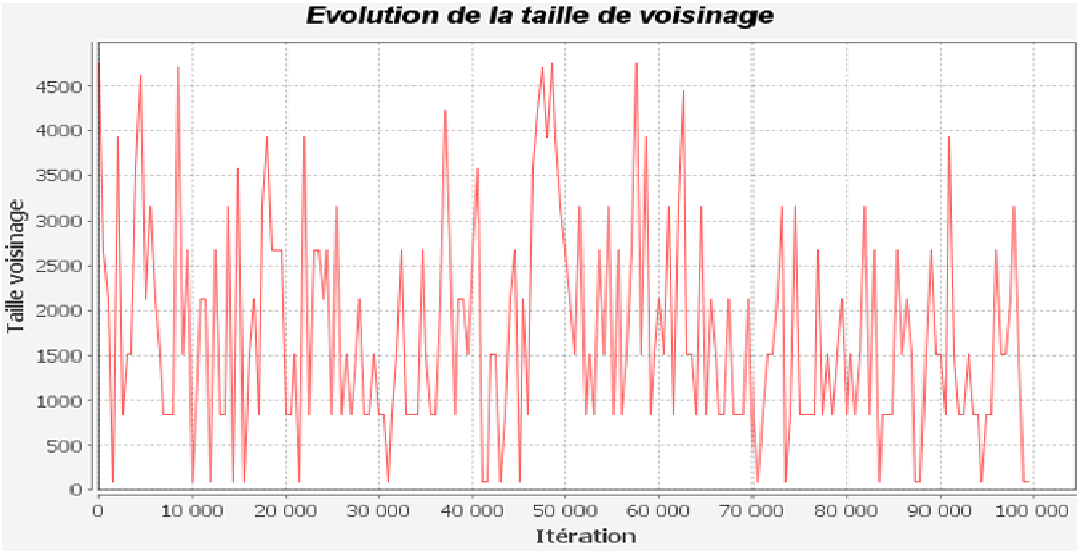

Figure 16 : Evolution de la taille de voisinage pour une instance de 100 villes. (Le voisinage s'alterne entre un Voisinage Complet « 4851= (99*98)/2 », et un Voisinage Adjacent « 99 ».

4.4. Résultat comparatif

Le tableau suivant présente des résultats comparatifs entre la méthode tabou avec voisinage fixe et la méthode tabou avec voisinage réactif.

Tableau 5 : Résultats comparatif entre voisinage réactif et voisinage fixe pour 100000 itérations.

Benchmark	**Taille (liste tabou)**	**Voisinage**	**Temps d'exec.**	**Solution de départ**	**Optimum (Amélioration de la fonction objective %)**
berlin52	20	Réactive	11 s	22205.617	8892.509(149%)
		Fixe Complet	18s		8957.772(147%)
	100	Réactive	12.5s		7911.124(180%)
		Fixe Complet	22s		7888.232(181%)
st70	30	Réactive	16s	3410.556	744.346(358%)
		Fixe Complet	34s		757.373(350%)
	100	Réactive	19s		751.883(353%)
		Fixe Complet	36s		719.638(373%)
KroA100	50	Réactive	35s	191393.738	27359.561(600%),
		Fixe Complet	74s		26218.719(627%),
	100	Réactive	38s		25173.627(660%)
		Fixe Complet	74s		24649.664(676%)

la réactivité sur le voisinage permet d'économiser un temps d'exécution considérable avec des qualités qui ne sont pas assez différente de celle résultante d'un voisinage fixe, mieux encore, nous constatons qu'à chaque fois que le nombre de villes augmente, le voisinage réactif offre des solutions mieux que celles d'un voisinage fixe (avec évidement l'avantage d'un gain de presque la moitié du temps d'exécution).

4.5. Parallélisme

Le Voisinage Réactif économise considérablement le temps d'exécution mais malheureusement ce mécanisme ne suffit pas au fur et à mesure que la taille du problème augmente. Nous introduisons pour cela un parallélisme [13] qui consiste à partitionner le voisinage à des sous-voisinages (selon le nombre de processeurs existants).

Le pseudo-code du Voisinage Parallèle peut se représenter comme suit :

Algorithme 10 : principe du Voisinage Parallèle

Procédure *Voisinage_ Parallèle* ($N_{Kmax}(s)$)
Poser *nbr_UC* ← *le nombre de processeur;*
Répéter
Pour i=1 Jusqu'à *nbr_UC* faire
K=K_max/nbr_UC (K taille de Voisinage)
Créer Un Thread ;
Choisir *s''* qui minimise *f(s'')* dans $N_K(s)$;
Fin Pour
S' ← *Min (Résultats obtenus par tous les threads créés) ;*
Jusqu'à ce que le critère de terminaison soit satisfait
Fin

4.6. Résultat

Le tableau suivant présente des résultats comparatifs entre la méthode Tabou avec voisinage réactif avec et sans parallélisme.

Tableau 6 : Résultats comparatif entre voisinage réactif et voisinage fixe pour 30000 itérations.

Benchmark	Liste Tabou	Voisinage	Temps d'exec.	Solution de départ	Optimum(Amélioration de La fonction objectif %)
fl417	100	Réactive	136s	55387.192	25507.846 (117.13%)
		Réactive Parallèle	86s		25507.846(117.13%)
	120	Réactive	138s		25489.408(117.29%)
		Réactive Parallèle	87s		25489.408(117.29%)
pr439	100	Réactive	180s	270651.604	188588.863(43.51%)
		Réactive Parallèle	115s		188588.863(43.51%)
	120	Réactive	183s		171720.748(57.61%)
		Réactive Parallèle	116s		171720.748(57.61%)

Le tableau montre clairement l'avantage du voisinage parallèle, le parallélisme permet d'économiser un temps d'exécution considérable.

5. Réactivité sur la liste

Pour s'extraire d'un optimum local et afin d'éviter d'y retomber périodiquement, une liste tabou qui contient les mouvements les plus récents est introduite, hélas ce mécanisme ne garantie pas l'absence total de cycles.

Le choix de la taille de la liste influe largement sur le comportement de l'algorithme, en effet, si la liste est trop courte, l'intensification est trop forte, l'algorithme reste autour d'un

optimum local et ne peut pas améliorer la solution courante. Ou dans le cas contraire, la diversification de la recherche est trop importante et l'algorithme ne découvre pas de bonnes solutions.

Généralement, nous somme obligé de mener plusieurs expérimentations avant de trouver la taille convenable à notre problème.

Afin d'automatiser le processus de sélection de la taille de la liste et combiner entre des intensifications et des diversifications, nous allons introduire un apprentissage sur l'espace des solutions déjà visitées.

5.1. Principe

Le principe d'une mémoire à long terme est introduit afin de stocker toutes les configurations déjà visitées. A chaque itération, l'espace de stockage est sollicité pour vérifier si la configuration courante y existe, si c'est le cas, une augmentation de la taille de la liste est procédée pour éviter que d'autres répétitions s'en suivent.

Après chaque augmentation, un timer qui est relatif à la longueur du cycle détecté est déclenché, si aucune répétition n'est détectée durant cette période, alors nous pouvons procéder à des diminutions (**Figure 17**).

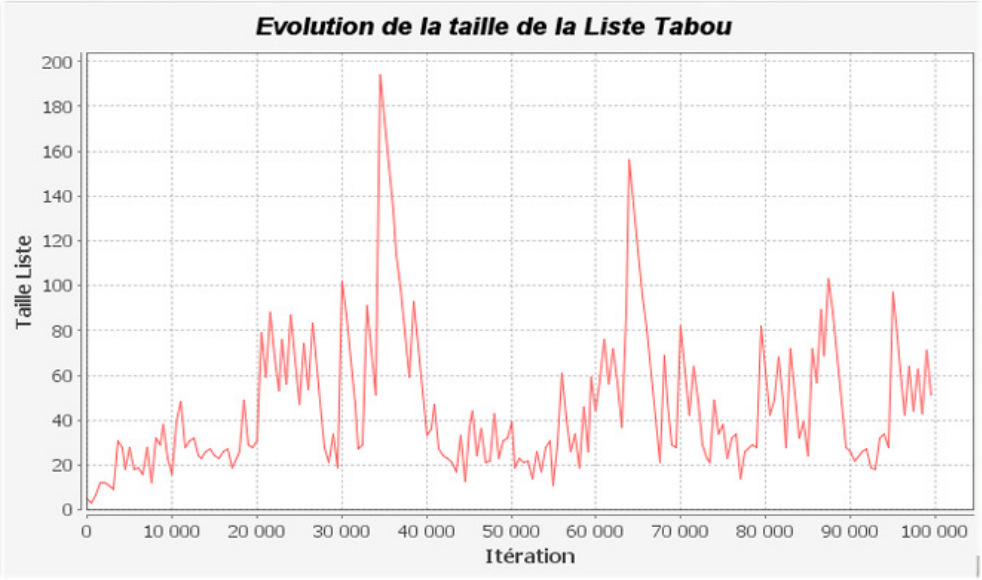

Figure 17 : Principe d'une Liste Réactive pour une Instance de 100 villes.

Contrairement au Voisinage Réactif, le pas d'augmentation est toujours plus important que celui de diminution. En effet, l'augmentation rapide peut briser n'importe quel cycle quelque soit sa taille. Quant à la diminution, elle est procédée, mais avec prudence par des petits pas pour éviter de retomber sur des cycles précédents.

Dans notre cas, la règle d'augmentation est prise à « * 2 », tandis que la diminution à « - 1 ». L'utilisateur pourra néanmoins changer ces règles.

Le pseudo-code de la Liste Réactive peut se représenter comme suit :

Algorithme 11 : principe de la liste réactive

```
Procédure Liste_Reactive (TL)
    Poser TL ← 1 ;              (TL taille de la liste tabou)
    Répéter
        Choisir s' qui minimise f(s') dans N (s) ;
        Chercher s' dans l'espace des solutions visités ;
        Si une boucle est détectée Alors
            TL ← Min (Augmenter (TL,Longuer_cycle), TLmax) ;
        Sinon
            TL ← Max (Diminuer(TL), 1) ;
    Jusqu'à ce que le critère de terminaison soit satisfait
Fin
```

5.2. Table de Hachage relative à l'espace de recherche

L'exploration de l'espace de recherche s'avère une tache complexe, il serait insignifiant d'explorer tout l'espace de recherche d'une manière simple (séquentiel), en effet, au fur et à mesure que l'espace de recherche augmente le temps que met l'algorithme pour l'explorer augmente aussi et d'une manière exponentielle.

Alors, nous faisons recours à une technique de Hachage (expliquée dans le chapitre précédant) qui consiste à appliquer une fonction sur la valeur de la fonction objectif (*Coût*) afin d'obtenir une entrée utilisé comme index pour un tableau.

Le problème est tel : quelle que soit la performance de la fonction de hachage, elle peut produire la même entrée pour deux valeurs différentes. On parle alors de collision.

Pour palier à ce problème, nous introduisons pour chaque entrée de tableau une liste qui contient les valeurs accidentées. Ainsi l'accès serait directe pour l'entrée du tableau et séquentiel pour la liste (**Figure 18**).

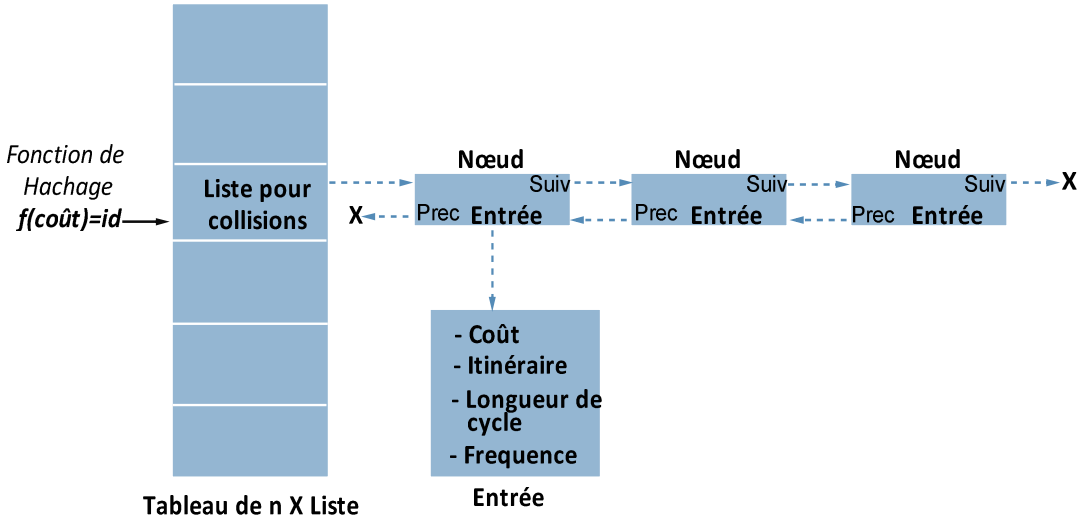

Figure 18: Structure de Hachage utilisée pour stocker l'espace de recherche.

La technique de Hachage permet d'avoir des accès instantanés qui économisent le temps d'exécution d'une manière considérable par rapport à la recherche classique (séquentielle) et dont le temps d'exécution ne diffère guère de la méthode tabou avec une liste Fixe (**Figure 19**).

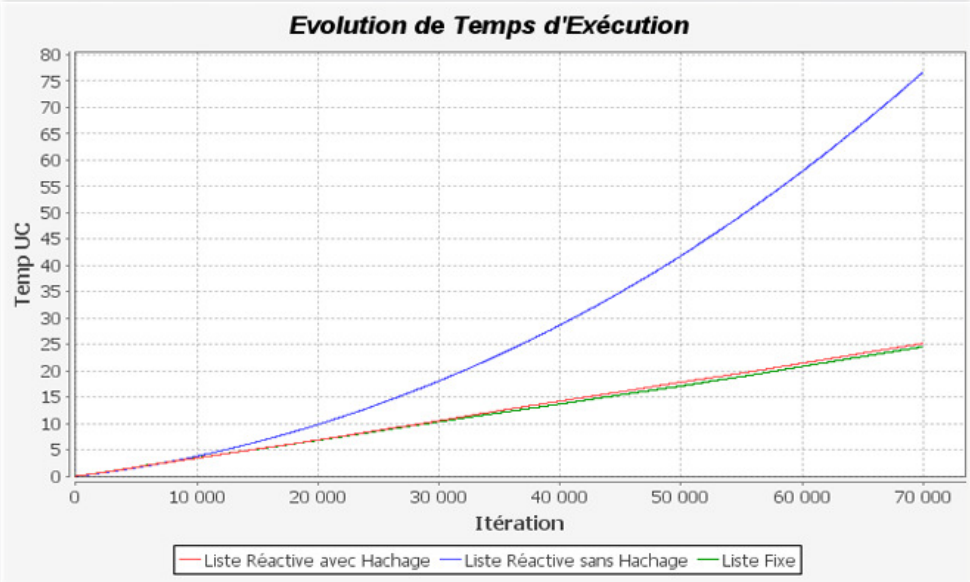

Figure 19: Comparaison entre le temps d'exécution des stratégies : liste réactive avec hachage, liste réactive sans hachage et une liste fixe. (Pour une instance de 100 villes)

5.3. Répétitions excessifs et stagnation

La détection des cycles n'est pas l'ultime objectif, en effet, il arrive parfois de retomber plusieurs fois sur certaines valeurs déjà visitées, ceci peut être expliqué par le fait que la trajectoire de recherche est bloquée dans une zone de recherche limitée, ce problème pourrait

être résolue par une détection des répétitions, si le nombre de répétitions dépasse une certaine valeur jugée critique (4 dans notre cas), un certain nombre de mouvements aléatoires est appliqué sur la meilleure solution déjà trouvée, et de relancer de nouveau la recherche.

Ce principe est appliqué aussi, si une stagnation d'une longe durée est détectée.

Le pseudo-code de la liste réactive avec des mouvements aléatoires peut se représenter comme suit :

Algorithme 12 : principe de la liste réactive aléatoire

```
Procédure Liste_Reactive_Aleatoire(TL)
    Poser TL ← 1 ;              (TL taille de la liste tabou)
    Répéter
        Choisir s' qui minimise f(s') dans N (s) ;
        Chercher s' dans l'espace des solutions visités ;
        Si Une Boucle est détectée Alors
           Incrémenter le Nbr_ocurrence ;
           TL ← Min (Augmenter (TL,Longuer_cycle), TLmax) ;
           Si Nbr_ocurrence > valeur_critique OU Une Stagnation est détectée Alors
              s'← Mouvement_Aleatoire(s*) ;
          Sinon
           TL ← Max (Diminuer(TL), 1) ;
     Jusqu'à ce que le critère de terminaison soit satisfait
Fin
```

5.4. Résultat

Le tableau suivant présente des résultats comparatifs entre la méthode Tabou avec liste réactive et une méthode tabou avec liste fixe.

Tableau 7 : Résultats comparatifs entre tabou réactive et tabou fixe pour 100000 itérations.

Benchmark	Liste Tabou	Voisinage	Temps d'exec.	Solution de départ	Optimum (Amélioration de la fonction objective %)
berlin52	Réactive	Réactive	12.5 s	22205.617	7899.052(181%)
		Fixe Complet	21s		7544.365(194%)
	Fixe(20,100)	Réactive	11s, 12.5s		8892.509(149%), 7911.124(180%)
		Fixe Complet	18s, 22s		8957.772(147%), 7888.232(181%)
st70	Réactive	Réactive	19s	3410.556	718.024(375%)
		Fixe Complet	36s		718.972(374%)
	Fixe(30,100)	Réactive	16s, 19s		744.346(358%),751.883(353%)
		Fixe Complet	34s, 36s		757.373(350%), 719.638(373%)

KroA100	Réactive	Réactive	34s	191393.738	23525.159(713%)
		Fixe Complet	77s		23610.518(710%)
	Fixe(50,100)	Réactive	35s, 38s		27359.561(600%),25173.627(660%)
		Fixe Complet	74s, 74s		26218.719(627%),24649.664(676%)
prA107	Réactive	Réactive	63s	62756.957	49721.450(26%)
	Réactive Aléatoire	Réactive	63s		46371.285(35%)
Ch130	Réactive	Réactive	80s	7575.286 (Heuristique)	6822.676(11%)
		Fixe Complet	160s		7041.323 (7.5%)

Nous activons le parallélisme pour les instances qui dépasse les 200 villes (deux Core du processeur CoreI5 sont utilisés) :

Tableau 8 : Résultats comparatifs entre tabou réactive et tabou fixe pour 20000 itérations.

Benchmark	**Liste Tabou**	**Voisinage**	**Temps d'exec.**	**Solution de départ**	**Optimum(Amélioration de La fonction objectif %)**
ts255	Réactive	Réactive	30 s	10299.896	5154.844 (99.8%)
		Fixe Partial (1/2)	55s		5237.650(96.6%)
	Fixe(20,100)	Réactive	30s, 29s		5525.347(86.4%),5204.363 (97%)
		Fixe Partial (1/2)	50s, 50s		5538.850(85.8%),5222.775(97.2%)
pr439	Réactive	Réactive	110s	270651.604	176290.340 (53.5%)
		Fixe Partial (1/2)	262s		172679.964 (56.7%)
	Fixe(30,100)	Réactive	80s, 111s		212422.141(27.4%),189105.225(43%)
		Fixe Partial (1/2)	260s, 259s		213010.618 (27%),197304.020 (37%)

La différence entre le temps d'exécution d'un voisinage réactif et un voisinage fixe est bien visible.

Les résultats montrent clairement que la réactivité sur la liste apporte le plus souvent un plus de qualité par rapport à une liste fixe (en plus de l'auto-adaptation).

Nous constatons aussi que l'aléatoire pourrait être très utile en apportant une certaine amélioration sur la qualité des solutions.

6. Présentation de l'interface de notre application

La **figure 20** présente un aperçu de l'interface principale de notre application.

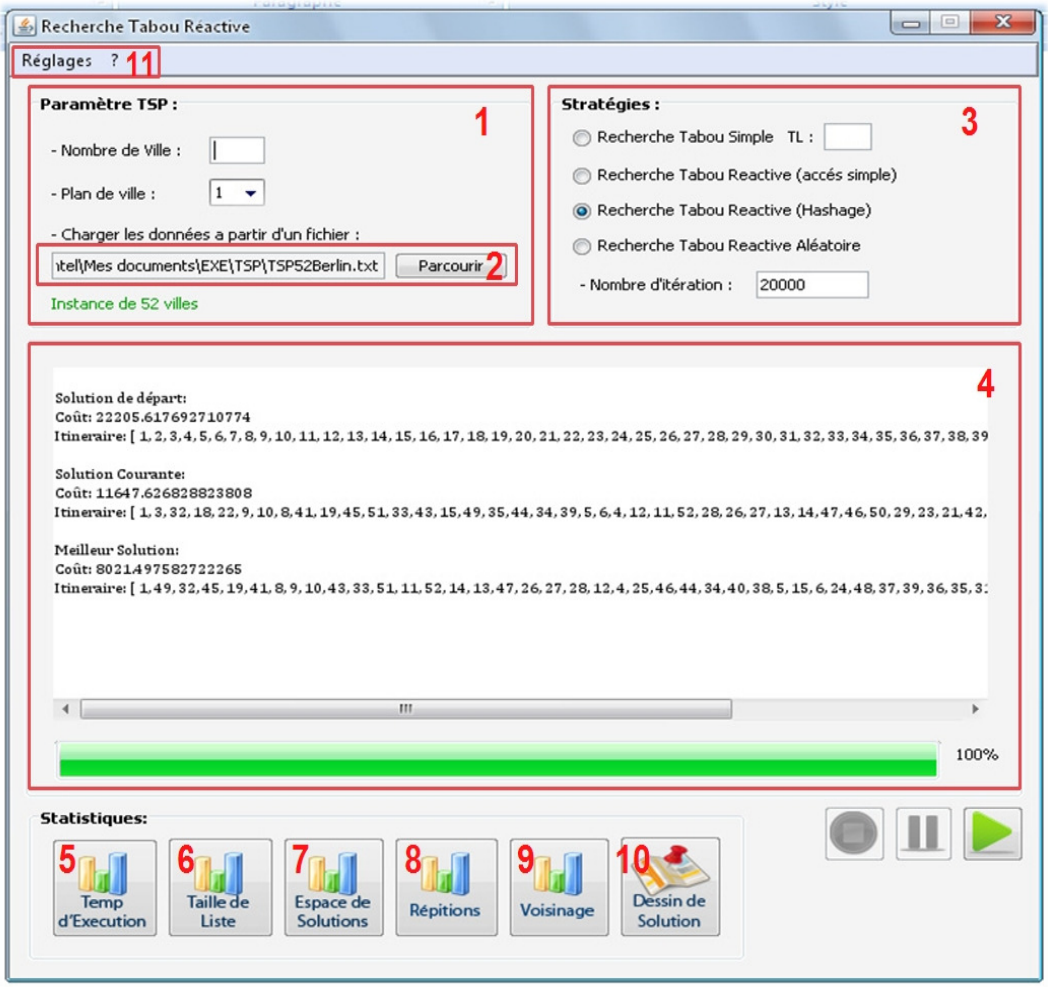

Figure 20 : l'interface principale de notre application.

1. Pour introduire les paramètres du problème de voyageur de commerce.
2. Importer les paramètres à partir d'un fichier (nombre de ville, cordonnées).
3. Choix de la stratégie voulue (tabou simple, réactive avec hachage,…).
4. Affichage du cout des solutions visitées, leur itinéraire, ainsi l'état de la liste tabou (pour savoir en détail le contenu de la liste tabou à chaque itération, ou pour une maintenance éventuelle).

5. Affichage de l'évolution du temps d'exécution :

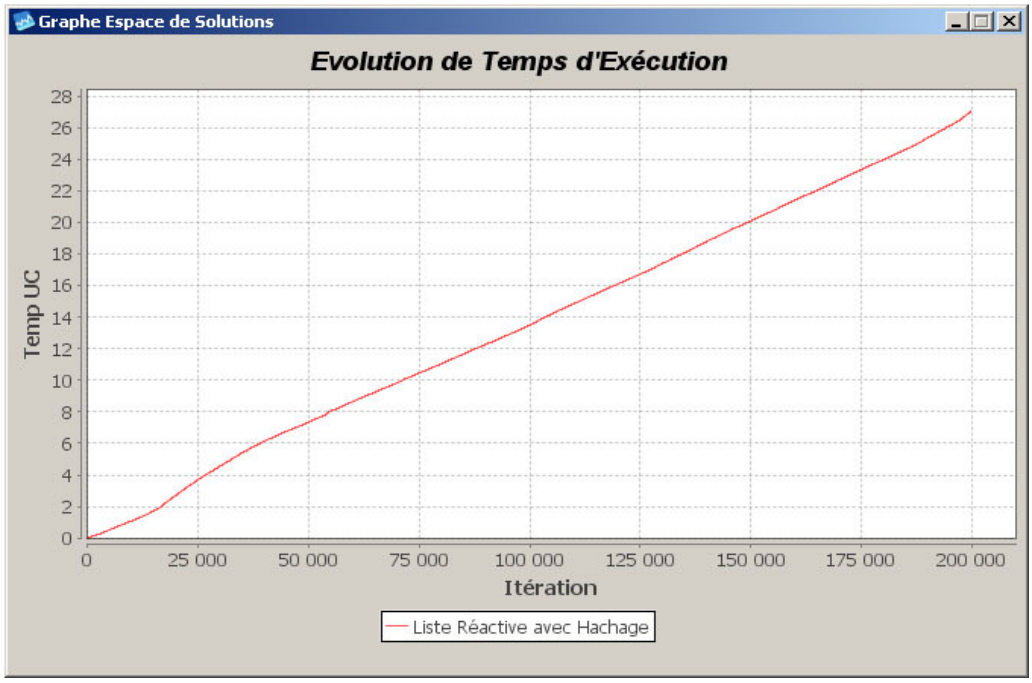

Figure 21 : interface de l'évolution du temps d'exécution.

6. Affichage de l'évolution la taille de la liste tabou :

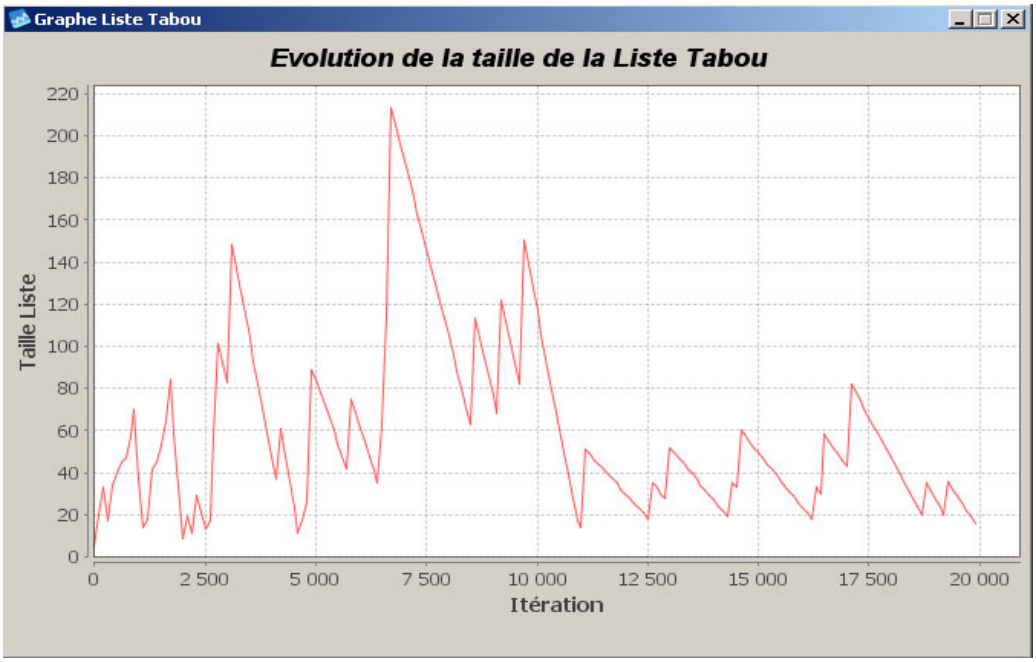

Figure 22 : interface de l'évolution la taille de la liste tabou.

7. Affichage de l'évolution de l'espace des solutions :

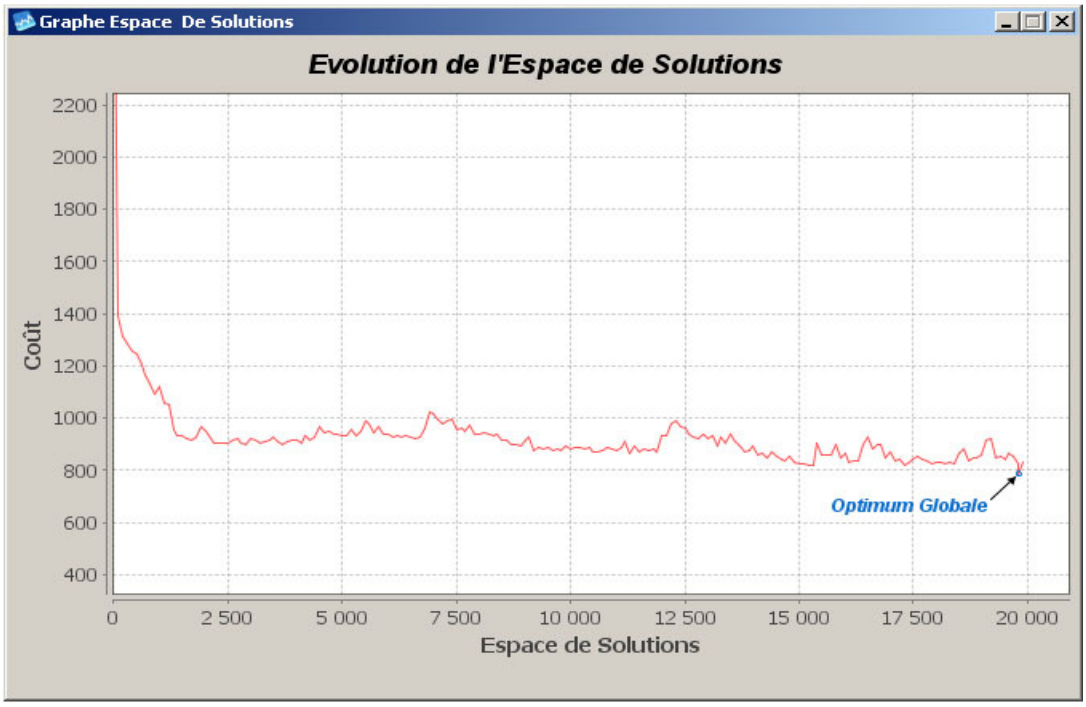

Figure 23 : interface de l'évolution de l'espace des solutions.

8. Affichage de l'évolution des répétions des solutions :

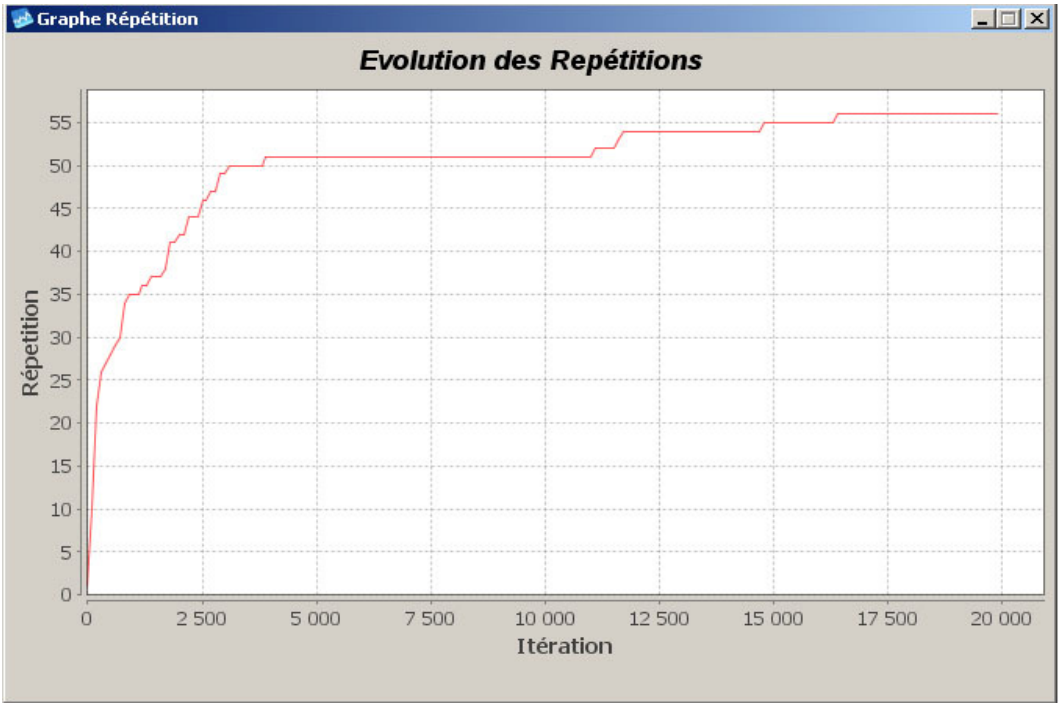

Figure 24 : interface de l'évolution des répétions des solutions.

9. Affichage de l'évolution de la taille de voisinage :

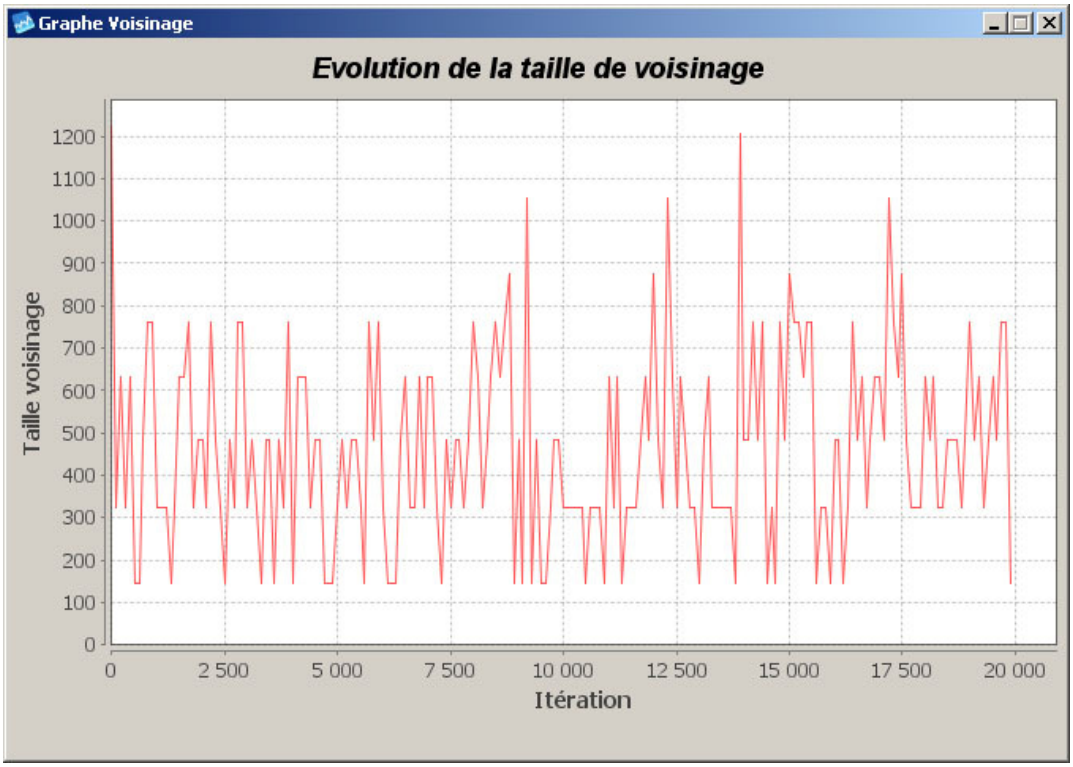

Figure 25 : interface de l'évolution de la taille de voisinage.

10. Affichage de l'itinéraire de la solution :

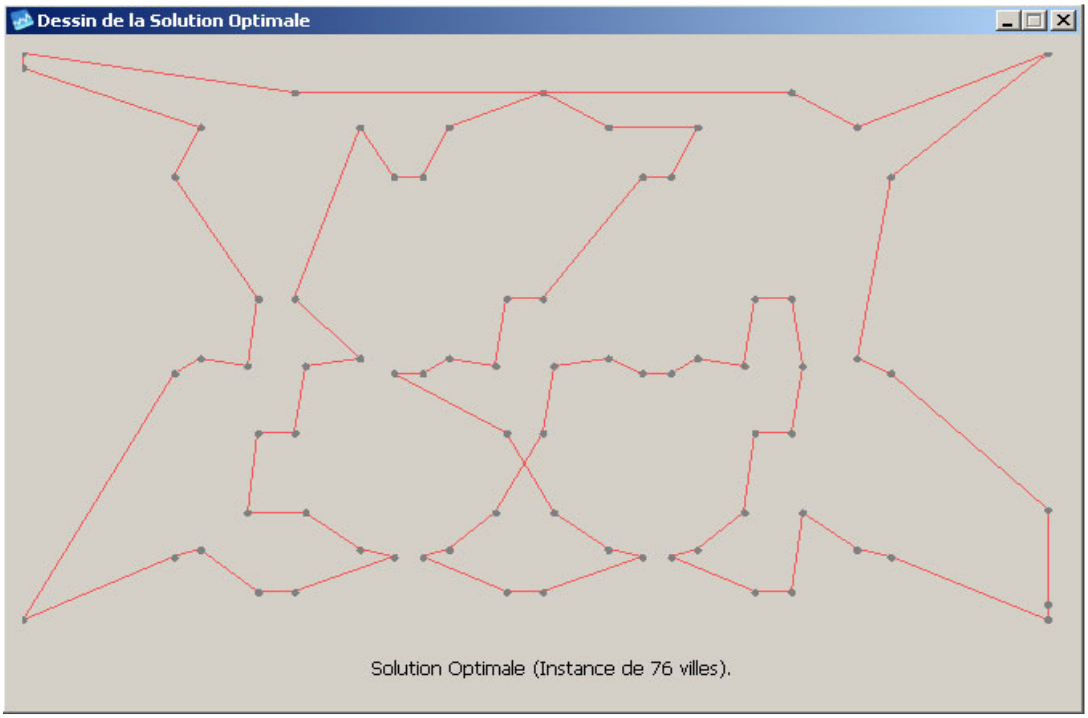

Figure 26 : interface de l'itinéraire de la solution.

11. Réglage avancé des paramètres (solution de départ, parallélisme …) :

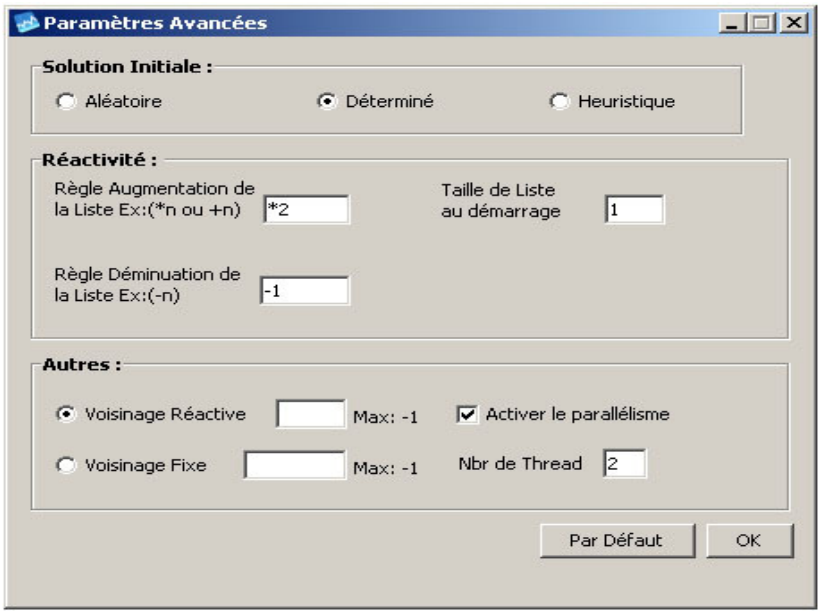

Figure 27 : interface pour le réglage avancé des paramètres.

7. Conclusion

Nous avons apporté un apprentissage intelligent afin d'automatiser le processus de paramétrage de l'algorithme de recherche tabou. La technique de hachage nous a permis d'explorer rapidement l'espace de recherche afin de comparer une solution candidate avec les solutions antérieurement visitées et de détecter d'éventuels cycles, ce mécanisme aide à l'adaptation dynamique de la liste tabou durant la recherche.

La qualité des solutions obtenues durant la recherche permet de prendre des décisions sur la taille de voisinage afin de le rendre dynamique et adaptatif.

Les expérimentations que nous avons présentées montrent clairement l'avantage de la Recherche Tabou avec liste et voisinage réactifs, non seulement sur le point de l'auto-configuration des paramètres mais aussi sur le temps d'exécution et la qualité des solutions obtenues.

CONCLUSION GENERALE

Au cours des trois dernières décennies, les algorithmes heuristiques ont permis de réaliser des progrès remarquables dans la résolution des problèmes difficiles d'optimisation combinatoire. Cependant, la conception de ces algorithmes relève encore plusieurs challenges importants en particulier ; le choix et la sélection de leurs paramètres.

Nous avons proposé plusieurs approches algorithmiques pour introduire une forme d'apprentissage dans les stratégies de recherche, dont le but général est de rendre les paramètres des métaheuristiques classiques, mieux adaptés aux problèmes traités. Un mécanisme adaptatif permet de modifier le comportement d'une méthode en fonction d'un historique de recherche.

Dans ce mémoire, nous avons travaillé sur la mise au point d'une métaheuristique de recherche locale réactive pour les problèmes d'optimisations combinatoire.

Premièrement, nous avons introduis quelques notions de base sur les problèmes d'optimisation combinatoire et les approches heuristiques. Nous avons porté une attention particulière aux méthodes de paramétrage des métaheuristiques, à savoir les méthodes hors ligne et en ligne, cette dernière approche peut, potentiellement améliorer le comportement des métaheuristiques en introduisant un apprentissage automatique et un mécanisme adaptatif à l'algorithme.

En suite, nous avons présenté des techniques et des méthodes qui permettent de rendre réactif et adaptatif certain paramètres qui sont communs pour la conception de la plupart des métaheuristiques. Pour la fonction objectif, nous avons présenté la technique de la Recherche Locale Guidée qui permet de varier la fonction objectif durant le processus de recherche, le but étant de rendre les minima locaux déjà visités moins attractifs. Pour le voisinage, nous avons présenté les méthodes de la Recherche à Voisinages Variables, qui utilisent méthodiquement plusieurs types de voisinages en fonction de l'espace de recherche visité, ce mécanisme permet de diversifier l'exploration de l'espace des solutions afin d'accéder à un plus grand nombre de régions intéressantes.

Après, Nous avons abordé le principe de la réactivité sur des paramètres qui sont propre à certain métaheuristiques connues. Pour la recherche tabou, nous avons abordé les techniques ; la méthode d'Elimination Inverse, la technique de Hachage et la technique de hachage combiné à l'arbre rouge-noir. Ces techniques permettent d'accélérer la recherche dans l'espace des solutions stockés afin rendre la taille de liste tabou réactive et plus adaptés

aux problèmes traités. Pour le recuit simulé, Nous avons présenté le principe de la réactivité sur son mécanisme de refroidissement, ainsi que pour son état d'équilibre.

Nous avons conclu notre travail par une mise en œuvre, qui consiste à introduire quelques concepts vus précédemment, à la recherche tabou classique. Nous avons doté la recherche tabou d'un apprentissage intelligent qui exploite les résolutions antérieures du problème traité, afin de trouver d'une part, la taille appropriée de la liste tabou, et d'autre part la taille adéquate du voisinage des solutions. Nous utilisons aussi une technique de hachage qui permet d'accélérer la recherche dans l'espace des solutions visitées afin de détecter l'apparition des boucles et leur longueur pour décider de garder, augmenter ou diminuer la taille de la liste tabou à chaque itération. Nous proposons aussi un voisinage dynamique de taille variable selon la qualité des solutions obtenues.

Notre méthode est appliquée au fameux problème du voyageur de commerce (PVC) et nous avons démontré par des expérimentations que les résultats obtenus par la recherche tabou avec voisinage réactif permet d'économiser un temps d'exécution considérable avec des qualités qui ne sont pas assez différente de celle résultante d'un voisinage fixe. Les résultats montrent clairement aussi, que la réactivité sur la liste apporte le plus souvent, un plus de qualité par rapport à une liste fixe, la sélection de paramètres les plus importants de l'algorithme (voisinage et liste tabou) devient automatique.

En guise de perspectives, Nous envisageons d'utiliser plusieurs structure de voisinage, nous nous somme intéressés plus particulièrement à l'adaptation de la taille de voisinage pour une seule structure de voisinage durant le processus de recherche. Certes les résultats expérimentaux ont montré clairement l'avantage de voisinage à taille variable (rapidité, qualité), cependant l'utilisation d'une seule structure de voisinage ne permet pas de diversifier radicalement l'exploration de l'espace des solutions. Dans notre cas (PVC), nous pensons intégrer un mécanisme qui sert à alterner entre les structures de voisinage suivant ; échange de deux villes, k-opt.

La seconde perspective envisagée est l'adaptation de la fonction objectif, nous nous somme intéressés à l'utilisation d'une seule fonction objectif durant tout le processus de recherche. Nous pensons alors de varier la fonction objectif durant le processus de recherche, la *Recherche Locale Guidé* nous semble une bonne piste à suivre.

BIBLIOGRAPHIE

[1] Y. Collette and P. Siarry. *Optimisation multiobjectif.* Eyrolles, 2002.

[2] C. H. Papadimitriou and K. Steiglitz, *Combinatorial optimization, algorithms and complexity*, Prentice-Hall, NJ, 1982.

[3] M.R. Garey, D.S. Johnson, et al. Computers and Intractability : A Guide to the Theory of NP-completeness. Freeman, San Francisco, U.S.A., 1979.

[4] D. Du and P.M. Pardalos. *Handbook of Combinatorial Optimization*. Springer, 2007.

[5] Porumbel D C, *Algorithmes Heuristiques et Techniques d'Apprentissage Applications au Problème de Coloration de Graphe*, Thèse de Doctorat, Université d'ANGERS, 2009.

[6] Devarenne I, *Etude en recherche locale adaptative pour l'optimisation combinatoire*, Université de Franche-Comte, Thèse de doctorat, 2007.

[7] H. Hoos and T. St utzle. *Stochastic Local Search Foundations & Applications*. Morgan Kaufmann Publishers Inc. San Francisco, CA, USA, 2004.

[8] G. Box, J. S. Hunter, andW. G. Hunter. *Statistics for Experimenters: Design*, Innovation, and Discovery. Wiley, 2005.

[9] J. D. Schaffer, R. A. Caruana, L. Eshelman, and R. Das. *A study of control parameters affecting online performance of genetic algorithms for function optimization*, 3rd International Conference on Genetic Algorithms, Morgan Kaufman,San Mateo, CA, 1989.

[10] M. D. Mckay, R. J. Beckman, and W. J. Conover. A comparison of three methods for the selecting values of input variables in the analysis of output from a computer code. Technometrics, 1979.

[11] D. Montgomery. Design and Analysis of Experiments. Wiley, 1984.

[12] M.Birattari, T. Stutzle, L. Paquete, and K.Varrentrapp. *A racing algorithm for configuring metaheuristics*. In W. B. Langdon et al., editors, Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'2002), Morgan Kaufmann, San Francisco, CA, 2002.

[13] El-Ghazali Talbi, *Metaheuristic From Design To Implementation*, John Wiley & Sons,2009.

[14] Roberto Battiti, Mauro Brunato and Franco Mascia, *Reactive search and intelligent optimization*, Technical Report DIT-07-049, July 2007.

[15] J. Maturana, *Contrôle générique de Paramètres pour les Algorithmes Evolutionnaires*. PhD thesis, University of Angers, France, 2009.

[16] C. Voudouris and E. Tsang. *Guided local search*. European Journal of Operational Research, 1999.

[17] P.Mills, E. Tsang, and J. Ford. *Applying an extended guided local search to the quadratic assignment problem*. Annals of Operations Research, 118:121–135, 2003.

[18] B. Freisleben and P. Merz. *A genetic local search algorithm for solving symmetric and asymmetric traveling salesman problems*. In Proceedings of IEEE International Conference on Evolutionary Computation, 1996.

[19] Farhad Kolhan, Ahmed Tavakoli, *The Effects of Different Neighborhood Generation Mechanisms On The Performance Of Tabu Search*, WSEAS TRANSACTIONS ON MATHEMATICS ISSUE 4 Volume 6, April 2007.

[20] N. Mladenovic, P. Hansen, *Variable Neighborhood Search*, European Journal of Operational Research, 2001.

[21]N. Mladenovic, P. Hansen, *Recherche à voisinage variable*, Technical report, Les Cahiers de GERAD, 2001.

[22] O. Braysy. *A reactive VNS for the vehicle routing problem with time windows*. INFORMS Journal on Computing. 2003.

[23] F. Glover. *Tabu search: Part I*. ORSA Journal on Computing, 1989.

[24] R. Battiti and G. Tecchiolli, *The Reactive Tabu Search*, ORSA Journal on Computing, 1994.

[25] F. Dammeyer and S. Voss, *Dynamic tabu list management using the reverse elimination method*, Annals of Operations Research 43,1993.

[26] D. L. Woodruff and E. Zemel, *Hashing vectors for tabu search*, Annals of Operations Research 41 ,1993.

[27] Jean-Philippe Collette. *Les tables de hachage*. Mise à jour le 7 juillet 2011. [En ligne]. http://jipe.developpez.com/articles/algo/table-hachage/?page=page_2 [Consultée le 20/07/2011]

[28] R. Bayer, *Symmetric binary b-trees: Data structure and maintenance algorithms*, Acta Informatica, 1972.

[29] N. Sarnak and R. E. Tarjan, *Planar point location using persistent search trees*, Communications of the ACM 29, 1986.

[30] S. Kirkpatrick, S. Gelatt, M. Vecchi. *Optimization by simulated annealing*, Science, Vol. 220,1983.

[31] V. Cerny. (1985). *Thermodynamical approach to the travelling salesman problem*, Journal of Optimization Theory and Applications, Vol. 45, 1985.

[32] N. Metropolis, A.W. Rosenbluth, M.N. Rosenbluth, A.H. Teller, E. Teller. *Equation of state calculations by fast computing machines*, Journal of Chemical Physics, Vol. 21, 1953.

[33] M. F. Cardoso, R. L. Salcedo, and S. F. Azevedo. *Non-equilibrium simulated annealing: A faster approach to combinatorial minimization*. Industrial Engineering ChemicalResearch, 1994.

[34] M. Ali, A. Torn, and S. Viitanen. *A direct search variant of the simulated annealing algorithm for optimization involving continuous variables*. Computers and Operations Research, N°29, 2002.

[35] L. Ingber. *Adaptive simulated annealing*. Control and Cybernetics, N°25(1),1996.

[36] G. Goos and J. Hartmanis, *The Traveling Salesman: Computational Solutions for TSP Applications*, Springer-Verlag, 1994.

[37] TSPLIB, *Best known solutions for symmetric TSPs*. Mise à jour le 10 Février 1995. [En ligne]. http://elib.zib.de/pub/mp-testdata/tsp/tsplib/stsp-sol.html [Consultée le 29/09/2011]

Résumé

La plupart des métaheuristiques, tels que la recherche tabou et le recuit simulé, bien qu'elles sont très efficace et utile pour de nombreux problèmes pratiques, elles sont très sensibles au choix de leurs paramètres. Par exemple, la Recherche Tabou dépend principalement de la taille de la liste tabou dont la valeur optimale peut varier en fonction de l'instance du problème traité.

Généralement, les concepteurs des métaheuristiques font recours à des séries d'expérimentations afin de sélectionner les bons paramètres (paramétrage hors-ligne), ce qui est coûteux en temps d'exécution et nécessite une connaissance approfondie de la stratégie de recherche utilisée ce qui n'est pas le cas pour un utilisateur final.

La Recherche d'Optimisation Réactive apporte une solution à ce type de problème en intégrant au sein de l'algorithme une auto-configuration des paramètres, qui sont ajustés automatiquement en fonction de la qualité des solutions obtenues, l'historique de recherche et d'autres critères.

Mots-Clés: métaheuristiques, Recherche tabou réactive, Recherche à voisinage variable, Problème du voyageur de commerce, Auto-configuration, Paramétrage en ligne.

Abstract

Most metaheuristic, such as Tabu Search and Simulated Annealing, though very efficient and useful in many practical applications, are very sensitive to their own parameters. For instance, Tabu search depends mainly on the size of tabu list whose optimal value can differ according to the problem instance being solved.

Designers usually resort to a series of experiments to select the suitable parameters (Off-line tuning), which is time consuming and requires a deep knowledge of the search strategy which is not the case for an end user.

Reactive Search Optimization provides a solution to this problem by including the parameter tuning mechanism within the search algorithm itself, parameters are adjusted by an automated feedback loop that acts according to the quality of the solutions found, the past search history and other criteria.

Keywords: metaheuristic, Reactive tabu search, Variable neighborhood search, travelling salesman problem, self-tuning, on-line tuning.