Titre: Title:	Environnement de programmation générique pour la recherche locale : Metalab
Auteur: Author:	Sylvain Crouzet
Date:	2005
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Crouzet, S. (2005). Environnement de programmation générique pour la recherche locale : Metalab [Ph.D. thesis, École Polytechnique de Montréal]. PolyPublie. https://publications.polymtl.ca/8197/

UNIVERSITÉ DE MONTRÉAL

ENVIRONNEMENT DE PROGRAMMATION GÉNÉRIQUE POUR LA RECHERCHE LOCALE : METALAB

SYLVAIN CROUZET
DÉPARTEMENT DE MATHÉMATIQUES ET DE GÉNIE INDUSTRIEL
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

THÈSE PRÉSENTÉE EN VUE DE L'OBTENTION
DU DIPLÔME DE PHILOSOPHIÆ DOCTOR (Ph.D.)
(MATHÉMATIQUES DE L'INGÉNIEUR)
DÉCEMBRE 2005

Your file Votre référence
ISBN: 978-0-494-48886-7
Our file Notre référence
ISBN: 978-0-494-48886-7

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Cette thèse intitulée :

ENVIRONNEMENT DE PROGRAMMATION GÉNÉRIQUE POUR LA RECHERCHE LOCALE : METALAB

présentée par : CROUZET Sylvain
en vue de l'obtention du diplôme de : Philosophiæ Doctor
a été dûment acceptée par le jury d'examen constitué de :

M. HERTZ Alain, Doct. ès. Sc., président
M. SAVARD Gilles, Ph.D., membre et directeur de recherche
M. HANSEN Pierre, D.Agr., membre
M. VAN HENTENRYCK Pascal, Ph.D., membre externe

À mes parents...

Remerciements

Je tiens tout d'abord à remercier mon directeur de recherche, Monsieur Gilles Savard, pour la grande confiance qu'il m'a faite tout au long de ce projet. En particulier, je tiens à le remercier pour l'aide financière et morale qui m'a été accordée avec beaucoup de générosité. De manière plus générale, je lui suis très humblement reconnaissant pour le cadre de travail exceptionnel qu'il m'a offert depuis mon arrivée au Canada en 1998.

Je tiens également à remercier mes collègues de bureau pour leur encouragement ainsi que leurs conseils toujours utiles. Partager avec eux le chemin qui mène au doctorat a été une expérience très enrichissante. En particulier, je voudrais exprimer ma gratidude envers Éric Rancourt sans qui la réalisation de cette thèse n'aurait certainement pas été possible. Je le rémercie pour sa disponibilité tout au long de mon apprentissage du C++.

Je voudrais aussi remercier l'ensemble du personnel, secrétaires, professeurs et chargés de cours, du GERAD et du département de mathématiques et de génie industriel de l'école polytechnique. C'est une grande chance que de pouvoir travailler dans un climat aussi professionnel que convivial.

Je souhaite également remercier très vivement mes collègues et amis. En particulier, je voudrais sincérement remercier Nikolaj, Caroline et Slim pour leur présence dans les moments difficiles.

Le doctorat, qui marque l'aboutissement de ma vie d'étudiant, est l'occasion d'exprimer ma profonde reconnaisssance et mon affection envers ma famille restée en

France. Je voudrais très particulièrement remercier mes parents dont la générosité et l'amour restent un modèle pour ma vie. Je leur suis très reconnaissant pour le soutien inconditionnel dont ils ont toujours fait preuve à mon égard. Cette thèse leur est dédiée.

Finalement, je tiens à t'exprimer ma gratitude, Nadia, pour cet enthousiasme que je trouve toujours auprès de toi. Tout le soin que tu as pris à m'entourer dans les moments difficiles est une lourde dette que je porte envers toi. Mes pensées vont bien aussi à Meriem et Salim qui m'ont accueilli auprès d'eux comme ma propre famille et dont l'affection a souvent fait la différence.

Résumé

Cette thèse porte sur les environnements de programmation consacrés à la recherche locale. Ce récent domaine d'étude, dont les avancées encourageraient certainement nombres d'applications, ne bénéficie pas d'un intérêt encore marqué. Peu de chantiers y sont en effet engagés en comparaison du volume de travaux touchant à la recherche locale. Sous d'autres latitudes, en programmation par contraintes, en nombres entiers, ou encore dans le domaine des graphes et réseaux, la réalisation d'ateliers de programmation bénéficie pourtant de climats stimulants. D'aucuns penseraient, non sans raisons, que la recherche locale se distingue par des schémas informatiques, certes représentables à un niveau formel, mais dont un langage ne saurait assimiler adéquatement les formes dégénérées prises au contact de situations particulières. Cette perspective ne manque pas d'affecter le secteur puisqu'elle prive, tant les practiciens, d'une méthode de programmation balisée, que les scientifiques, d'une harmonisation de leurs protocoles expérimentaux. Notre objectif est ici de montrer que des environnements de programmation peuvent pourtant répondre avec pragmatisme aux exigences de la recherche locale. Notre démarche est tout à la fois théorique et pratique. Théorique, car elle s'attache à faire la synthèse de connaissances portant sur la nature informatique de la recherche locale. Pratique, car elle se concentre progressivement sur la réalisation d'un utilitaire logiciel. Au plan théorique, la thèse affiche deux ambitions. Premièrement, nous mettons en perspective les différentes pensées informatiques sur lesquelles reposent la programmation assistée de la recherche locale. Cette synthèse permet au programmeur scientifique, habitué à une informatique procéduriale, d'assimiler et de comparer les différentes approches, lesquelles s'appuient sur une informatique tantôt déclarative, orientée-objet ou encore générique. Deuxièmement, nous proposons un modèle informatique original et néanmoins très logique

de la recherche locale. Ce modèle, fondé sur la programmation générique, sert de socle à l'architecture de l'utilitaire conçu par la suite. Au plan pratique, nous présentons METALAB, un atelier de programmation pour la recherche locale développé au cours de ce travail de thèse. Des études de cas illustrent l'apport du logiciel pour des tâches de programmation représentatives. D'une part, nous montrons comment utiliser METALAB pour résoudre de nouveaux problèmes à partir d'algorithmes rendus disponibles par l'atelier de programmation. D'autre part, nous montrons comment utiliser METALAB pour traduire de nouvelles idées algorithmiques.

La thèse débute par une classification des différentes approches rapportées dans la littérature. Trois classes d'environnements de programmation sont identifiées : les bibliothèques, les cadres d'application et enfin les langages déclaratifs. Nous définissons les bibliothèques comme des collections d'algorithmes préfabriqués mais seulement fonctionnels au sein d'une catégorie de problèmes pour lesquels le codage des solutions et mouvements demeure similaire. Les cadres d'application constituent également des collections d'algorithmes, qu'il reste toutefois à finaliser en y greffant des structures de solution et de mouvement relatives à un problème. Bien que souvent limitée à des formes de recherche locale très standardisées, cette deuxième approche, qui s'appuie sur le paradigme de la programmation générique, offre des patrons d'algorithmes valides quelle que soit l'application. Les langages déclaratifs se concentrent enfin sur les problèmes de satisfaction de contraintes. Pour ces problèmes, une itération de la recherche locale vise à modifier les valeurs d'une ou plusieurs variables de sorte à améliorer certaines grandeurs caractéristiques du degré de satisfaction du système de contraintes. Les langages déclaratifs permettent, d'une part, de formuler ces grandeurs en fonction des variables de décision et des données du problème, et d'autre part, d'exprimer les coordonnées variable-valeur des mouvements à sélectionner en fonction de ces grandeurs. La contribution principale de la thèse est de proposer un environnement de programmation d'un nouveau type. L'environnement que nous

présentons, baptisé METALAB, constitue une extension des cadres d'application que nous qualifierons de librairie générique.

De même que les cadres d'application rencontrés dans la littérature, le modèle informatique sur lequel repose METALAB emprunte à la programmation générique. Dans cette optique, le noyau du logiciel programme l'ossature d'une recherche locale en fonction de structures informatiques périphériques passées en paramètre par l'usager. Sur un plan formel, ces structures sont définies indépendamment d'un contexte d'application. À moins de limiter la portée de l'environnement à un champ restreint de problèmes ou d'algorithmes, il est par contre inconcevable d'anticiper leur forme et leur fonctionnement interne. La définition générique d'un type périphérique se borne donc à en spécifier l'interface en précisant quelles opérations doivent être rendues accessibles. Par exemple, METALAB définit un Attribut comme une pièce logicielle permettant de sonder une propriété donnée autour d'un mouvement. Un Attribut dispose pour cela d'une opération, notée `scan()`, qui calcule les champs de l'Attribut en fonction de l'état de la recherche associé au mouvement observé. La réalisation conforme aux spécifications de la librairie de chaque pièce logicielle manquante, par exemple d'un Attribut pour calculer le coût d'un mouvement, est à la charge de l'usager.

Une librairie générique se distingue toutefois des cadres d'application puisqu'au lieu de programmer des patrons génériques d'algorithmes, METALAB décompose la programmation de la recherche en autant d'unités logicielles que possible. Cette approche conduit à indroduire trois grandes familles de structures de données périphériques : les Représentations, les Variables et enfin les Contrôles. Les Représentations correspondent à des structures directement reliées au problème à résoudre. Les Solutions et Mouvements sont par exemple des types de Représentations. Les Variables correspondent, elles, à des mesures effectuées sur les Représentations rencontrées en cours de recherche. Ces Variables peuvent être temporaires ou mémorisées. Le coût

d'une Solution, d'un Mouvement, une liste taboue, ou encore toute propriété d'un Mouvement constituent des exemples de Variables. Enfin, les structures de Contrôle rassemblent les pièces de programme qui participent à la formulation de la trajectoire de recherche. Un Serveur, qui énumère des Mouvements correspondant au voisinage courant, un Explorateur, qui sélectionne un Mouvement à appliquer en fonction de certaines Variables qu'il observe, sont par exemple des formes de Contrôle. La syntaxe de METALAB, grâce à laquelle l'usager met en relation les différentes pièces logicielles pertinentes, permet finalement de composer, de moduler et d'expérimenter facilement les programmes de recherche locale désirés.

Un avantage des librairies génériques est qu'elles offrent un remarquable potentiel d'expressivité, de réutilisation et d'efficacité. D'abord, signalons qu'en sus de la programmation de Variables triviales directement connectées sur les Représentations d'un problème, il est possible de composer des Variables entre elles afin de monter des observations plus complexes. Cette possibilité favorise l'*expressivité* de la librarie. Précisons aussi que dans notre modèle, les structures de Représentation et de Contrôle n'interagissent autant que possible que par le biais des Variables. En recherche locale, le calcul d'une trajectoire (cf. types de Contrôle) est en effet effectué en fonction d'observations (cf. types de Variables), à leur tour calculées sur les mouvements et solutions (cf. types de Représentation). Comme on le verra sur le terrain du génie logiciel, le découplage qui en découle permet de garantir une *réusabilité* maximale de chaque unité de code. Remarquons enfin que la décomposition en pièces de programme favorise le perfectionnement ad-hoc des stratégies métaheuristiques sur les parties logicielles effectivement concernées. L'*efficacité* des programmes qu'on peut ainsi mettre au point est d'autant plus remarquable que METALAB s'appuie uniquement sur le polymorphisme statique. Tous les facteurs qui préfigurent la trempe d'un algorithme sont donc déjà pris en compte à la compilation, ce qui laisse un exécutable concentré exclusivement sur le déroulement de la recherche locale.

Il reste à préciser qu'une librairie générique ne prétend pas se substituer aux difficultés techniques relatives à la programmation d'une représentation de problème, d'une forme de mémoire ou de tout autre pièce de programme caractéristique de la recherche locale. Lorsqu'elles sont applicables, les approches de langages déclaratifs ou de bibliothèques, même si elles se livrent à une algorithmique ou à des domaines de problèmes plus restreints, sont à ce titre plus rentables. METALAB propose avant tout une solide méthode de développement : dans un premier temps, le programmeur réalise les pièces logicielles qui lui manquent conformément à leur définition générique, et dans un deuxième temps, il effectue une composition des pièces pertinentes à partir desquelles le langage génère son programme. D'une part, cette méthodologie constitue un excellent outil de gestion de projet. Elle permet en effet de diviser la tâche de programmation, de former des pièces de programmes réutilisables et de recomposer facilement des programmes. D'autre part, cette méthodologie encourage l'accumulation et l'échange de savoir-faire. Pour le chercheur en algorithmique, elle permet en effet de former de nouvelles métaheuristiques qu'on peut tester rapidement sur des représentations de problèmes témoins disponibles à travers le logiciel. Le practicien, qui propose de son côté de nouveaux problèmes, dispose à son tour de schémas algorithmiques récupérables. En ce sens, METALAB constitue un intéressant cadre laboratoire pour la recherche locale.

Abstract

This dissertation is concerned with local search programming environments. As those ones may constitute the ground of many applications, this area of study deserves a particular interest. However, while other paradigms like constraint programming, integer programming or graph computing have stimulated the creation of programming environments, few work has still been done in the field. One may observe that local search contrasts with other solving methods as it presents computing schemes, representable at a formal level, but for which a programming langage would hardly take into account the degeneracies once applied to a given problem. This statement unfortunately does not help as it prevents the practionners to have a well-known programming methodology and the scientists to harmonize their experimentations. We show, in this thesis, how programming environments may answer to the challenges of local search computing. The approach is both theoretical and practical. Theoritical, as we propose to make a synthesis of the computing nature of local search. Practical, as we focus on the creation of a programming environment. We follow two goals on the theoritical level. We first give an outlook of the different computing paradigms on which can be engineered the local search programming tasks. For those used to procedurial langages, it should help to assimilate and compare the declarative, object-oriented or generic features of the different programming environments proposed in the literature. Second, we propose an original computing model of local search. This model, which profiles the design of our software, is based on generic programming. In practical, the thesis presents the software METALAB which has been developped during our work. Some case studies illustrate the different features of the langage. On one side, we show how to use METALAB to solve new problems on the basis of algorithms distributed by the programming environment. On the other side, we show how to use METALAB to express new metaheuristics.

The dissertation begins with a classification of the different approaches found in the literature. Three classes of programming environments are identified: the librairies, the frameworks and the declarative langages. By a library, we mean a collection of prebuilt algorithms but only suitable for a family of problems for which the computing of the solutions and movements remains similar. Frameworks also collect algorithms but in the form of patterns to be finalized once they are connected to some user-defined types of solutions and movements. While those patterns are limited to standard local search schemes, they remain valid whatever the problem to solve. Declarative langages focus on constraint satisfaction problems. For those problems, a movement of the local search consists of modifying the value of one or several decision variables so that some quantities related to the degree of satisfaction of the system are improved. A declarative langage allows, first, to formulate those quantities upon the decision variables and the datas of the problem, and second, to compute the coordinates variable-value of the movement to be chosen according to those quantities. The main contribution of the thesis is to propose a new type of programming environment. The software we present, called METALAB, can be viewed as an extension to the framework approach.

Like the frameworks reported in the literature, METALAB relies on the paradigm of generic programming. In this model, the kernel of the software computes the skeleton of a local search according to some peripherical data structures to be given by the user. So as to keep the scope of the environment as general as possible, those peripherals are just described on a formal level independently of any application. The generic definition of a type, which concentrates on its interface, gives the set of services it must yield. For instance, METALAB defines an Attribute as a type with an operation, called scan(), that allows to measure a given property related to a movement. For a given problem, the user is commited to compute those functionalities according to the METALAB specifications.

Otherwise, METALAB distinguishes with the frameworks given in the literature as it does not rely on a set of patterns of well-known algorithms. As well as the representation of a problem has been decomposed by the computing of movements and solutions in the framework approach, our software also proposes to decompose the computing of the search process. This approach introduces three families of peripherical data structures: Representation types, Variable types and Control types. Representations are concerned with those structures related to the problem. Solutions and Movements, for instance, are forms of Representation. Variables are concerned with observations done on some Representation states encountered during the search process. Variables may be temporal or memorized. The cost of a Solution, of a Movement, a tabu list, or any Attribute of a Movement are examples of Variables. Finally, Control types are those ones that contribute to the search process. A Server, that enumerates some Movements according to the neighborhoods of the curent Solution, an Explorer, that selects one of those Movements to be applied are some examples of Control types. The syntax of METALAB, which is used to coordinate all the different pieces, allows to generate, modulate and experiment the final local search program.

As illustrated, the model of METALAB offers a great potential of expressivity, reusability and efficiency. First, let us point out that, allong with the computing of simple Variables directly connected to some Representation types, it is possible to compose Variables together so as to express complex observations. This first feature gives the environment a great *expressivity.* Also, let us notice that our model minimizes the communication between Representation types and Control types which is done via Variables. It is the case in local search where the decision of the search path (cf. Control types) relies on some observations (cf. Variable types) made in turn on the Solutions and Movements (cf. Representation types). On a software engineering level, this second feature allows to warrant an optimal *reusability* of each piece of program. We finally argue that the decomposition into elementary pieces of code promotes

the adaptation of metaheuristics as one can concentrate its work on the code truly concerned. Notice that the very polymorphic feature of METALAB, which allows to conceive *efficient* algorithms, has no cost at run-time. As METALAB only relies on static polymorphism, all the factors that shape the final algorithm are in fact taken into account at compile-time. Executables are then exclusively processed toward the run of the local search.

It remains to precise that METALAB does not help to compute any technicities related to a given local search application. Although they are limited to a given scope of problems and algorithms, the libraries or declarative langages approaches are more concerned with this goal. The software we propose constitutes above all a strong development methodology which isolates those technicities and allows to store the know-how they are related to. METALAB drives the user to create the missing pieces for his project and then to compose them to generate the desired local search. On the one hand, this methodology gives an excellent tool to manage local search projects. It allows to divide the computing task, to create reusable pieces of code, and to easily recompose programs. On the other hand, this methodology allows to store and to share the state of the art. The one interested in testing new metaheuristics may for instance benefit from some test problems that have already been represented in METALAB. In its turn, the practitionner may benefit from algorithms provided through METALAB to solve his new problems. All in all, METALAB can be viewed as the framework of a local search laboratory.

Table des matières

Liste des tableaux

Liste des figures

Liste des annexes

Introduction

La recherche locale constitue une approche algorithmique remarquable pour la résolution de problèmes d'optimisation et de satisfaction de contraintes. Elle propose une recherche de solutions heuristique où l'exactitude ainsi que l'estimation de l'erreur commise sont délaissées au profit d'une grande efficacité à générer une succession de réponses progressivement satisfaisantes. Le traitement d'un grand nombre de problèmes dans le cadre de cette approche témoigne d'un intérêt tout particulier. La nature heuristique de la recherche locale, dont il ne faut pas manquer de relativiser les insuffisances à l'égard d'une modélisation souvent déjà approximative, permet en effet d'aborder la résolution de problèmes autrement intraitables dans des temps de calcul raisonnables.

Un algorithme de recherche locale exerce un parcours dans un graphe dont chaque nœud dénote une solution et chaque arc, le mouvement d'une solution qu'on transforme en une solution voisine. La puissance et l'apparente canonicité des algorithmes appartenant à ce paradigme ont largement contribué à faire de la recherche locale un domaine d'étude privilégié. L'expérimentation de différents types de graphes pour certaines classes de problèmes a ainsi permis de comprendre les critères conduisant au choix d'une structure de solution et d'un schéma de mouvement convenables pour la résolution efficace d'un problème donné. De même, les algorithmes de parcours, initialement fondés sur l'amélioration monotone d'un critère objectif, ont été perfectionnés par l'intégration de mécanismes métaheuristiques permettant de débloquer la recherche au-delà de solutions localement optimales. Avec l'apparition de ces mécanismes dont la méthode taboue et le recuit-simulé constituent les exemples les plus célèbres, la recherche locale a été souvent perçue comme une collection de méthodes

bien distinctes mais relevant d'un même principe. Les constructions algorithmiques ont été malgré tout étudiées transversalement et des notions essentielles comme l'intensification et la diversification ont été approfondies. L'actuelle convergence entre méthodes de recherche locale et méthodes à base d'une population de solutions traduit cette vision composite. Ces avancées ont été naturellement portées par des progrès informatiques matériels, comme la parallélisation des exécutables et la puissance grandissante des calculateurs, mais aussi par des progrès informatiques logiciels, notamment dans le souci d'intégrer la recherche locale comme solveur sous-jacent à d'éventuels systèmes d'aide à la décision.

La recherche locale connaît malgré tout des limites que d'autres paradigmes ont franchi avec davantage de succès. Il est ainsi remarquable qu'un cadre de résolution aussi populaire ne dispose pas d'un langage de programmation propre et largement reconnu. C'est pourtant chose faite en programmation par contraintes, en programmation en nombres entiers ou encore dans le domaine des graphes et réseaux, où des logiciels permettent de vulgariser la création d'applications. Sans doute, la recherche locale est un domaine bien plus complexe qu'il n'y paraît et la formalisation de ses schémas informatiques, une étape nécessaire à la réalisation d'un langage, constitue un véritable défi. Ceci est d'autant plus critique que de nouveaux mécanismes métaheuristiques sont fréquemment proposés et que les algorithmes déjà existants sont souvent modifiés afin de mieux tenir compte des spécificités des problèmes résolus. Des techniques appropriées de génie logiciel ont toutefois permis d'exploiter différents jeux d'hypothèses réduisant la nature informatique de la recherche locale. Trois grandes approches ont été jusque là prometteuses.

- Dans la première approche, on choisit de se concentrer sur une famille cohérente de classes de problèmes. Les ateliers de programmation qui en découlent se présentent comme une *bibliothèque* d'algorithmes clef-en-main. Le recours à un tel environnement logiciel se limite naturellement au cadre rigide des problèmes ciblés

par la bibliothèque. En revanche, les algorithmes dont on dispose peuvent prendre les formes spécifiques propres au domaine pris en charge.

- Dans la deuxième approche, on cherche à découpler le schéma algorithmique relatif à une recherche locale des structures de données relatives à un problème particulier. Les ateliers de programmation qui en découlent prennent la forme d'un *cadre d'application* : le noyau du cadre d'application implémente l'ossature d'une recherche locale en fonction de structures de données périphériques que l'usager programme et passe en paramètre. Le découplage et l'interfaçage des deux parties logicielles sont ici de nature à restreindre les schémas de recherche locale à des formes très standardisées. Pour ces formes, l'usager bénéficie en revanche de patrons d'algorithmes génériques pouvant être utilisés pour les structures de représentation de problème qu'il passe en paramètre.

- Dans la troisième approche, on tire profit des formes de recherche dont un mouvement consiste à réaffecter une ou plusieurs variables. Ce cadre, qu'on rencontre en satisfaction de contraintes, permet de réduire une recherche locale à la formulation de grandeurs caractéristiques en fonction desquelles s'expriment les coordonnées d'une réaffectation. Ces environnements constituent des *langages déclaratifs* : l'usager donne une expression aux grandeurs caractéristiques dont le langage maintient la validité et en déduit la trajectoire de recherche automatiquement.

Le projet de recherche que rapporte cette thèse concerne la formalisation, la réalisation ainsi que l'expérimentation d'une nouvelle approche : les *librairies génériques.* Ce type d'atelier de programmation, dont l'architecture repose sur un modèle informatique innovant ainsi que sur des techniques de programmation génériques avancées, vise principalement les qualités de *généricité* et d'*expressivité.*

- De même que pour les cadres d'application, le type d'atelier visé est *générique.* Le codage ainsi que le fonctionnement d'une pièce de programme reste donc indépendant du type de problème à résoudre et/ou des spécificités de l'algorithme utilisé.

Les bibliothèques ou les langages déclaratifs réduits par construction, soit à une famille de problèmes donnée, soit aux problèmes de satisfaction de contraintes, ne permettent pas d'atteindre cette généricité. De même que les cadres d'application, une librairie générique prévoit de découpler, d'une part, la programmation des éléments constituant le graphe de recherche, et d'autre part, la programmation des mécanismes autour desquels se décide la trajectoire de recherche.

- Les cadres d'application permettent d'isoler la représentation des problèmes en mouvements et solutions en face de patrons d'algorithmes préfabriqués. Une librairie générique, qui décompose non seulement la représentation du problème mais aussi l'algorithmique de la recherche locale, prolonge nettement cette approche. Au besoin, ce type d'environnement permet en fait de construire ainsi que de composer n'importe quelle pièce élémentaire de la méthode de résolution visée. La programmation devient modulaire à tous les niveaux, tant pour les structures de problème que pour les mécanismes de recherche. Cette caractéristique favorise l'*expressivité* des librairies génériques.

Avec l'usage, une librairie générique se présente comme un couple de bibliothèques extensibles. L'une rassemble des structures de problème, l'autre, des structures de contrôles. Lorsqu'ils sont compatibles, les éléments de ces bibliothèques peuvent être mis en relation par l'usager. Le noyau du logiciel génére alors l'algorithme correspondant. Cette approche privilégie un certain nombre de situations propres à l'activité de programmation en recherche locale.

- Elle encourage la recherche en algorithmique. Des usagers intéressés à développer de nouvelles métaheuristiques disposent en effet d'une grande modularité permettant de traduire nombres de mécanismes pour contrôler la trajectoire de recherche. Ils bénéficient par ailleurs de structures de problèmes témoins que réunit la bibliothèque de problèmes et qu'alimentent d'autres types d'usagers.

- Elle encourage la résolution de nouveaux problèmes. Les usagers désireux de résoudre un problème particulier disposent en effet de structures de recherche réutilisables et mises à contribution par les usagers oeuvrant en algorithmique. La modularité de la méthode de développement permet en outre de concentrer l'adaptation des algorithmes pour un problème donné sur les structures de contrôle effectivement concernées.

- Elle permet finalement de constituer des laboratoires où se rencontrent et s'accumulent l'expertise algorithmique et la connaissance de problèmes particuliers. Ces laboratoires peuvent être mis en oeuvre à l'échelle d'une équipe de recherche pour une famille donnée de problèmes. Une librairie générique constitue dans ce cas un très bon outil de gestion de projet. Ces laboratoires peuvent aussi être mis en oeuvre à l'échelle d'une communauté de chercheurs. Une librairie générique permet dans ce cas d'harmoniser les protocoles expérimentaux et de synthétiser le savoir-faire relatif au domaine.

Le travail présenté s'articule en trois phases. Premièrement, la phase d'étude s'attache à dessiner les contours d'un langage de programmation pour la recherche locale. Cette étape décrit l'intérêt, la réalisabilité, les conditions de fonctionnement ainsi que les limites des différentes approches possibles. Deuxièmement, la phase théorique propose de formuler la recherche locale dans le paradigme de la programmation générique. Cette étape donne un modèle informatique à la recherche locale en même temps qu'elle fournit un canevas pour l'architecture des librairies génériques. Enfin troisièmement, la phase d'expérimentation démontre la pertinence du modèle théorique par la présentation et la mise à l'épreuve de la librairie générique METALAB. À noter que chacune de ces trois parties a été rédigée le plus indépendamment possible des notions mises en jeu par ailleurs de sorte que le lecteur puisse parcourir l'ouvrage dans l'ordre qui correspond à ses intérêts et à sa méthode, selon qu'il vient des mathématiques ou de l'informatique, selon qu'il cherche à théoriser la recherche locale ou à en faire l'application. Le contenu de chaque partie est organisé comme suit.

La première partie donne un aperçu des enjeux poursuivis dans la thèse. Une synthèse de la recherche locale est proposée en préambule. La notion de graphe de recherche, constitué des solutions et mouvements, amorce l'exposé. Les formes heuristiques concourant à orienter le parcours sur ce graphe sont ensuite examinées. Quelques exemples d'algorithmes représentatifs complètent ce préalable. L'analyse informatique de la recherche locale se poursuit par l'examen des différents styles de programmation qu'on y rencontre et par une étude de réalisabilité concernant les environnements de programmation. L'intérêt et les conditions de fonctionnement de chacune des trois approches existantes - bibliothèques, cadres d'application et langages déclaratifs - sont passés en revue. Les caractéristiques des librairies génériques sont finalement mises en relief.

La deuxième partie propose de formuler la recherche locale dans le paradigme de la programmation générique. Trois principes de modélisation, conduisant à une lecture originale de la recherche locale, sont discutés. Le *principe de découplage*, qui introduit trois familles de pièces logicielles, favorise une réusabilité maximale du code. La première famille concerne les entités relatives au graphe de recherche, comme les solutions et mouvements, la deuxième, les observations consultées et/ou archivées sur le parcours de recherche, et la troisième, les structures de contrôle, qui opèrent un parcours sur le graphe de recherche en fonctions des observations reçues. Pour une combinaison de pièces logicielles, le *principe de modulation* permet d'effectuer la synthèse d'algorithme la plus efficace possible en fonction des parties entrant effectivement en interaction. Le *principe de composition*, qui autorise de définir une observation en fonction d'observations déjà existantes, étend enfin l'expressivité du modèle. La partie théorique se poursuit par la spécification de chaque unité logicielle dans le formalisme d'une librairie générique. Conformément à ces spécifications, il devient possible de définir de nouveaux types de fonctionnement pour chaque catégorie de pièces de même que les entités existantes peuvent être combinées différemment

afin de synthétiser de nouveaux programmes. Le volet théorique s'achève par l'interprétation du modèle. Le cycle de vie et les modes d'interaction de chaque catégorie de pièces logicielles y sont examinés. Quelques aspects techniques concernant l'implantation de ce modèle sont parallèlement présentés en annexe. Ce volet technique tend à démontrer que la surcharge de calcul des programmes générés par une librairie générique comparativement à des algorithmes codés à la main reste très négligeable.

La troisième partie, qui présente l'utilitaire METALAB, démontre la faisabilité ainsi que l'intérêt des librairies génériques. Cette partie débute par une description didactique de METALAB permettant à l'usager de programmer les applications qui l'intéressent. Quelques études de cas représentatives des possibilités qu'offre la librairie générique sont enfin exposées. Les scénarios présentés démontrent la simplicité avec laquelle il est possible d'assurer la gestion d'un projet en recherche locale. Utiliser METALAB demande en effet peu d'investissement concernant l'écriture du code initiant le projet tout en permettant d'apporter des modifications significatives dans la poursuite du projet. La réalisation de nouvelles méthodes taboues efficaces pour la résolution du voyageur de commerce illustre enfin l'apport de la librairie METALAB pour la conduite de projets avancés.

Après avoir révisé les avantages et inconvénients d'une librairie générique, nous proposons en conclusion quelques extensions pertinentes. Les mécanismes de coordination propres aux algorithmes à base d'une population de solutions et aux méthodes utilisant plusieurs schémas de mouvements sont par exemple passés en revue. Chaque langage informatique véhiculant une forme singulière de pensée, le concours de cette thèse appartient en grande partie aux lecteurs pour qui le modèle proposé paraîtra naturel. Nous espérons en ce sens que la valeur didactique de cette thèse en constitue la principale contribution. Des contributions plus manifestes peuvent être par ailleurs résumées comme suit.

- Une étude transversale des environnements de programmation déjà existants pour la recherche locale est proposée. Cette étude nécessite une synthèse préalable de la recherche locale indépendante des a priori relatifs à l'optimisation et à la satisfaction de contraintes.

- Un modèle générique de recherche locale est donné autour des trois principes de découplage, de modulation et de composition. Les deux premiers principes, qu'on retrouve au coeur de la STL, la librairie générique de structures de données du C++, ont été transposés à la recherche locale. Le troisième principe est propre à la recherche locale. Le modèle proposé, qui constitue le socle de la librairie METALAB, fournit une grille de lecture originale et néanmoins très logique de la recherche locale.

- Une librairie générique est proposée pour la création d'applications de recherche locale. Cet utilitaire fournit des conditions très réalistes où il reste possible, pour les programmeurs intéressés à la résolution d'un problème, d'adapter les pièces algorithmiques pertinentes à celles du problème, et pour les programmeurs en algorithmique, de transposer en pratique de nouvelles idées heuristiques.

- De nouveaux algorithmes tabous pour le voyageur de commerce sont proposés. Les techniques exposées montrent leur efficacité sur des problèmes allant jusqu'à quelques milliers de nœuds. Ces algorithmes se basent sur la segmentation du voisinage 2-*opt* en se limitant à des mouvements n'inversant que de courtes sous-séquences de la solution courante. Les diverses techniques d'incrémentation possibles pour le voisinage 2-*opt* sont également étudiées. Enfin, des techniques d'anticipation sont proposées pour renforcer la couverture de la recherche en phase d'intensification.

Première partie

Analyse informatique de la recherche locale

La première partie de la thèse introduit le contexte et les éléments de réflexion qui ont motivé la réalisation d'un utilitaire de programmation pour la recherche locale. L'analyse de la problématique comprend deux chapitres.

Dans le chapitre 1, nous dressons un portrait synthétique de la recherche locale. Après avoir encadré les principes de fonctionnement et les modes heuristiques de ce paradigme, nous passons en revue quelques algorithmes représentatifs du domaine. Nous discutons enfin de la pertinence de la recherche locale pour des situations auxquelles peuvent être confrontés les programmeurs autour des quatre critères de congruence, de navigabilité, d'ergonomie et de topologie. Le chapitre se termine par une mise en perspective des idées avancées dans l'exposé.

Dans le chapitre 2, nous discutons des approches possibles pour le développement de métaheuristiques sous un environnement de programmation spécialisé. Nous commençons par décrire les différents styles de programmation rencontrés et analysons dans chaque cas les besoins d'un langage. Nous discutons alors de la réalisabilité d'un utilitaire de programmation et passons en revue les trois types d'approche - bibliothèques, cadres d'application et langages déclaratifs - présentées dans la littérature. Nous établissons enfin l'intérêt d'une approche innovante offrant davantage de flexibilité et de possibilités que celles existantes pour des objectifs que nous préciserons.

Cette approche, qui repose principalement sur la programmation générique, peut être vue comme un prolongement des cadres d'application.

CHAPITRE 1 : MÉTHODES DE RECHERCHE LOCALE

Au même titre que la programmation en nombres entiers (Nemhauser et Wolsey, 1988), la programmation par contraintes (Marriot et Stuckey, 1998) ou la programmation linéaire et non linéaire (Luenberger, 1984), la recherche locale (Reeves, 1993) constitue un paradigme de résolution pour les problèmes d'optimisation et de satisfaction de contraintes. Ce paradigme ne propose pas de résolution exacte aux problèmes d'optimisation : la solution apportée par un algorithme n'est pas forcément optimale. Il ne propose pas non plus de résolution complète aux problèmes de satisfaction : le fait qu'un algorithme ne trouve pas une solution réalisable ne signifie pas qu'il n'en existe pas. C'est dans l'efficacité à trouver de bonnes solutions que réside l'intérêt de la recherche locale. En optimisation, l'expérience montre en effet que des solutions quasi-optimales peuvent être généralement trouvées dans des temps de calcul très courts sinon raisonnables comparativement à ceux rédibitoires des méthodes exactes (Johnson et McGeoch, 1997). De même, en satisfaction de contraintes (Bohlin, 2002), des problèmes de grandes tailles peuvent être traités (Gu 1992, Hoos et Stutzle, 1999, Hoos et Stutzle, 2000). Des solutions réalisables peuvent être facilement générées dans le cas sous-contraint alors que dans le cas sur-contraint le blocage de la résolution sur des solutions fortement non-réalisables peut fournir une indication de non-réalisabilité. On trouve ainsi une motivation toute particulière à utiliser la recherche locale pour la résolution de problèmes réputés difficiles pour lesquels des méthodes exactes ou complètes demanderaient des temps de calcul trop longs pour des tailles de problèmes intéressantes.

1.1 Principes de la recherche locale

En recherche locale, la construction d'une réponse s'effectue en modifiant des solutions intermédiaires par des transformations successives qu'on appelle mouvements. Ces mouvements suivent un schéma préétabli qu'on peut noter par une fonction $M : (s, c) \rightarrow M(s, c)$. Étant donnée une solution courante s, le schéma de mouvement peut être paramétré par des coordonnées c différentes, pour lesquelles on obtient le mouvement $M(s, c)$. On notera $s \oplus M(s, c)$ la nouvelle solution obtenue en appliquant un tel mouvement. On note M_s l'ensemble des mouvements définis autour de la solution s, et $V_M(s) = \{s \oplus m, m \in M_s\}$, l'ensemble des solutions voisines de s. Comme il apparaît dans les deux exemples suivants, il est souvent possible de visiter le voisinage d'une solution en énumérant sur les coordonnées pertinentes.

Problème du voyageur de commerce. Considérons une version du problème du voyageur de commerce (Reinelt, 1991), qu'on notera PVC. Soit un graphe complet non dirigé $G = (V, E)$ où V désigne l'ensemble des nœuds et E l'ensemble des arêtes. Étant donnée une fonction coût positive $C : e \rightarrow C(e)$ définie pour chaque arête $e \in E$, le PVC consiste à trouver un circuit de moindre coût qui passe par tous les nœuds. Comme illustré sur la figure 1.1, étant donné un indexage de l'ensemble des nœuds $v \in V$, on représente une solution du PVC par une permutation d'entiers sur l'ensemble $\{1...|V|\}$. Un exemple de schéma de mouvement appelé 2-*opt* consiste alors à choisir une sous-séquence de la permutation et à l'inverser. Dans ce cas, un système de coordonnées possible consiste à désigner les deux arêtes, notés $\{c_{00}, c_{01}\}$ et $\{c_{10}, c_{11}\}$, sortant de la tournée courante. En tenant compte des symétries et en éliminant l'inversion des sous-séquences vides ou singletons, on obtient $\frac{1}{2}|V|.(|V|-3)$ mouvements possibles pour chaque solution. La figure 1.1 ci-dessous présente une solution et son voisinage pour le PVC d'un graphe complet à cinq nœuds.

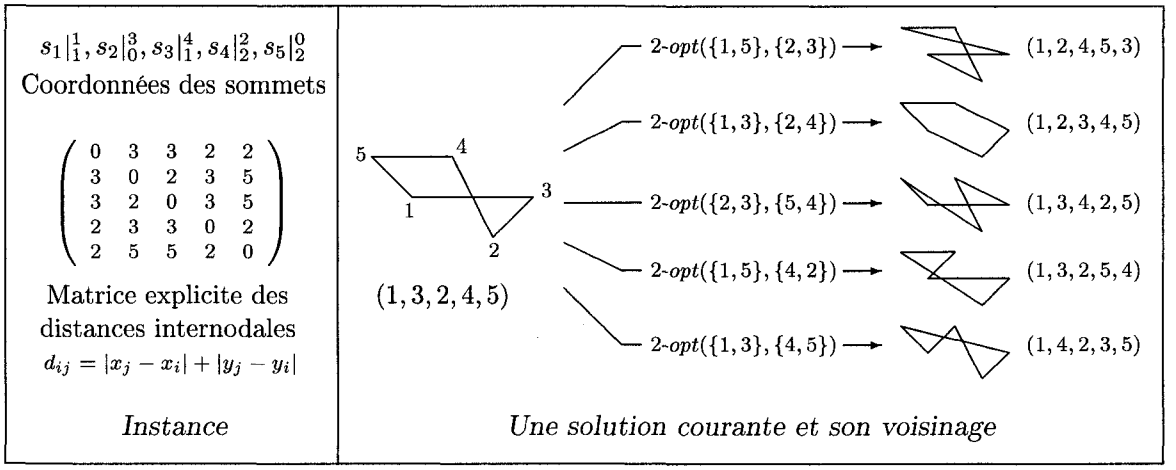


Figure 1.1 – Une solution et son voisinage 2-*opt* pour une instance de PVC

Problème de satisfaction de contraintes. Considérons une version du problème de satisfaction de contraintes, qu'on notera PSC. On se donne un ensemble de variables $(X_i)_{i \in I}$, chacune associée à un domaine fini D_i de valeurs, ainsi qu'un ensemble de contraintes reliant ces variables. Il s'agit de trouver une affection $v_i \in D_i$ à chacune des variables de sorte que toutes les contraintes du système soient vérifiées. Dans ce contexte, on représente une solution par une affectation $(v_i)_{i \in I}$, c'est-à-dire $X_i = v_i, \forall i \in I$. La classe de problèmes SAT se restreint ici au cas où chaque variable est binaire et chaque contrainte une disjonction de littéraux impliquant un sous-ensemble des variables. Un schéma de mouvement souvent utilisé, appelé $flip$, consiste à choisir une variable, par exemple celle qui apparaît dans le plus grand nombre de contraintes violées, pour en modifier la valeur. Dans ce cas, les coordonnées $c = (i, v_i)$ d'un mouvement spécifient la variable réaffectée ainsi que la nouvelle valeur retenue. Pour une variable i donnée, on obtient $|D_i| - 1$ mouvements possibles. La figure 1.2 présente l'ensemble des solutions et mouvements pour un SAT à trois variables et quatre contraintes. Pour chaque solution, seuls sont considérés les mouvements impliquant la (ou les) variables comportant un maximum de conflits.

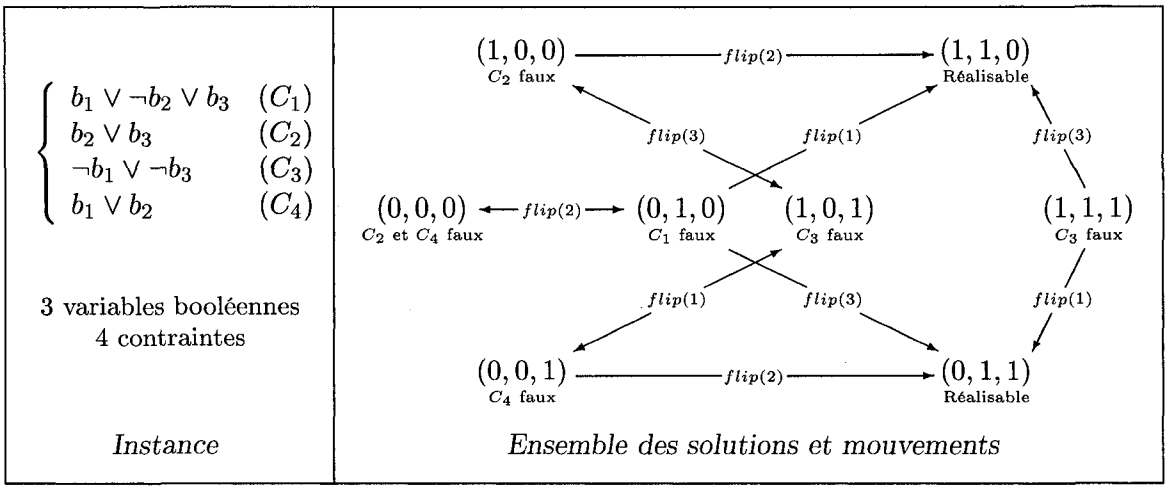


Figure 1.2 – Solutions et mouvements $flip$ pour un problème SAT

Étant donnée une représentation des solutions ainsi qu'un schéma de mouvement M, on se donne l'ensemble S des valeurs possibles de solutions ainsi que l'ensemble $A = \{(s, s'), s' \in V_M(s), s \in S\}$ des transitions qu'on considère entre deux solutions quelconques. On désigne par espace de recherche (ou graphe de recherche) le graphe dirigé $E = (S, A)$. Dans cette thèse, on considère qu'un programme de recherche locale est un programme qui génère un parcours de recherche dans l'espace de recherche E. Cette définition s'étend naturellement au cas où plusieurs schémas de mouvement sont utilisés. On posera par exemple que $A = \{(s, s'), s' \in V_{M_1}(s) \cup V_{M_2}(s), s \in S\}$ si deux schémas $M1$ et $M2$ sont simultanément utilisés.

Concernant l'espace de recherche, on souhaite en général que l'ensemble des transitions définisse un graphe connexe (Hertz, Taillard et De Werra, 1995) de sorte que toutes les solutions - dont les solutions optimales en optimisation et les solutions réalisables en satisfaction de contraintes - puissent être éventuellement atteintes. Puisque l'inversion des sous-séquences de taille 2 permet de générer l'ensemble des permutations, l'espace de recherche pour un PVC muni d'un 2-*opt* est connexe.

Par contre, le PSC pour lequel l'ensemble de variables $(X_i)_{i=1,2,3,4,5} \in \{0,1\}^5$ est contraint par les deux inégalités $X_1 + X_2 + X_3 \leq 1$ et $X_1 + X_4 + X_5 \leq 1$ présente deux solutions $(1,1,1,1,1)$ et $(0,1,1,1,1)$ à partir desquelles aucun mouvement $flip$ ne permet d'accéder au reste de l'espace de recherche si on se limite aux variables de conflit maximal. Pour des problèmes de satisfaction de contraintes où les variables sont booléenes, cette version limitée de $flip$ se réduit en effet, dans les cas non dégénérés où une seule variable correspond au maximum de conflits, à un seul voisin par solution. Une alternative consiste alors à étendre le voisinage en permettant de modifier la valeur d'une variable quelconque. Le graphe de recherche obtenu, désormais connexe peu importe l'instance, est présenté à la figure 1.3 pour le problème SAT précédent.

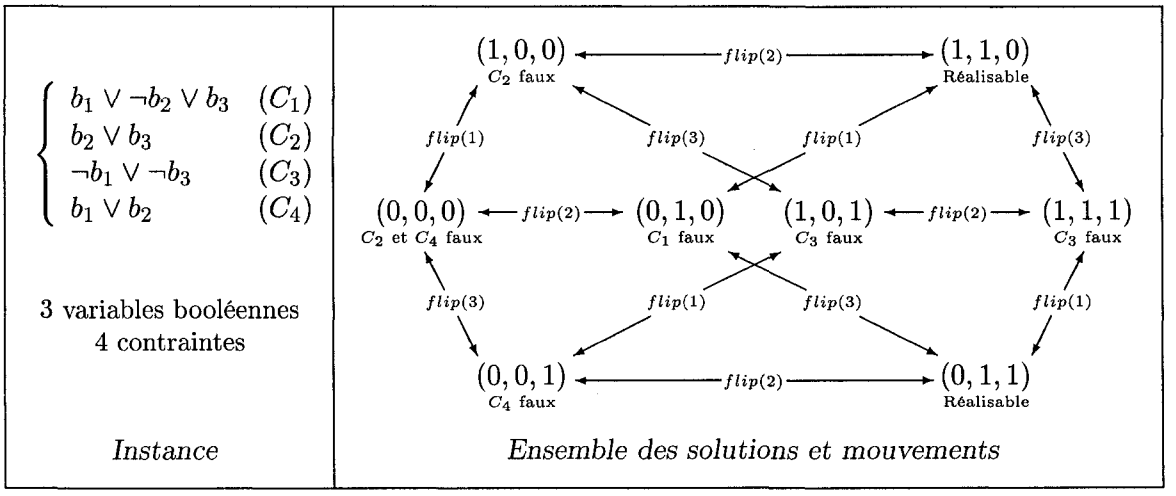


Figure 1.3 – Graphe de recherche connexe pour un problème SAT

Dans le cadre d'algorithmes simples, la recherche est de nature linéaire et markovienne. Nous dirons que la recherche est linéaire lorsque le parcours des solutions visitées traverse un chemin fini $(s_0, s_1, ..., s_L)$ où $s_{i+1} \in V_M(s_i), \forall i = 0, ...L-1$. Nous dirons que la recherche est markovienne lorsqu'une itération de la recherche,

c'est-à-dire le choix et l'application d'un mouvement, ne dépend pas de l'historique de la recherche mais seulement de la solution courante. Pour des algorithmes plus complexes, la recherche locale s'éloigne souvent du cadre linéaire et markovien. Par exemple, lorsqu'on développe un ensemble de recherches locales parallèles, plusieurs chemins de recherche coexistent et, c'est souvent le but recherché, interfèrent. Il est également possible que le choix d'un mouvement dépende non seulement de la solution courante mais aussi de la trajectoire de la recherche.

Étant donné un espace de recherche, décrire un algorithme de recherche locale revient habituellement à en préciser les critères d'arrêt, le système de règles de sélection d'un mouvement et la procédure d'énumération du voisinage. Les critères d'arrêt arbitrent la sortie du programme. Ces critères peuvent porter par exemple sur le nombre maximal d'itérations à effectuer, sur la vitesse marginale à laquelle de meilleures solutions sont trouvées ou encore sur tout critère estimant la stagnation de la recherche. Le critère d'arrêt peut enfin tout simplement consister à donner une borne sur le temps d'exécution alloué. Le système de règles de sélection arbitre la sélection du mouvement conduisant à modifier la solution courante. Il existe plusieurs niveaux de sélection : certaines règles réclament d'écarter certains mouvements, d'autres permettent de les comparer entre eux. Certaines règles permettent également de court-circuiter l'exploration du voisinage courant lorsqu'un mouvement visité est jugé satisfaisant. La procédure d'énumération du voisinage détermine enfin le sous-ensemble et l'ordre dans lesquels sont explorés les mouvements d'un voisinage. Généralement, la totalité du voisinage courant est potentiellement explorable à chaque itération. Dans le cas de voisinages à grandes échelles, comme par exemple les voisinages à profondeur variable (Lin et Kernighan, 1973, Ahuja, Ergun, Orlin et Punnen, 2002, Applegate, Cook et Rohe, 2003), il peut être cependant plus efficace de limiter l'énumération à une partie judicieuse du voisinage. Les coordonnées des mouvements sont souvent parcourues de manière séquentielle, aléatoire ou encore selon leur gain.

1.2 Les formes heuristiques en recherche locale

Les algorithmes de recherche locale appartiennent à la classe des heuristiques. À ce titre, la programmation d'une recherche locale occasionne la mise en place de mécanismes intuitifs visant à orienter la recherche vers des zones jugées prometteuses (Pearl, 1984). En intelligence artificielle, la mise en oeuvre de ces mécanismes repose typiquement sur la transposition d'une expertise relative au phénomène modélisé et/ou sur une connaissance statique sinon dynamique de l'espace de recherche. En recherche locale, l'intégration de mécanismes heuristiques s'appuie particulièrement sur l'acquisition dynamique d'information. D'une part, soulignons que la génèse de cette information est d'autant plus nécessaire que les renseignements dont on dispose pour diriger la recherche locale sont au départ très limités. Par construction, le parcours des solutions intermédiaires est en effet markovien (la solution courante et son voisinage constituent les seules sources d'information) et linéaire (l'information balayée se focalise sur une seule dimension). Mais d'autre part, notons que la maintenance et l'utilisation d'une information supplémentaire peuvent s'avérer coûteuses en temps de calcul comme en mémoire. L'intérêt d'un mécanisme heuristique est donc souvent relatif à la complexité de calcul qu'il induit. Cette section propose de décrire les différentes formes heuristiques rencontrées en recherche locale. Nous commentons d'abord deux grands principes heuristiques qu'une recherche locale s'emploie généralement à suivre, soit le principe d'amélioration et le principe d'intensification/diversification. Nous passons ensuite en revue quelques techniques permettant de réaliser ces principes. Les algorithmes que nous décrivons dans la section 1.3 illustreront par la suite cet exposé.

On se donne pour la suite une fonction coût $\delta(m)$ définie pour tout $m \in \cup_{s \in S} M_s$. Étant donnée une fonction coût associée aux solutions $\psi : s \rightarrow \psi(s)$ définie pour chaque $s \in S$, $\delta(m)$ mesure habituellement (mais pas forcément) le coût différentiel

d'un mouvement, et dans ce cas, $\psi(s \oplus m) = \psi(s) + \delta(m)$. Par exemple, pour le PVC muni d'un 2-*opt*, on définit $\psi(s)$ par l'aggrégation des coûts rencontrés sur chaque arc de la solution s et $\delta(m)$ par la différence entre le coût aggrégé des deux arcs entrants et le coût aggrégé des deux arcs sortants. Pour le PSC muni d'un $flip$, si $\psi(s)$ indique le nombre de contraintes violées, on formulera $\delta(m)$ comme la différence entre le nombre de contraintes nouvellement violées et le nombre de contraintes nouvellement satisfaites par l'application du mouvement m. Sans perte de généralité, on se place maintenant dans la situation de minimiser la fonction $\psi(s)$.

1.2.1 Principes heuristiques de la recherche locale

En recherche locale, la conduite heuristique d'un algorithme suit deux principes essentiels : le principe d'amélioration ainsi que le principe d'intensification/diversification. Le principe d'amélioration envisage simplement de modifier la solution courante en choisissant à chaque itération un mouvement qu'on estime bénéfique, en général, parce qu'il améliore la valeur objectif. Le principe d'intensification/diversification vise à rationaliser la fouille de l'espace de recherche. Selon ce deuxième principe, certaines composantes de la recherche approfondissent la fouille de zones de l'espace ayant présenté des solutions intéressantes (phases d'intensification), alors que d'autres visent à atteindre des zones encore inexplorées (phases de diversification).

Principe d'amélioration. Les mécanismes d'amélioration, ou de descente lorsqu'on minimise, constituent l'ingrédient central d'une recherche locale. Sachant qu'on cherche à minimiser $\psi(s)$, un mécanisme de descente part d'une solution $s = s_0$ et tente d'améliorer $\psi(s)$ le long d'un chemin $(s_0, s_1, ...s_L)$. Considérons, dans un cadre formel, une fonction d'utilité $U_r : m \rightarrow U_r(m)$ définie pour chaque mouvement. Dans le contexte d'une recherche non markovienne, l'indice r indique que la trajectoire de recherche est succeptible de paramétrer la fonction d'utilité. Deux types de parcours

fondés sur ce mécanisme de descente sont particulièrement utilisés. Dans le cas d'un *parcours en meilleure amélioration*, chaque itération sélectionne le mouvement m vérifiant $U_r(m) = \max_{m' \in M_s} U_r(m')$. Dans le cas d'un *parcours en première amélioration*, chaque itération sélectionne le premier mouvement m visité vérifiant $U_r(m) > 0$. Un mécanisme de descente pure correspond au cas où $U_r(m) = -\delta(m)$, et dans ce cas, la descente échoue au premier optimum local rencontré de la fonction ψ. Lorsqu'on souhaite étendre la descente au-delà d'un optimum local, la fonction d'utilité adjoint à δ une composante auxiliaire dépendant de la trajectoire des solutions visitées et/ou d'un critère cherchant à éviter le blocage de la recherche.

Principe d'intensification/diversification. Bâtie autour du seul principe d'amélioration, une recherche locale constitue une trajectoire linéaire de l'espace de recherche. Si une telle trajectoire a l'avantage de se déployer très rapidement, elle offre en revanche peu de visibilité sur la topologie de l'espace de recherche, et ce faisant, peut être difficile à orienter. Heureusement, on observe souvent que les optima locaux de la fonction ψ sont regroupés par grappes dans l'espace de recherche (Hansen et Mladenovic, 2002). Lorsqu'un optimum est rencontré, il peut donc être judicieux de fouiller la zone courante de la recherche puisqu'elle présente probablement d'autres optima locaux, éventuellement meilleurs. Il est toutefois aussi important d'orienter la trajectoire des solutions vers des zones inexplorées de sorte à découvrir d'éventuelles nouvelles grappes. Pour cette raison, l'algorithmique d'une recherche locale présente idéalement l'équilibre de deux phases heuristiques. D'une part, les composantes d'intensification entraînent des phases de recherche approfondie autour de la zone en cours d'examen. D'autre part, les composantes de diversification encouragent à parcourir des zones encore inconnues de l'espace de recherche. Il est intéressant de faire le parallèle avec les notions d'exploration et d'exploitation mises en avant par l'étude des algorithmes de type évolutionniste. D'une certaine manière, la diversification constitue une phase d'exploration au cours de laquelle est acquise une connaissance :

la localisation éventuelle de nouvelles grappes. L'intensification constitue, elle, une phase d'exploitation au cours de laquelle est utilisée cette connaissance : on fouille les grappes d'optima découvertes. Au même titre que l'exploration et l'exploitation, la diversification et l'intensification constituent des composantes de recherche complémentaires.

1.2.2 Techniques heuristiques en recherche locale

Nous présentons maintenant trois types de techniques couramment utilisées pour programmer la conduite heuristique de la recherche. Premièrement, les techniques de perturbation consistent à modifier localement la topologie de l'espace de recherche en adjoignant à la fonction U_r une composante annexe indicative de l'intérêt associé à chaque mouvement. Deuxièmement, les techniques de relance consistent à réitérer un certain nombre de recherches locales sous des conditions initiales différentes afin de renforcer la couverture de l'espace de recherche. Enfin troisièmement, les techniques de parallélisation servent à diversifier ainsi qu'à émuler la recherche locale entre plusieurs parcours de recherche.

Techniques de perturbation. Les techniques de perturbation permettent de modifier localement et virtuellement la structure du graphe de recherche ainsi que la topologie induite par la fonction ψ. Formellement, on peut voir une perturbation comme l'ajout ou la modulation d'un terme dans la fonction d'utilité U_r. Le terme de perturbation peut être judicieusement aléatoire, ce qui revient à bruiter l'orientation du critère de sélection des mouvements. Le terme de perturbation peut aussi être une fonction déterministe de l'état de la recherche. Lorsque les termes de perturbation sont infinis, le filtrage est impartial. Dans ce cas, certains mouvements sont tout simplement ignorés ou d'autres automatiquement sélectionnés. Lorsque les termes de perturbation prennent la forme d'une pénalité finie, le filtrage est incitatif.

Dans ce cas, certains mouvements sont favorisés et d'autres sanctionnés selon qu'ils adhèrent ou s'écartent de la stratégie de recherche. Les techniques de perturbation permettent typiquement de catalyser la descente dans des situations particulières. Par exemple, elles permettent d'éviter un cyclage de la trajectoire des solutions visitées autour d'un optimum local courant lorsque des dégradations de la fonction ψ sont tolérées (Hansen, 1986). La pondération dynamique des pénalités associées aux contraintes violées utilisée en satisfaction de contraintes pour sortir de régions où ψ reste monotone constitue un autre exemple (Bohlin, 2002).

Techniques de relance. Les techniques de relance proposent de systématiquement réamorcer une descente lorsqu'un optimum local bloque la recherche. Par exemple, un schéma classique de relance consiste à générer aléatoirement un ensemble de solutions initiales à partir desquelles est ensuite exécutée une phase de descente. On observe cependant que cette procédure reste assez limitée (Lourenço, Martin et Stutzle, 2002). En effet, pour de grands problèmes, chaque descente reste très souvent bloquée à un certain pourcentage de l'optimum, ce pourcentage étant intrinsèque à la classe et à la taille du problème. Également, chaque descente est une réinitialisation du programme qui n'utilise pas d'information acquise par les recherches précédentes. Enfin, le choix aléatoire des solutions initiales ne permet pas d'assimiler la distribution en forme de grappes des régions contenant les optima locaux. Une amélioration aux techniques de relance aléatoires consiste généralement à réamorcer chaque descente à partir d'une solution structurellement proche d'un optimum local déjà obtenu (on reste sur une grappe d'optima) et cependant distante sur le graphe de recherche (on ne retombe pas sur l'optimum déjà rencontré). On peut dans ce cas conjuguer plusieurs idées : modifier l'optimum local courant au moyen d'une courte trajectoire aléatoire et/ou d'une structure de voisinage différente. La relance peut être ensuite effectuée sur la base du même schéma de mouvement ou encore d'un schéma de mouvement différent.

Techniques de parallélisation. De manière générale, un algorithme de recherche locale se parallélise naturellement autour d'un ensemble de phases de descente. Éga-

lement, les problèmes décomposables pour lesquels on applique une recherche locale à chaque sous-problème peuvent être très logiquement parallélisés. Les problèmes de routage pour lesquels un ensemble de véhicules doit couvrir à moindre coût tous les nœuds d'un graphe constituent un exemple : étant donné un partitionnement des nœuds, il se décompose en un ensemble de PVC (Rochat et Taillard, 1995, Laporte et Semet, 2001). Lorsque chaque fil de recherche possède juste son propre domaine d'information, l'algorithme obtenu correspond tout compte fait à une parallélisation matérielle des techniques de relance ou de décomposition. Étant donnée la structure du problème traité, les différents fils de recherche peuvent toutefois partager un domaine d'information global à partir duquel le programmeur cherchera à émuler et/ou à faire collaborer les différents fils de recherche. De tels algorithmes empruntent naturellement leurs mécanismes de sélection et de coordination aux algorithmes de recherche à base d'une population de solutions (Calegari, Coray, Hertz, Kobler et Kuonen, 1999). Une classification des techniques de parallélisation est proposée par Crainic et Toulouse (1997).

1.3 Quelques algorithmes représentatifs

Pour terminer cette introduction, nous donnons un aperçu de quelques algorithmes représentatifs. Le panorama qu'on propose permet d'illustrer les différentes perspectives de la recherche locale que nous avons jusque-là exposées. Nous commençons par la présentation d'algorithmes très simples (Algorithmes de première amélioration, de meilleure amélioration, de recuit-simulé) et terminons par la description d'algorithmes plus complexes (Algorithmes tabou, à voisinage variable, à voisinage à profondeur variable). Chaque algorithme utilise un graphe de recherche caractéristique de la recherche locale et, comme nous en discuterons, déploie à sa manière les principes et techniques heuristiques que nous avons décrits.

Algorithme de première amélioration. Une itération de l'algorithme de première amélioration explore progressivement le voisinage de la solution courante s. Dès qu'un mouvement m vérifie la condition $\delta(m) < 0$, l'énumération est terminée et on effectue le mouvement vers la solution $s \oplus m$. La recherche s'achève lorsqu'il n'est plus possible de trouver un tel mouvement. Il est clair que dans le cas du PVC et du PSC, l'algorithme converge en un nombre fini d'itérations. Ce sera toujours le cas dès lors que le graphe de recherche comporte un nombre fini de solutions. Le résultat de l'algorithme n'est la plupart du temps qu'un minimum local.

Algorithme de meilleure amélioration. Une itération de l'algorithme de meilleure amélioration énumère de manière exhaustive le voisinage de la solution courante s. Le mouvement m sélectionné vérifie la condition $\delta(m) = \min_{m' \in M_s} \delta(m') < 0$ et on effectue ce mouvement vers la solution $s \oplus m$. La recherche s'arrête dès qu'il n'est plus possible de trouver un tel mouvement. Les propriétés de convergence sont identiques à l'algorithme de première amélioration. L'algorithme de meilleure amélioration converge généralement pour un plus petit nombre d'itérations que l'algorithme de première approximation mais chaque itération exige un temps de calcul plus long. Bien que ce phénomène dépende pour une bonne part de l'ordre d'énumération, on remarque assez souvent que l'optimum local obtenu en meilleure amélioration est moins bon que celui obtenu en première amélioration. L'intuition selon laquelle une descente moins rapide converge finalement sur des solutions plus performantes est au centre de l'algorithme de recuit-simulé suivant.

Algorithme de recuit-simulé. On utilise dans le recuit-simulé (Kirkpatrick, Gelatt et Vecchi, 1983) un paramètre exogène, appelé température et noté T. Ce paramètre sert à contrôler la part aléatoire du processus de sélection d'un mouvement. À chaque itération, on énumère le voisinage courant jusqu'à ce qu'un mouvement vérifie la condition $\lambda < e^{-\frac{\delta(m)}{T}}$ où, pour chaque mouvement visité, $\lambda \in [0, 1]$ est tiré au hasard selon une loi uniforme. Un mouvement bénéfique est par ailleurs toujours

accepté. Au cours des itérations de l'algorithme, la température suit généralement une fonction partant d'une valeur initiale positive et décroissant géométriquement vers 0. Il existe bien sûr des schémas de refroidissement plus complexes mais l'essentiel reste que le critère de sélection est d'autant plus impartial que la recherche se rapproche des solutions quasi-optimales alors que des mouvements dégradants sont plus facilement acceptés dans la phase initiale de descente. Les critères d'arrêt du recuit-simulé se formulent naturellement par rapport au schéma de refroidissement, par exemple lorsque la température atteint une valeur seuil infinitésimale. Contrairement aux précédents algorithmes, la recherche d'un recuit-simulé ne bloque pas sur un minimum local. Tirer profit de cette propriété dépend malgré tout de la vitesse à laquelle refroidit la température de sorte qu'en pratique l'algorithme exige des temps de calcul conséquents. Le mécanisme de sélection des mouvements du recuit-simulé basé sur la température constitue un premier exemple de métaheuristique.

Algorithme tabou. La recherche taboue présente un vaste éventail de mécanismes métaheuristiques fondés sur une mémoire de la trajectoire de la recherche (Glover et Laguna, 1997). Nous présentons ici l'algorithme le plus fréquemment mis en œuvre initialement introduit par Glover (1986) et Hansen (1986). Dans cet algorithme, certains mouvements sont filtrés au moyen d'une mémoire associative liée aux derniers mouvements effectués. Dans ce contexte, il est nécessaire de mettre en place la notion d'attribut dont les valeurs servent à indexer la mémoire associative. Un attribut de mouvement représente une propriété particulière d'un mouvement. Par exemple, pour le PVC, il est courant de définir deux attributs, l'un concernant la création d'adjacence, et l'autre, la destruction d'adjacence. En effet, comme l'illustre la figure 1.1, un mouvement 2-*opt* introduit deux fois une adjacence entre deux couples de nœuds ainsi qu'il détruit deux adjacences entre deux couples de nœuds. Pour le mouvement 2-$opt(\{1,5\},\{2,3\})$, la valeur de création d'adjacence serait ainsi représentée par le couple $(\{1,2\},\{3,5\})$. Dans une version possible de l'algorithme tabou pour le PVC,

on retient en mémoire à chaque mouvement effectué et pour un nombre d'itérations T les valeurs de création d'adjacence. Étant donnée cette mémoire, l'algorithme tabou visite de manière exhaustive le voisinage de la solution courante s. Le mouvement m choisi est celui qui minimise le coût différentiel δ parmi les mouvements non tabous, à savoir, ceux dont aucune des valeurs de destruction d'adjacence n'intercepte l'ensemble des valeurs de création d'adjacence stockées en mémoire. Le mécanisme de filtrage interdit ainsi de sélectionner un mouvement qui conduirait possiblement à des solutions fraîchement visitées (depuis moins de T itérations). Contrairement à une meilleure amélioration, il n'y a pas de condition $\delta(m) < 0$, ce qui permet à l'algorithme de continuer sa recherche lorsqu'un minimum local est atteint. Il y aurait cependant autour d'un tel minimum local un risque certain que la recherche s'enferme dans un cycle, situation que permet justement de prévenir le mécanisme de filtrage. Les raffinements d'algorithmes tabous sont nombreux, le principe directeur étant de sonder la trajectoire de recherche afin de compiler une mémoire dynamique puis de contrôler le choix des mouvements par des filtres fonctionnant sur la base de cette mémoire. Les filtres peuvent être impératifs, comme c'est le cas dans la version que nous avons présentée mais ils peuvent aussi prendre la forme d'une pénalité finie aggrégée à la fonction δ. Par exemple, une pénalité proportionnelle à la fréquence des mouvements effectués permettrait de ne pas utiliser trop souvent les mêmes mouvements (on visiterait sinon les mêmes zones de l'espace de recherche). Cette pénalité constitue un exemple de stratégie de diversification réalisé au moyen de techniques de perturbation.

Algorithme de recherche à voisinage variable. L'algorithme de recherche à voisinage variable (Mladenovic et Hansen, 1997, Hansen et Mladenovic, 2003) fonctionne sur la base d'un ensemble M_k de structures de voisinage, souvent imbriqués, c'est-à-dire tel que $V_{M_k}(s) \subset V_{M_{k+1}}(s), \forall s \in S$. Pour une structure M_k donnée, une phase de recherche partant d'une solution s comporte un premier mouvement effectué

au hasard suivi d'une descente. Si l'optimum local s' obtenu au bout du compte est meilleur que s, le chemin de recherche est validé. Dans ce cas, on pose $k = 1$, $s = s'$ et on recommence la procédure à partir de la solution fraîchement obtenue. Dans le cas contraire, on recommence la phase de recherche avec la structure de voisinage M_{k+1} suivante. L'algorithme débute avec M_1 et termine lorsque le dernier voisinage ne permet pas d'améliorer la solution initiant la phase de recherche. Comme un optimum local relatif à M_k n'est pas forcément un optimum local relatif à M_{k+1}, la recherche ne bloque pas systématiquement au premier optimum local obtenu. Contrairement au recuit-simulé qui exige un lent refroidissement et donc s'accompagne d'un grand nombre de mouvements inutiles et contrairement à l'algorithme tabou qui exige de maintenir et de consulter une mémoire, le mécanisme qui permet d'échapper aux optima locaux se traduit ici par un moindre allourdissement de la recherche. Il est intéressant de remarquer que lorsqu'une recherche à voisinage variable trouve un optimum local relatif à une certaine structure de voisinage, il visite le voisinage de la solution courante de manière approfondie au moyen des structures de voisinage annexes. La recherche à voisinage variable présente à ce titre un exemple de stratégie d'intensification réalisé au moyen de techniques de relance. Pour finir, et légèrement en dehors du cadre $V_{M_k}(s) \subset V_{M_{k+1}}(s)$, signalons qu'un changement de voisinage orthogonal a pour effet de transposer la solution courante localement optimale au voisinage d'une autre grappe, structurellement différente, dans le nouvel espace de recherche. Cette propriété peut constituer une remarquable conjugaison d'intensification et de diversification.

Algorithme de recherche à voisinage à profondeur variable. Introduits par Kernighan et Lin (1970) pour des problèmes de partitionnement, puis Lin et Kernighan (1973) pour le PVC, les algorithmes de recherche à voisinage à profondeur variable proposent de visiter des voisinages de grande taille de manière heuristique. Ce type de technique a été par la suite étendu entre autres par Glover

(1992) sous le vocale des chaînes d'éjection. La version de l'algorithme pour le PVC que nous esquissons ici (Helsgaun, 2000) adopte à chaque itération un mouvement noté $(e_1^-, e_1^+, ..., e_r^-, e_r^+)$ où chaque e_i^- (resp. e_i^+) dénote un arc sortant (resp. rentrant) de la solution courante et r la profondeur variable du mouvement. Si on note $g_i = C(e_i^-) - C(e_i^+)$ et $G_i = g_1 + ... + g_i$, G_r dénote le gain du mouvement. Au lieu d'envisager l'ensemble des mouvements du voisinage pour une profondeur r fixée, on construit progressivement la suite $(e_1^-, e_1^+, ...)$ de sorte que $(e_1^-, e_1^+, ..., e_r^-, e_r^+)$ définisse un circuit (comme c'est le cas pour un 2-*opt*). Pour chaque $i \geq 2$, le choix d'un e_i^- doit permettre une éventuelle fermeture e_i^+ (définie implicitement) sur un mouvement préservant la structure de tour et le choix d'un e_i^+ doit conduire à $G_i > 0$. Enfin, l'ensemble des arcs sortants et entrants doivent rester disjoints. La recherche des suites $(e_1^-, e_1^+, ..., e_r^-, e_r^+)$ vérifiant ces conditions suit un parcours en profondeur d'abord. Dès qu'un e_i^- conduit à une fermeture vérifiant $G_i > 0$, le mouvement est effectué et la recherche du prochain mouvement est réinitialisée. Lorsqu'un parcours $(e_1^-, e_1^+, ...)$ se termine sans pouvoir satisfaire les conditions énoncées, seule l'exploration des nœuds initiaux du parcours (par exemple, le choix des quatres premiers : e_1^-, e_1^+, e_2^- et e_2^+) sont prolongés à nouveau pour de nouvelles alternatives. L'algorithme termine lorsqu'il n'est plus possible de trouver une suite $(e_1^-, e_1^+, ...)$ valide. La puissance d'un tel algorithme de recherche à voisinage à profondeur variable tient de la visite partielle mais efficace d'un voisinage de très grande taille. En effet, lorsqu'une somme partielle (ici G_r) est positive, il existe une permutation de ses termes de sorte que chaque sous-somme partielle soit positive : en se limitant à la recherche d'un mouvement correspondant à cette permutation remarquable, l'algorithme de Lin & Kernighan absorbe la combinatoire relative à la recherche exhaustive d'un mouvement vérifiant $G_r > 0$.

1.4 Portée et limites du paradigme

La recherche locale démontre une utilité certaine dans des situations très diverses. Elle permet avant tout d'apporter une solution approchée à des problèmes intraitables de manière exacte. Glover et Laguna (1997) présentent par exemple une liste significative de problèmes résolus de front au moyen de la méthode taboue. La recherche locale est aussi utilisée accessoirement, lorsqu'on souhaite par exemple calculer des bornes utiles en programmation en nombres entiers. Une autre propriété intéressante des algorithmes de recherche locale vient de ce que l'algorithmique est itérative ce qui implique qu'une interruption entre deux itérations quelconques érode rarement le produit de la recherche. Les algorithmes de recherche locale sont donc préemptifs et peuvent à ce titre être beaucoup plus facilement intégrés au sein de systèmes d'aide à la décision asynchrones (Zilberstein 1993, Strosnider et Paul, 1994, Garvey et Lesser, 1994) que ne pourraient l'être d'autres algorithmes de programmation mathématique (Séguin, Potvin, Gendreau, Crainic et Marcotte, 1997, Crouzet et Savard, 2002). Cette dernière propriété rencontre des besoins naissant dans le sillon des progrès technologiques en matière de systèmes d'information. Des problèmes de logistique rencontrés dans le domaine des transports ou encore de la santé et devant être résolus sous des contraintes molles de temps-réel ont donné lieu à quelques exemples significatifs (Psaraftis, 1995, Brotcorne, Farand, Laporte et Semet, 1999, Gendreau, Guertin, Potvin et Taillard, 1999).

Si la recherche locale apporte une contribution déterminante à la résolution de certaines problématiques, elle atteint cependant très vite ses limites en dehors des situations auxquelles elle est adaptée. Nous proposons à cet effet quatre axes de réflexion autour desquels le programmeur devrait être en mesure d'estimer le potentiel de la recherche locale pour des problèmes auquel il serait confronté. Au même titre, Hertz et Widmer (2002) ainsi que Hertz (2005) proposent une série de recommandations

générales sur l'utilisation de métaheuristiques tant pour les méthodes de recherche locale que pour les méthodes basées sur une population de solutions. Hertz, Taillard et De Werra (1995) illustrent ces aspects méthodologiques pour le cas de la méthode taboue. Également, Hansen et Mladenovic (2003) décrivent les étapes d'une documentation préalable à un recours avisé à la recherche locale.

Congruence de la recherche locale. La recherche locale, proposant généralement une méthode de résolution simplifiée et économe en calcul, est essentiellement utilisée pour générer des solutions approchées à des problèmes insolubles de manière exacte. Il arrive cependant que la recherche locale demeure un cadre de calcul trop lourd pour certains problèmes de très grande taille. Certaines instances de *PVC* comportant plusieurs millions de nœuds sont ainsi parfois traitées au moyen d'autres paradigmes de programmation comme les algorithmes d'approximation (Bartholdi et Platzman 1988, Reinelt 1992, Bentley 1992). À l'opposé, certains problèmes simples ne méritent pas d'être résolus par des méthodes de recherche locale. Il convient donc de considérer si la recherche locale constitue une échelle de calcul adéquate au regard du cadre d'utilisation des solutions. Cette analyse doit tenir compte, entre autres, de l'approximabilité des réponses, du temps de résolution disponible ainsi que de la taille et difficulté théorique du problème.

Navigabilité de la recherche locale. Il arrive qu'un cadre de résolution ait été préalablement mis en œuvre pour traiter de manière exacte des problèmes de taille modeste. L'approche consistant à dériver une résolution heuristique de ce premier cadre mérite souvent d'être étudiée. On dispose en effet d'une structure (par exemple un arbre de recherche en programmation en nombres entiers ou en programmation par contraintes) ou de constructions dérivées d'un critère d'optimalité (par exemple le coût réduit dans un simplexe) pouvant servir de support efficace à des méthodes heuristiques. Il n'est pas étonnant que ces heuristiques puissent être plus performantes qu'une recherche locale. Même si par exemple, le cadre d'une recherche arborescente

semble plus lourd qu'un parcours linéaire et markovien propre à la recherche locale, il permet de disposer en contre-partie d'une traçabilité qui rend la trajectoire de recherche plus repérable et donc plus orientable au moyen d'heuristiques. Certains travaux de recherche à l'intersection de la recherche locale et de la programmation par contraintes visent d'ailleurs à hybrider les deux types d'approches (Laburthe et Caseau, 1998).

Ergonomie de la recherche locale. La programmation d'une recherche locale exige la mise en place d'une structure de voisinage. Généralement, ce prérequis limite son utilisation à des problèmes discrétisables. Dans le cadre des problèmes d'optimisation à deux niveaux (Migdalas, Pardalos, Värbrand, 1998), on doit ainsi bâtir la recherche locale sur l'ensemble discret des équilibres canoniques autour desquels pivote le comportement des suiveurs plutôt qu'autour des variables de décision du meneur, plus naturelles a priori, mais à valeurs continues (Gendreau, Marcotte et Savard, 1996). Lorsqu'un problème offre la perspective de concevoir un voisinage, il reste que ce dernier doit présenter une structure permettant d'être exploré efficacement. Cette structure devrait permettre, d'une part, d'énumérer les coordonnées du voisinage de manière séquentielle, et d'autre part, d'estimer virtuellement le coût d'un mouvement, c'est-à-dire sans avoir à l'effectuer. Dans certains cas, il peut aussi s'avèrer décisif d'entretenir à chaque itération des structures d'informations à partir desquelles on peut accélérer les calculs nécessaires à l'itération suivante. Bien qu'en général ces structures ne doivent être modifiées que marginalement suite à l'application d'un mouvement, le gain obtenu est relatif au coût de maintenance et à la taille de la structure à entretenir. Le rôle déterminant de l'étape de modélisation est bien connu en programmation mathématique. En recherche locale, une modélisation efficace requiert notamment de trouver une représentation des solutions ainsi qu'une structure de voisinage offrant une ergonomie avantageuse.

Topologie de la recherche locale. Le choix d'une structure de voisinage et d'une fonction de coût pour les mouvements induit une topologie dans l'espace de recherche.

Il est essentiel d'éviter une topologie qui contiendrait de larges zones de l'espace de recherche ne présentant pratiquement aucune variation. Il serait très difficile de guider la recherche dans ces zones. Également, un espace de recherche en dent de scie rendrait erratique la recherche d'un optimum global. Comme dans le cas des critères portant sur l'ergonomie, le choix d'un voisinage et d'une fonction de coût au regard de la topologie induite est une phase cruciale de la modélisation en recherche locale. Un exemple éloquent concerne le problème de coloration de graphe (Galinier et Hertz, 2006). L'expérience montre en effet qu'une recherche locale cherchant à minimiser directement le nombre de couleurs utilisées dans une coloration en modifiant à chaque itération la couleur d'un sommet constitue une méthode de résolution laborieuse. En revanche, l'approche visant à chercher une coloration réalisable pour un nombre de couleur fixe en minimisant le nombre de couples de sommets en conflit s'avère plus efficace. Dans ce cas, un mouvement modifie la couleur d'une extrémité d'un sommet en conflit. Chaque fois qu'une coloration s réalisable est trouvée, on procède à une nouvelle recherche locale en décrémentant le nombre de couleurs disponibles et en partant, par exemple, de la solution s précédemment trouvée dont les sommets d'une certaine couleur sont réaffectés.

1.5 Conclusion

Dans ce chapitre d'introduction, nous avons exposé, d'abord, les principes de la recherche locale, et ensuite, un ensemble d'algorithmes. Ce plan de présentation visait à se démarquer du cataloguage en une série de métaheuristiques qui est souvent introduit lorsqu'est abordé le paradigme. Le plan que nous avons suivi s'accorde par ailleurs avec les lignes du modèle informatique que nous proposerons comme fondement des librairies génériques pour la recherche locale : un algorithme est constitué d'un graphe de recherche (formé de solutions et de mouvements), d'un certain

nombre d'observations (par exemple, le coût et les attributs de mouvements) et de mécanismes filtrant la trajectoire de recherche en fonction de ces observations (par exemple, une descente en première amélioration).

Nous avons donc d'abord défini la recherche locale comme le parcours d'un ensemble de solutions dans un graphe de recherche au moyen de mouvements. Nous avons délibérément introduit le concept de fonction de coût associée aux mouvements après cette définition afin de préparer l'idée selon laquelle ce coût fait plutôt partie des observations à calculer en cours de recherche. Au plan conceptuel de la recherche locale, un mouvement est un opérateur alors qu'un coût, de même qu'un attribut de mouvement, est une observation.

Nous avons ensuite discuté des modes heuristiques en recherche locale. Les observations faites en cours de recherche constituent les données des mécanismes métaheuristiques. Nous avons séparé les principes heuristiques fondamentaux (principe de descente et principe d'intensification/diversification) des techniques qui permettent de les réaliser (techniques de perturbation, de relance et de parallélisation). Nous avons finalement illustré notre argumentaire par la présentation de quelques algorithmes de recherche locale bien connus et représentatifs de la richesse de ce paradigme.

Nous avons enfin abordé la recherche locale sous un angle plus critique. Bien que la puissance de calcul et l'aspect interruptif de la recherche locale constituent de précieux atouts, nous avons pris le soin d'en encadrer la pertinence en fonction de critères généraux comme la congruence, la navigabilité, l'ergonomie et la topologie. La compréhension de ces quelques critères permet d'aborder plus solidement la modélisation d'un problème de combinatoire par la recherche locale.

CHAPITRE 2 : LANGAGES DE RECHERCHE LOCALE

La plupart des paradigmes de programmation mathématique dispose d'utilitaires permettant de vulgariser la création d'applications. En programmation linéaire ainsi qu'en programmation en nombres entiers, des logiciels comme CPLEX ou ABBACUS (Junger et Thienel, 1998) sont couramment utilisés. Une librairie comme LEDA (Mehlhorn et Naher, 1995) propose un éventail de structures de données et d'algorithmes dans le domaine des graphes et réseaux. En programmation par contraintes, il existe un très grand nombre de langages de programmation, parmi lesquels on note ILOG Solver et CHIP (Dincbas, Van Hentenryck, Simonis, Aggoun et Graf, 1998). Bien que l'utilisation d'un logiciel contribuerait certainement à simplifier le développement d'applications, la recherche locale accuse un curieux retard à ce jour. Les programmeurs ne disposant en effet encore que d'utilitaires en phase prototype, la très grande majorité des applications est directement programmée sans assistance logicielle. Cette situation s'explique certainement par la difficulté de réaliser un langage de recherche locale, et cela pour deux raisons qui seront discutées : les structures de données servant à représenter les problèmes sont fortement typées et les schémas d'algorithmes sont volatiles. Nous verrons malgré tout que certaines solutions logicielles proposées dans la littérature ont permis de contourner ces obstacles. Quel que soit l'espace des possibilités techniques qui s'offre à la réalisation d'un atelier de programmation, il ne faut toutefois pas oublier qu'un langage ne se révèle utile que s'il répond aux attentes des programmeurs. Pour le cas de la recherche locale, les contributions potentielles d'un langage dépendent en grande partie du niveau de programmation et d'expertise que requiert la conduite d'un projet particulier. Nous proposons donc pour commencer d'examiner les différents styles de programmes habituellement rencontrés dans le domaine.

2.1 Les différents styles de programmation

On distinguera naturellement trois catégories de programmes de recherche locale : les recherches locales pures, les recherches locales avec métaheuristiques et enfin les recherches locales composites. Comme nous allons voir, cette classification s'inscrit manifestement dans une perspective historique qui retrace les différentes générations d'algorithmes. Il n'empêche que selon le type de liens qui rattachent un projet et ses intervenants aux méthodes de la recherche locale, chacune des trois approches répond encore aujourd'hui à nombre d'applications. Ceci est d'autant plus vrai que le choix des structures de solution et de mouvement importe bien davantage que le degré de métaheuristique qui équipe un algorithme.

Recherches locales pures. En recherche locale pure, l'algorithmique repose seulement sur un mécanisme d'amélioration pure. Sans critère d'évasion, la recherche s'arrête donc sur le premier optimum local rencontré. Il faut remarquer que la simplicité du cadre de la recherche permet de se concentrer sur des structures de voisinage plus complexes. Certains des algorithmes appartenant à cette classe peuvent être ainsi aussi raffinés et efficaces que d'autres avec métaheuristiques ou qui hybrident des méthodes à base de population de solutions. C'est par exemple le cas des algorithmes utilisant des voisinages à profondeur variable présentés au chapitre 1. Les travaux de Lin et Kernighan (1970, 1973), pour lesquels le concept de métaheuristique n'existait tout simplement pas à l'époque, fournissent ainsi des solutions qui résistent encore à des méthodes de recherche locale plus élaborées. Mise à part la structure de voisinage, les algorithmes de recherche locale pure sont très simples à programmer. Ils sont généralement plus rapides qu'une recherche locale élaborée mais convergent par défaut vers des solutions plus pauvres. Hormis le fait qu'il puisse simplifier le codage des structures relatives au voisinage, l'utilité d'un langage est dans le cas général assez restreinte pour ce type de programmation. Il permettrait malgré

tout d'organiser de manière systématique les différentes composantes du programme de recherche locale.

Recherches locales avec métaheuristiques. Dans le cadre d'une recherche locale avec métaheuristiques, le programmeur se réfère typiquement à la version standard d'un algorithme doté d'un critère d'évasion comme le recuit-simulé ou la méthode taboue présentés au chapitre 1. Ces algorithmes restent relativement simples à programmer tout en permettant d'éviter l'écueil sur un optimum local. Par contre, la résolution s'effectue parfois au prix d'un long temps de calcul. Mis à part le choix déterminant d'une structure de mouvement, la principale difficulté de la programmation tient ici du réglage des paramètres de la métaheuristique, comme par exemple, la longueur de la liste taboue ou le schéma de refroidissement d'un recuit-simulé. C'est dans ce cadre de programmation que la plupart des problèmes réels sont abordés. L'utilité d'un langage de recherche locale est ici plus sensible. Il serait par exemple fort appréciable qu'un langage permette au programmeur de n'avoir seulement à construire que ses structures de voisinage et de solution, lesquelles seraient ensuite passées en paramètre à des algorithmes avec métaheuristiques préfabriqués pour générer la recherche désirée.

Recherches locales composites. Les recherches locales composites conjuguent les ingrédients les plus élaborés du domaine. On s'attache dans ce cadre à combler les insuffisances d'une recherche linéaire et markovienne en transposant directement les principes d'intensification et de diversification à l'examen des structures mathématiques caractéristiques du problème à résoudre. Souvent, deux nouveaux aspects de programmation s'introduisent. D'une part, on peut être amené à monter des mécanismes capables de mémoriser un ensemble d'informations rencontrées au cours de la recherche. Ce premier point permet de pallier l'évanescence d'une recherche markovienne. D'autre part, on peut être tenté de développer l'algorithme autour d'un ensemble de fils de recherche (Crainic, Toulouse et Gendreau, 1997) qui éventuellement

interfèrent. Ce deuxième point permet de consolider l'aspect linéaire de la recherche locale. Dans ce contexte, la recherche locale composite tend à rejoindre le champ des méthodes de type évolutionniste comme les algorithmes génétiques (Holand 1975, Goldberg 1989, Calegari, Coray, Hertz, Kobler et Kuonen, 1999) où la notion de mouvement est finalement perçue comme un opérateur de mutation. Soulignons que les algorithmes composites s'avèrent parfois si complexes qu'il devient délicat de les énoncer rigoureusement. Cette situation est malheureusement propice à bien des égarements. Quelques publications ont proposé d'édifier ce champ de recherche en termes de programmation à mémoire adaptive (Taillard, Gambardella, Gendreau et Potvin, 1998), de méthodes hybrides (Talbi, 2000) ou encore de systèmes multi-agents (Roli et Milano, 2001). En recherche locale composite, les algorithmes sont plus efficaces, en rapidité comme en précision. Un langage serait dans ce contexte d'autant plus utile que le développement des programmes est une tâche complexe. Réaliser un environnement de programmation s'avère cependant ici très ambitieux. Le langage se doit par exemple de formaliser les différents types de mémoire utilisées ainsi que les différentes opérations élémentaires fondées sur celles-ci. Il devrait enfin également permettre d'articuler la recherche parallèle de solutions ainsi que de gérer l'ensemble des solutions selon les mécanismes des méthodes évolutionnistes.

Tableau 2.1 – Classification des styles de programmation en recherche locale

Recherche locale	Exemples	Simplicité	Rapidité	Précision
Pure	Meilleure amélioration première amélioration	+	+	−
Avec métaheuristiques	Méthode taboue recuit simulé	+	−	+
Composite	Programmation à mémoire adaptive	−	+	+

Dans l'hypothèse où une même structure de voisinage serait utilisée, le tableau 2.1 résume les propriétés des programmes obtenus dans le cas de chaque type de programmation. Les trois générations d'algorithmes se succèdent de haut en bas. D'abord, la recherche locale pure se caractérise par l'élaboration de solutions pauvrement optimales. La recherche locale avec métaheuristiques apporte ensuite davantage de précision aux solutions générées au détriment du temps de calcul. Cette perte de rapidité, qui a été finalement compensée par la recherche locale composite, conduit à son tour à des algorithmes difficiles à programmer. Un langage de recherche locale, à condition qu'il permette de couvrir la recherche composite, contribuerait logiquement à réunir les trois critères de simplicité, de rapidité et de précision.

2.2 Réalisabilité d'un atelier de programmation

Bien que le bref historique précédent suggère dans le principe l'éboration de langages de recherche locale, ces derniers restent à ce jour impopulaires. Deux raisons peuvent certainement être invoquées à ce sujet : les structures de représentation de problème sont fortement typées et les schémas d'algorithmes sont volatiles. La première raison découle de ce que la programmation des solutions et des mouvements qui composent la recherche locale est fortement dépendante des problèmes à traiter. La seconde raison provient de ce que la recherche locale est une discipline en pleine évolution, une conjoncture qui freine les activités de standardisation, et donc à terme, d'application.

Typage des structures de données. Prenons le cas de la programmation linéaire ou de la programmation en nombres entiers. Formuler un problème dans ce cadre revient à exprimer un modèle au moyen d'un ensemble de variables entières ou réelles, d'une liste de contraintes liant ces variables par des opérateurs élémentaires algébriques (additions, multiplications) et logiques (comparaisons) et enfin d'une fonction objectif. Tout problème s'exprime donc dans un cadre analytique présentant des

types de données et une syntaxe universelle, par exemple, sous la forme d'une inégalité matricielle. Il est alors concevable de factoriser la représentation de même que la résolution de tout problème autour d'un solveur canonique. En réduisant leurs champs d'application à un ensemble précis de domaines de variables et de contraintes, les logiciels de programmation par contraintes se ramènent également à des situations où les termes en fonction desquels on peut formuler un problème sont connus à l'avance. En recherche locale, la représentation d'un problème correspond à la formulation des solutions et mouvements. Ces deux structures et les opérations qui leur sont appliquées ne se prêtent malheureusement pas à être factorisées. Leur définition, qu'on peut seulement énoncer à un niveau très abstrait, prennent des formes qui dépendent irréductiblement des problèmes à traiter. Il est par la suite inconcevable de couvrir les différentes représentations de problème autour d'un même format canonique.

Volatilité des algorithmes. Étant donnée la nature heuristique des algorithmes de recherche locale, le programmeur cherche consciencieusement à tirer profit de la structure du problème auquel il est confronté. Cette attitude se traduit par une adaptation du schéma de la recherche locale qui s'éloigne critiquement des algorithmes standards publiés dans la littérature. Alors que la plupart des algorithmes de recherche locale s'énoncent de manière très générale, leurs implantations sont donc fortement corrélées aux spécificités de chaque type de problème traité. À ce titre, les possibilités d'algorithmes que propose un langage de recherche locale constituent des versions souvent insuffisantes au regard de la version adaptée à laquelle songe le programmeur. Notons aussi que la recherche locale est une discipline en pleine expansion qui propose fréquemment de nouveaux mécanismes de recherche difficiles à anticiper. Par exemple, les langages supportent difficilement les aspects novateurs des recherches locales composites comme les mécanismes de parallélisation et de mémorisation de la recherche.

Malgré ces deux obstacles, la littérature dispose d'un certain nombre de pistes que

nous proposons de séparer en trois catégories : les *bibliothèques*, les *cadres d'application* et enfin les *langages déclaratifs*.

Tableau 2.2 – Contexte d'utilisation des différents types d'utilitaires d'environnements de programmation

bibliothèque	cadre d'application	langage déclaratif (bas-niveau)
Préprogrammation de la représentation du problème		
entièrement prise en charge	spécification de l'interface des mouvements et des solutions	mécanismes de maintenance pour des catégories de termes intervenant dans le choix des mouvements
Personnalisation de la représentation du problème		
choix et calibrage des composantes utilisées	programmation impérative des fonctions associées aux mouvements et solutions	formation des critères conduisant au choix d'un mouvement par la composition de termes pertinents
Préprogrammation de la structure algorithmique		
entièrement prise en charge	factorisation des algorithmes selon un patron modulable	formalisation de la structure algorithmique
Personnalisation de la structure algorithmique		
choix et calibrage des composantes utilisées	choix, calibrage et hybridation des algorithmes utilisés	modulation et calibrage des points de contrôle de l'algorithme

Le tableau 2.2 résume les caractéristiques de chacune de ces trois catégories d'utilitaires. Dans chaque colonne, on précise les charges qui reviennent à l'usager ainsi qu'à l'utilitaire tant au niveau de la programmation du graphe de recherche que du parcours dans ce graphe. À gauche, les bibliothèques proviennent du domaine de l'optimisation. Elles prennent la forme de librairies utilisables pour un ensemble de classes de problèmes similaires. Elles rassemblent des voisinages et des algorithmes

clefs-en-main pour la famille de problèmes prise en compte. Au centre, les cadres d'application sont également issus du domaine de l'optimisation. Ils mettent en oeuvre une interface entre, d'un côté, un schéma générique de recherche locale, et de l'autre, n'importe quelles structures de solution et de voisinage. L'utilitaire dispose d'algorithmes raisonnablement simples (méthode taboue, recuit-simulé) vérifiant le schéma générique. L'usager code lui-même ses solutions et voisinages dans le cadre réglementé de l'utilitaire. Ses structures peuvent dès lors être greffées à un patron particulier d'algorithme afin de finaliser la recherche locale désirée. À droite, enfin, les langages déclaratifs proviennent de travaux propres à la satisfaction de contraintes. Ces langages factorisent les similitudes algorithmiques qu'emploie la recherche locale dans ce domaine. Ici, l'utilisateur est tenu de programmer son voisinage comme le choix d'une variable à réaffecter (approche de bas-niveau). Dans ce cas, le langage prend en charge l'implantation et la maintenance des éléments qui participent au choix de la réaffectation tel que déclaré par l'utilisateur. Il est également possible de proposer à l'utilisateur une famille de contraintes à partir desquelles ce dernier compose directement son modèle et pour lesquelles le langage lui-même définit le voisinage correspondant et assure les opérations sous-jacentes nécessaires à la conduite de la recherche locale (approche de haut-niveau). Dans ce dernier cas, les voisinages utilisés ne sont pas programmables puisqu'ils sont imposés par le logiciel. Les trois sections suivantes décrivent plus en détails chacune de ces catégories de logiciel.

2.3 Les bibliothèques

Les bibliothèques proposent de factoriser la programmation d'algorithmes pour des problèmes présentant des similarités structurelles. Lorsque deux classes différentes de problèmes ont en commun une représentation analogue des solutions, la recherche locale peut en effet parfois utiliser des voisinages similaires et il devient possible d'écrire

des composantes de recherche pour les deux classes de problèmes. De manière générale, cette idée peut être étendue à une collection offrant un ensemble cohérent de structures de données pour les solutions, et dans chaque cas, leurs voisinages. Par exemple, une collection d'algorithmes et de voisinages valides pour des problèmes dont les solutions se codent comme des permutations d'entiers pourrait servir à résoudre, entre autres, le cas particulier des instances de PVC, d'autres problèmes de routage, ou encore des instances de problèmes d'affectation quadratique (Burkard, Karisch et Rendl, 1997). Dans le contexte des problèmes d'ordonnancement, un même type d'approche de recherche locale peut également être factorisé autour de graphes de préséance (Van Hentenryck et Michel, 2004). Une bibliothèque centrée autour des problèmes d'affectation a été par exemple proposée par Ferland, Hertz et Lavoie (1996).

Les bibliothèques permettent à un usager d'obtenir des algorithmes assez fidèles à ceux qu'il aurait programmés puisque les outils qu'elle fournit peuvent tenir compte de la spécificité du domaine. Par contre, de tels utilitaires souffrent d'un manque de flexibilité puisque leur champ d'application est restreint par construction. Il serait par exemple regrettable qu'une bibliothèque mette à disposition des outils permettant de résoudre une classe de problèmes mais que pour une certaine extention ou un raffinement de ces problèmes, elle ne puisse tout à coup plus être mise à contribution.

2.4 Les cadres d'application

Les cadres d'application ne font pas d'hypothèses sur les structures de solution et de mouvement qui servent à représenter les problèmes à résoudre. Ces composantes, dont l'utilitaire règlemente seulement l'interface, sont en effet codés par l'usager. Elles sont alors passés en paramètres à des patrons d'algorithmes préfabriqués dans l'utilitaire pour générer l'application désirée.

Ces utilitaires sont généralement programmés en C++ (Stroustrup, 1997). Le mécanisme d'interfaçage repose sur un polymorphisme statique et/ou dynamique. Dans le premier cas, l'usager définit une classe qui doit être valide pour les classes de l'utilitaire auxquelles elle est passée en paramètre gabarit (Austern, 1999). Dans le deuxième cas, l'usager dérive une classe abstraite dont il fournit une définition aux méthodes virtuelles (Gamma, Helm, Johnson et Vlissides, 1994). Ces utilitaires prennent la forme de cadres d'application (framework en anglais) : le cadre d'application factorise l'ossature des applications propres à la recherche locale au moyen d'une hiérarchie de classes abstraites et/ou gabarit. Les composantes étendues par l'usager se greffent finalement aux composantes de l'ossature pour générer l'algorithme de résolution.

Le diagramme UML que représente la figure 2.1 schématise l'organisation typique des classes d'un cadre d'application pour la recherche locale. Dans cette organisation, le programmeur définit les structures `Solution` et `Mouvement`, qui renferment respectivement les coordonnées d'une solution et d'un mouvement. Ces deux classes sont ensuite passées en paramètres gabarits (cf. Annexe A.1). Dans une classe qu'il dérive de `Configuration`, qu'on appelle ici par convention `ConfigurationUsager`, l'usager apporte une définition aux fonctionnalités associées à une `Solution`. Il doit ainsi programmer la fonction `valeur` qui initialise la valeur d'une solution. Dans une classe dérivée de `Exploration`, a priori notée `ExplorationUsager`, l'usager apporte une définition aux fonctionnalités associées à un `Mouvement`. Il doit ici coder les fonctions `faireMouvement`, `deltaValeur`, `prochainMouvement` et `unMouvement`, sur la base desquelles sont ensuite élaborées par défaut des fonctionalités telles que `meilleurMouvement`, `premierMouvement`. La fonction `recherche` de la classe `RechercheLocale` déploie le schéma générique de l'algorithme selon une technique qu'en orienté-objet on nomme patron de conception stratégie (Géron et Twabi, 1999). Les étapes `choisitMouvement` ainsi que `testeArret`, qui sont déclenchées dans

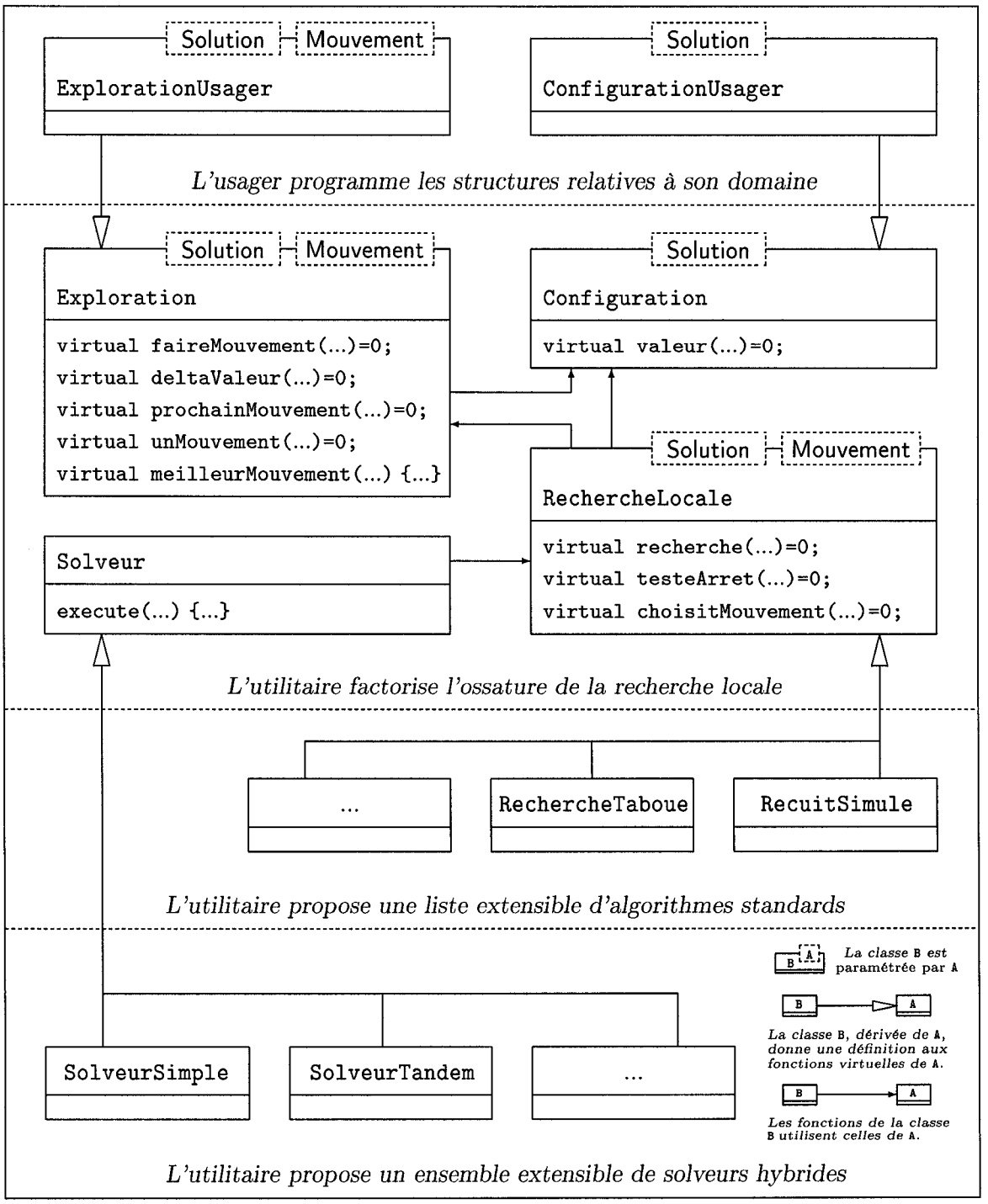


Figure 2.1 – Diagramme simplifié des cadres d'application

`recherche`, sont ainsi spécialisées dans des classes dérivées comme `RechercheTaboue`, `RecuitSimule`, ou tout autre classe relative à un algorithme que propose l'utilitaire. Un solveur abstrait, noté `Solveur`, vient enfin coordonner l'utilisation des algorithmes de recherche. Chaque spécialisation de `Solveur` en une classe dérivée permet de mettre en œuvre un type d'hybridation séquentiel des algorithmes de recherche disponibles, par exemple `SolveurSimple` et `SolveurTandem`.

SEARCHER (Andreatta, Carvalho et Ribeiro, 1998). Cet utilitaire met l'emphase sur les possibilités d'hybridation séquentielles entre les différentes formes de recherche locale qu'il propose. Le logiciel réserve la possibilité de changer de voisinage en cours d'exécution. Il supporte donc dans une certaine mesure la recherche à voisinage variable. L'utilitaire supporte également l'hybridation des méthodes constructives qui sont utilisées afin d'initialiser les solutions à partir desquelles est effectuée la recherche. SEARCHER a été testé pour le problème de la construction d'arbres phylogénétiques (Savaré, 1995).

EASYLOCAL++ (Di Gaspero et Schaerf, 2001a, 2001b). Cet utilitaire a été testé entre autres sur les problèmes de coloriage de graphe et sur des problèmes de construction d'horaires de cours. La figure 2.2 présente des extraits de code pour le premier cas. La classe générique `Solution` qui prend ici la forme de `Coloration` contient un tableau `couleur` dans lequel on fait correspondre une couleur à chaque sommet. On convient également que `Coloration` maintient la liste des sommets en conflit, notée `en_conflit`, ainsi que le nombre `n_couleurs` de couleurs utilisées. La classe `Mouvement`, qu'on note ici `Recoloration`, contient les coordonnées (v, i, j) d'un changement de couleur $i \rightarrow j$ pour un sommet v. La classe `ExplorationUsager`, qu'on note `ExplorationRecoloration`, implante les services d'une `Exploration`. La méthode `prochainMouvement` est illustrée en détail.

En plus de la fonction objectif définie sur les solutions et les mouvements, EASYLOCAL++ laisse la possibilité de définir une fonction pour mesurer et tracer la violation

```
struct Coloration
{
  int n_couleurs, *couleur;
  list<int> en_conflit;
};

struct Recoloration
{
  int v, i, j;
};

struct ExplorationRecoloration : Exploration<Coloration, Recoloration>
{
  void prochainMouvement(const Coloration& col, Recoloration& rc)
  {
    rc.j = (rc.j + 1) % col->n_couleurs;
    if (rc.j == rc.i) rc.j = (rc.j + 1) % col->n_couleurs;
  }
  ...
};
```

Figure 2.2 – Représentation d'une coloration de graphe avec EASYLOCAL++

des contraintes. La fonction coût synthétique est une agrégation de la fonction objectif et de cette mesure de violation. EASYLOCAL++ vise par ailleurs à réduire l'activité de développement. D'une part, la plupart des fonctionalités qui peuvent avoir un comportement par défaut possède une implantation dans l'utilitaire, qu'on peut toutefois adapter. D'autre part, chaque classe que l'usager est amené à spécialiser dispose d'un modèle d'en-tête qu'il suffit de compléter en codant le corps des fonctions. Des tests numériques ont montré qu'utiliser EASYLOCAL++ occasionne une surcharge du temps d'exécution de 30%.

HOTFRAME (Fink, Voss et Woodruff, 1998a, 1998b). Cet utilitaire fait un usage quasi-exclusif du polymorphisme statique. Les relations de dérivation du diagramme de la figure 2.1 doivent donc être remplacées par des types gabarits passés en paramètres, comme le sont d'ailleurs déjà les classes `Solution` et `Mouvement`. Comme l'illustre la figure 2.3, la fonction `recherche` prend ainsi la forme d'une fonction générique dont le type de sélection de mouvement est passé par le paramètre gabarit `Selection` plutôt que d'être spécifiée en interne en réalisant `choisitMouvement` dans une classe dérivée.

```
Solution recherche<Solution, Mouvement, Selection>(Solution s_courant)
{
  Solution s_meilleur = s_courant;
  do {
    if (s_courant.valeur < s_meilleur.valeur) s_meilleur = s_courant;
  }
  while ( Selection<Solution , Mouvement>(s_courant) );
  return s_meilleur;
}
```

Figure 2.3 – Composition de la recherche locale avec HOTFRAME

Cette approche présente deux avantages. D'une part, HOTFRAME est théoriquement plus rapide puisqu'il ne repose pas sur l'appel de fonctions virtuelles, celles-ci induisant inévitablement une surcharge de temps de calcul en raison du déréférencement en temps d'exécution des méthodes virtuelles et de l'impossibilité de les mettre en ligne à la compilation (Meyers, 1995). D'autre part, la plupart des erreurs de programmation sont détectées à la compilation. La syntaxe peut devenir par contre assez lourde. Ainsi, si `PlusFortePente` désigne une recherche où le meilleur mouvement est retenu à chaque itération, pour un problème de coloration, l'algorithme s'écrit : `recherche<Coloration,Recoloration,PlusFortePente<Coloration,Recoloration>>`. On remarque que les types `Coloration` et `Recoloration` apparaissent doublement dans la syntaxe. Dans la construction d'une méthode taboue, la syntaxe devient encore plus redondante. Le logiciel a été testé sur une série de problèmes dont le problème SAT, le problème de sac-à-dos multidimensionnel et des problèmes de séquencement. Même si conceptuellement, le logiciel se limite à des schémas métaheuristiques très canoniques, il est à noter que HOTFRAME propose une grande variété de schémas de refroidissement pour le recuit-simulé et de gestion de la liste des attributs pour la méthode taboue. L'utilitaire supporte par exemple des schémas de refroidissement multiphases ou encore des mécanismes gérant dynamiquement la taille de la liste taboue (Woodruff, Zemel, 1993 ; Battiti, Tecchiolli, 1994).

En conclusion, l'approche des cadres d'application présente l'avantage de contourner élégamment le problème de typage des structures de données en passant ces

structures en paramètres gabarits. Si par ce mécanisme, les cadres d'applications permettent de supporter la plupart des problèmes, l'interfaçage entre composants relatifs au domaine et composants relatifs aux algorithmes ne permet malheureusement pas de raffiner de manière simple et modulaire les algorithmes standards en fonction des spécificités des problèmes que rencontre l'usager. En fait, dès que l'algorithmique s'éloigne des schémas métaheuristiques traditionnels, l'usager se retrouve involontairement à étendre le cadre d'application comme s'il en était le concepteur. Les cadres d'application, en fournissant une palette de patrons d'algorithmes standards, se destinent avant tout à un style de recherche locale doté de métaheuristiques conventionnelles.

2.5 Les langages déclaratifs

Les langages déclaratifs offrent une approche alternative de résolution à la recherche arborescente mise en oeuvre par les solveurs complets de programmation par contraintes. Ces langages, qui héritent de perspectives informatiques propres à la satisfaction de contraintes, se distinguent nettement de ceux que nous venons de présenter en optimisation tant au niveau de l'architecture logicielle que du cadre de pensée dans lequel on s'emploie à modéliser les problèmes. Comme il en est des solveurs complets, ces langages s'appuient en général sur deux niveaux de programmation. Le premier niveau de programmation est déclaratif, il permet de modéliser les composantes d'une recherche locale qui dépendent du problème traité. Le deuxième niveau de programmation est impératif, il sert à diriger la recherche de solutions.

Il est remarquable qu'en satisfaction de contraintes la résolution de problèmes très différents peut toujours s'appuyer sur une fonction objectif et des mouvements essentiellement similaires. D'une part, la fonction objectif à minimiser mesure le degré

de non-réalisabilité de la solution courante. D'autre part, un mouvement consiste à choisir une variable en conflit (elle apparaît dans une contrainte non satisfaite) et à modifier sa valeur de sorte à rapprocher la solution de la réalisabilité au sens de la fonction objectif. Les spécificités d'une recherche locale appliquée à un problème particulier se ramènent donc ici principalement à une modulation du voisinage et du contrôle de la recherche. La modulation du contrôle est généralement indépendante du type de problème, elle concerne le type de descente utilisé et la présence éventuelle de métaheuristiques. La modulation du voisinage concerne la manière de choisir une variable en conflit ainsi qu'une nouvelle valeur pour cette variable au regard de la fonction mesurant l'écart à la réalisabilité. Bien que dépendant a priori du problème, la modulation du voisinage repose en fait sur des grandeurs caractéristiques du système de contraintes. Par exemple, si on est intéressé à choisir la variable qui entre en conflit dans un maximum de contraintes (resp. à minimiser le nombre de contraintes violées, resp. à minimiser le nombre de variables en conflit, etc), le solveur doit maintenir le nombre de contraintes violées par chaque variable (resp. le nombre de contraintes violées, resp. le nombre de variables en conflit, etc). De la même manière, on remarque que les mécanismes de la méthode taboue peuvent être réalisés indépendamment du problème à traiter puisque la liste des attributs peut toujours être formée par l'ensemble des indices correspondant aux variables fraîchement réaffectées. Les langages déclaratifs proposent, pour les types de contraintes qu'ils supportent, de maintenir automatiquement ces grandeurs caractéristiques à partir desquelles le programmeur exprime le choix d'un mouvement dans le voisinage.

LOCALIZER (Michel et Van Hentenryck 1998, 1999), LOCALIZER++ (Michel et Van Hentenryck, 2001) et COMET (Michel et Van Hentenryck, 2005). La syntaxe déclarative de LOCALIZER permet au programmeur de spécifier quelles grandeurs caractéristiques de la recherche, qualifiées d'invariants dans le langage, doivent être maintenues. Le programmeur est ensuite à même de préciser l'exploration de son

voisinage et l'évaluation des mouvements en fonction des informations qu'il compose à partir d'invariants. Un invariant peut être défini comme une expression pouvant contenir des données statiques du problème, des variables de décision et d'autres invariants. LOCALIZER assure une mise à jour incrémentale des invariants. Une syntaxe permet également d'exprimer le contrôle de la recherche locale comme les différents types de descente et critères d'acceptation. Le solveur de LOCALIZER renferme pour chaque catégorie d'invariants les structures devant être maintenues cohérentes. LOCALIZER++ suit les mêmes principes que LOCALIZER mais prend la forme d'une librairie C++ extensible. L'ajout de nouvelles catégories d'invariants ainsi que l'intégration du solveur dans un système d'information est ainsi facilitée. Le dernier outil de cette famille, COMET, propose enfin une couche d'abstraction supplémentaire en termes d'objets différentiels. Ces objets se caractérisent par un ensemble de fonctionnalités permettant de prédire la variation de l'objet suite à l'application d'un mouvement. Ces objets, qui peuvent être bâtis en fonction d'invariants, permettent de modéliser très naturellement des constructions intervenant en recherche locale comme des fonctions ou des contraintes. COMET se distingue également par la possibilité de fusionner des voisinages hétérogènes ainsi que par la possibilité d'introduire des points de contrôle à l'intérieur de l'énumération d'un voisinage. La figure 2.4 présente à titre d'illustration l'extrait d'un programme écrit avec LOCALIZER pour la résolution du problème SAT. La lisibilité et la concision du code constitue ici un atout remarquable des langages déclaratifs. Par contre, la surcharge de calcul est plus importante que dans le cadre des bibliothèques centrées autour d'un domaine ou des cadres d'application. Les auteurs font état d'une surcharge de calcul de 238% pour la résolution du problème SAT et de 482% pour la coloration de graphe. Pour certains types de compositions d'invariants, une propagation de la maintenance par couches est par ailleurs parfois nécessaire, ce qui peut augmenter encore la surcharge de calcul. Inversement, pour des modèles peu canoniques, c'est-à-dire se distinguant par une grande variété de constructions intervenant dans la recherche, la surcharge de calcul relative diminue.

```
Data:                                    Représentation du problème : nombre
 m:integer = ...;                        de contraintes m, nombre de variables
 n:integer = ...;                        n et définition des contraintes dans un
 cl:array[1..m] of clause = ...;         tableau cl de m clauses.

Variable:                                Déclaration des variables : affectations
 a:array[1..n] of Boolean;               des booléens présents dans les clauses.

Invariant:
Définition des grandeurs automatiquement mises à jour à chaque itération : nombre
de littéraux valides nbtl[i] de cl[i] et nombre de clauses satisfaites nbClSat.
 nbtl[i in 1..m] = sum(j in cl[i].p)a[j] + sum(j in cl[i].n) !a[j];
 nbclsat = sum(i in 1..m)(nbtl[i]>0);

Satisfiable:                             Formulation du critère d'arrêt :
 nbClSat = m;                            toutes les clauses sont satisfaites.

Objective Function:                      Formulation de l'objectif : maximiser
 maximize nbClSat;                       le nombre de contraintes satisfaites.

Neighborhood:                            Programmation du voisinage : parmi
 select best a[i]:=!a[i] where           les affectations, choix de l'interversion
 i in 1..n accept when noDecrease;       conduisant au meilleur objectif.

Start:                                   Initialisation de la recherche locale :
 forall(i in 1..n) random(a[i]);         affectation aléatoire des booléens.
```

Figure 2.4 – Exemple de code LOCALIZER pour le problème *SAT*

TABOUCSP (Galinier et Hao, 2000). TABOUCSP met à disposition un ensemble de types de contraintes à partir desquelles le programmeur modélise son problème de satisfaction. Pour chaque type de contraintes, TABOUCSP définit une fonction qui mesure la distance entre la validité de la contrainte et son état pour une certaine affectation courante des variables. TABOUCSP minimise une agrégation des fonctions de distance apparaissant dans le système de contraintes. Contrairement à LOCALIZER, TABOUCSP n'offre pas de couche de contrôle programmable : l'algorithme utilisé est invariablement une recherche taboue basée sur l'interdiction de changer la valeur

d'une variable fraîchement réaffectée. TABOUCSP associe lui-même les grandeurs caractéristiques à tracer en fonction des contraintes présentes et de l'algorithme fondateur utilisé. Également, le programmeur ne peut pas étendre le vocabulaire formé par les types de contraintes pris en charge par le solveur. Ce manque de flexibilité devrait naturellement trouver compensation dans une surcharge en temps de calcul moins importante que dans le cas de LOCALIZER.

Dans la perspective d'un langage de programmation pour la recherche locale, l'expressivité et la concision que propose au programmeur la couche déclarative des langages qui viennent d'être présentés constituent un intérêt tout particulier. Des voisinages assez complexes comme les voisinages à profondeur variable ont ainsi pu être écrits de manière compacte et dans un style proche d'une description en pseudo-code (Michel et Van Hentenryck, 2001). Cependant, même si nombres de problèmes mathématiques peuvent être ramenés à la fois dans le champs de l'optimisation et dans celui de la satisfaction de contraintes, il reste que les langages déclaratifs, centrés sur la satisfaction de contraintes, ne sont pas conçus pour résoudre efficacement ceux dont la combinatoire concerne, par nature, l'optimisation. L'expérience montrerait certainement que les frontières entre problèmes qui se formulent efficacement comme une satisfaction de contraintes ou comme de l'optimisation sont les mêmes, qu'on se situe dans le cadre de résolution heuristique de la recherche locale ou dans le cadre de résolution exact des méthodes arborescentes. La conception des langages déclaratifs, qui repose sur une modélisation analytique des problèmes à résoudre, ne se transpose par ailleurs pas toujours aux problèmes d'optimisation pour lesquels la recherche locale repose parfois sur des mouvements à caractère géométrique.

2.6 Conclusion

Dans ce chapitre, nous avons documenté la problématique concernant l'élaboration d'un utilitaire de programmation pour la recherche locale. Nous avons pour cela tenté

de déterminer, d'un côté, les besoins des programmeurs, et de l'autre, l'espace des possibilités techniques. Les environnements de programmation que nous avons présentés suite à cette analyse visent chacun des objectifs particuliers. Dans le cas des bibliothèques, il s'agit de proposer au programmeur une palette d'algorithmes directement utilisables pour une famille de problèmes similaires. Ces algorithmes gardent leur efficacité puisqu'ils tiennent compte des spécificités des problèmes à traiter. Dans le contexte des cadres d'application, l'usager est tenu d'écrire lui-même les structures de données dépendant de son domaine, il dispose alors d'une famille de métaheuristiques qu'il peut tester, comparer ou encore hybrider. Cette approche vise la réusabilité des algorithmes classiques de recherche locale. Enfin, dans le cas des langages déclaratifs, l'usager peut programmer très simplement une recherche locale pour le problème qui l'intéresse à partir du moment où la programmation par contraintes autorise une bonne modélisation. Dans cette thèse, nous proposons d'étudier une quatrième approche, les librairies génériques. L'objectif visé est d'établir un protocole autour duquel peuvent s'échanger deux types d'expertise, l'une correspondant à la représentation des problèmes et l'autre à la composition de schémas algorithmiques. D'un côté, les usagers désireux de faire de la recherche en algorithmique peuvent ainsi travailler sur de nouveaux élements heuristiques qu'ils testent sur des structures de problème disponibles à travers l'utilitaire. De l'autre, les usagers désirant développer une application particulière fournissent leurs structures de problème et bénéficient des éléments algorithmiques disponibles à travers l'utilitaire. Pour remplir ce double objectif, une librarie générique devrait présenter les caractéristiques suivantes de généricité, de modularité et d'efficacité.

Généricité. Un utilitaire permettant de développper de nouvelles méthodes de recherche locale et de les utiliser pour de nouveaux problèmes doit être en mesure de supporter n'importe quel domaine d'application. Dans ce contexte, les bibliothèques, centrées autour d'un domaine, ainsi que les langages déclaratifs, se concentrant sur

des problèmes de satisfaction de contraintes, perdent un peu de leurs intérêts. Les cadres d'application, applicables à n'importe quel problème formulable en terme de recherche locale, constituent par contre le point de départ de notre réflexion. Notons cependant que ces derniers visent principalement à offrir une palette d'algorithmes standardisés. Si cette ambition a l'avantage d'offrir à l'usager des métaheuristiques toutes faites, elle présente deux inconvénients majeurs. Premièrement, l'architecture de l'utilitaire porte la marque des algorithmes dont il dispose. Par exemple, les composantes qui capturent les notions d'attributs d'une méthode taboue apparaissent de manière hétérogène dans les schémas de conception. L'intégration de nouvelles métaheuristiques n'est ainsi pas favorisée. Deuxièmement, on ne peut pas adapter efficacement les versions canoniques des algorithmes proposés. Ce manque de flexibilité, qui va à l'encontre d'une personnalisation coutumière des schémas de recherche locale connus, est dû à la centralisation des schémas algorithmiques autour d'une seule structure, par exemple, `RecuitSimule` ou `RechercheTaboue` dans la figure 2.1.

Modularité. Les librairies génériques corrigent les inconvénients des cadres d'application à deux titres. Premièrement, nous proposons de prolonger la décomposition de la recherche locale au niveau des représentations de problèmes. À titre d'illustration, mentionnons que les fonctions `faireMouvement`, `deltaValeur` et `prochainMouvement`, qui apparaissent dans la même entité logicielle sur la figure 2.1, correspondent dans notre approche à trois composantes différentes, la première en tant que mouvement, la seconde en tant qu'attribut de mouvement et la dernière en tant que serveur de mouvement. Il devient alors possible de considérer simultanément plusieurs fonctions de coût ou encore plusieurs schémas d'énumération sans changer pour autant de voisinage, une situation ne pouvant être reproduite naturellement dans le contexte des cadres d'application. Deuxièmement, nous proposons d'introduire une décomposition au niveau des schémas algorithmiques. Cette perspective requiert d'examiner la recherche locale au-delà d'un examen propre à chaque algorithme puisqu'il s'agit

de dégager les concepts élémentaires et transversaux (Qu'est ce qu'un mouvement, une solution, un type de parcours, un serveur de mouvement, un attribut de mouvement ?) à partir desquels il devient possible de formuler chaque algorithme. Au plan du langage, chaque concept comporte un ensemble de spécifications qui assure son intégration dans une recherche locale quelque soit le fonctionnement interne programmé par l'usager, lequel programme finalement sa recherche locale comme une composition propre à laquelle participent les unités logicielles qui l'intéressent.

Efficacité. Un dernier critère important pour la conception des librairies génériques concerne l'efficacité des méthodes de résolution obtenues. Il s'agit là d'un aspect non négligeable puisque les temps de calculs et les résultats numériques constituent un argument déterminant des résultats publiés dans la littérature. Nous avons vu que la surcharge de calcul pouvait s'établir à des normes de 30% dans le cas d'un cadre d'application et de 238% à 482% dans le cas d'un langage déclaratif. Le critère d'efficacité doit considérer deux facteurs autour desquels il faut trouver un bon compromis. D'un côté, la modularité d'une librarie générique permet d'adapter au cas par cas les parties algorithmiques pertinentes, ce qui œuvre dans le sens d'une meilleure efficacité des algorithmes obtenus. Mais de l'autre, plus on propose de possibilités à l'usager, moins on peut faire d'hypothèses de fonctionnement, et plus il est difficile de spécialiser et d'optimiser les canaux de programmes pris en charge. Nous soulignons à cet égard que la programmation générique s'appuie exclusivement sur le polymorphisme statique. Cette technologie présente l'avantage de modeler la forme de l'algorithme en fonction des spécifications de l'usager dès la compilation. Contrairement aux cadres d'application bâtis, à l'exception de HOTFRAME, sur les fonctions virtuelles du C++, l'exécutable qui découle d'une librarie générique est donc exclusivement concentré sur le déroulement de la recherche locale.

Alors que la deuxième partie de la thèse introduit une base formelle à l'approche des libraries génériques, la troisième partie en démontre la réalisabilité par la présentation de l'utilitaire METALAB et en illustre l'intérêt par quelques études de cas.

Deuxième partie

Modélisation de la recherche locale

La deuxième partie de l'ouvrage, théorique, se consacre à la conception des librairies génériques pour la recherche locale. Nous avons affiché pour ce type d'environnement deux objectifs : faciliter le développement de nouvelles constructions d'algorithmes de recherche locale et faire en sorte que cette collection extensible soit générique, c'est-à-dire appliquable à de nouveaux problèmes. Très peu d'hypothèses pouvant être ici formulées, le travail de conception consiste à formaliser ce qui constitue la recherche locale au plan informatique. Compte tenu de la volatilité des algorithmes, cette formalisation est inévitablement réductrice et nous tâcherons surtout de trouver le meilleur compromis entre couverture et compacité du modèle. Le polymorphisme des structures de mouvements et de solutions constitue en outre un problème fondamental à contourner puisqu'on souhaite ne pas se limiter à une famille remarquable de problèmes. Les cadres d'application ont à cet égard mis en évidence l'approche qui semble la plus pertinente et notre travail en prolonge les perspectives : notre modèle de recherche locale est établi dans les termes de la programmation générique. Notre exposé couvre deux objectifs. Premièrement, il a semblé bénéfique de rendre compréhensible en préalable la programmation générique, mais tout en montrant progressivement comment les constructions informatiques qu'elle autorise peuvent être investies judicieusement dans la construction de librairies génériques pour la recherche locale. Deuxièmement, nous établissons des mécanismes qui maintiennent automatiquement l'état de la recherche en fonction des unités logicielles entrant en interaction.

Le chapitre 3 commence par introduire le formalisme et la méthode de la programmation générique. Les concepts, qui représentent dans ce paradigme des catégories de composantes informatiques, sont tout d'abord introduits. Sous les figures de mouvements, de solutions ou encore d'attributs, nous montrons sous quels aspects ces derniers apparaissent en programmation générique en général, et en recherche locale en particulier. Le chapitre montre ensuite comment structurer la programmation de la recherche locale par l'organisation de ses concepts. Les composantes informatiques sont au préalable découplées en trois familles : les concepts de Contrôle, de Variables et de Représentation. Les concepts de Contrôle, qui correspondent aux composantes algorithmiques, paramétrisent la recherche en fonction d'échantillons d'information qu'on appelle des Variables, et qui sont prélevés sur les formes de Représentation du problème rencontrées en cours de recherche. La famille des Variables est ensuite hiérarchisée selon la portée et l'objet des informations recueillies. Enfin, on introduit le diagramme de composition des Variables, lequel permet de modéliser le cas de Variables complexes dont l'expression s'appuie sur d'autres Variables.

Dans le chapitre 4, on interprète le modèle en précisant le mode d'interaction, le cycle de vie ainsi que l'algorithme de maintenance de chaque composante. Nous commençons par illustrer l'intérêt de maintenir automatiquement la recherche en comparant deux programmes, l'un assemblé automatiquement au moyen de la librairie METALAB, et l'autre déroulé manuellement. Pour les trois événements succeptibles de modifier l'état de la recherche (l'initialisation de la recherche, l'énumération d'un nouveau Mouvement et la transition vers une Solution voisine), nous établissons ensuite les mécanismes de maintenance en nous appuyant sur des diagrammes de composition qui reflètent l'interdépendance des composantes devant être mises à jour. La réalisation de ces mécanismes constitue le moteur de la librairie METALAB.

CHAPITRE 3 : CONCEPTION GÉNÉRIQUE DE LA RECHERCHE LOCALE

La programmation de la STL, la librairie standard du C++, s'appuie sur la programmation générique. Cette librairie très populaire, dont l'architecture présente certaines analogies avec notre structuration de la recherche locale, n'a cependant été assimilée que très partiellement par la majorité des programmeurs. Le fait que la programmation générique s'illustre pour un petit nombre de chantiers comparativement à des méthodologies plus courantes comme l'orienté-objet, explique certainement pourquoi ses méthodes de conception ont encore mal pénétré les esprits. Nous prenons donc soin de constituer progressivement notre modèle générique de recherche locale tout en familiarisant le lecteur au formalisme nécessaire. Nous espérons que ce parallèle favorise, d'une part, l'assimilation des méthodes de conception génériques, et d'autre part, la compréhension de la nature informatique de la recherche locale.

3.1 Formalisme et méthodologie

Cette section introduit le formalisme des concepts utilisés en programmation générique. Après avoir mis en évidence la notion de concept (section 3.1.1), nous montrons comment les décrire au moyen d'expressions valides (section 3.1.2) et de types associés (section 3.1.3). Le contexte des librairies génériques pour la recherche locale accompagne les trois parties de ce développement au moyen d'exemples compréhensibles qui anticipent certaines constructions utilisées par la librairie METALAB.

3.1.1 Mise en évidence des concepts

En programmation générique, un concept représente un ensemble de structures informatiques jouant un même rôle dans un contexte de programmation donné. Par exemple, le concept de mouvement apparaît naturellement dans le cadre de la recherche locale. En effet, pour des applications différentes, on distingue au sein des programmes une entité fonctionnellement invariante qui assure la transformation de la solution courante vers une solution voisine. En pratique, l'entité de mouvement n'est cependant pas mise en relief puisque sa réalisation et son utilisation sont souvent intimement noyées dans le reste du programme. Comme nous allons le voir, la programmation générique permet de mettre en évidence et d'isoler de telles entités. Il devient alors possible de mettre en œuvre des pièces de programme interchangeables, interactives et correspondant chaque fois à un concept. Cette manière de concevoir un programme conduit finalement à une forme de langage pour la recherche locale dont les concepts servent de pièces élémentaires.

Pour la suite, nous désignons les concepts au moyen d'une police appropriée : par exemple, Mouvement, pour le concept représentant les structures de mouvement. Par ailleurs, les éléments de code (exclusivement du C++) sont représentés au moyen d'une seconde typographie. Par exemple, le type `Twex` (pour *Two-exchange*) représenterait la structure de donnée (en C++, une classe) du mouvement 2-*opt* dans le contexte d'un PVC. Dans ce cas, on dira indifféremment que `Twex` est un modèle de Mouvement, une réalisation de Mouvement, ou tout simplement, un Mouvement.

Un concept est une notion ensembliste, et à ce titre, peut être décrit de deux façons duales. D'une part, un concept peut être défini comme une liste de spécifications caractéristiques d'une famille de types. D'autre part, un concept peut être vu comme l'ensemble des types qui vérifient ces spécifications. Une modélisation cohérente de

programmation générique consiste à ce que cet ensemble de types corresponde à un certain invariant fonctionnel au sein du domaine de programmation étudié. Les spécifications d'un concept comportent deux aspects. D'une part, certaines spécifications décrivent les expressions valides que doit vérifier un type afin qu'il modélise le concept correspondant. D'autre part, certaines spécifications portent sur des types externes qui lui sont associés, c'est-à-dire ceux sur lesquels reposent son fonctionnement interne ainsi que son intégration dans la structure logicielle à laquelle il participe.

3.1.2 Expressions valides associées à un concept

Examinons le code C++ de la figure 3.1. Nous rappelons que le mot-clé `template` permet de programmer des composantes gabarits, c'est-à-dire paramétrées par des types formels (cf. annexe A.1).

```
template <class T> T& min(T& a, T& b) { return (a<b) ? a : b;}

main()
{
  std::cout << min(1,2) << std::endl;        Utilisation de min<int>
  std::cout << min(1.1,2.8) << std::endl;    Utilisation de min<float>
}
```

Figure 3.1 – Mise en évidence des expressions valides

La fonction `min` est ainsi écrite en fonction du type `T`. Dans ce cas, lorsque le compilateur détecte que l'expression `min(1,2)` (resp. `min(1.1,2.8)`) correspond au contexte du type `int` (resp. `float`), il copie une version de `min` en remplaçant chaque `T` par `int` (resp. `float`) et compile le programme sur la base du code obtenu. Lorsqu'on utilise ce mécanisme, il est impératif de déterminer pour quels types de paramètres, la fonction générique demeure valide, c'est-à-dire compilable. Ce serait ainsi le cas pour la fonction `min`, dès lors que deux objets de type `T` permettent d'être comparés. Le

fait d'être comparable donne lieu à un premier exemple de concept, Comparable, qui désigne l'ensemble des structures `T` pour lesquelles `t1 < t2` est une expression valide lorsque `t1` et `t2` appartiennent au type `T`. Comparable, au même titre qu'Affectable (`t1 = t2` est une expression valide), que Déréférençable (`*t1` est une expression valide) sont des concepts très simples et très généraux.

Des concepts plus complexes se représentent par une liste plus longue d'expressions valides. Dans le cadre de la recherche locale par exemple, un Serveur est une entité responsable de l'énumération des coordonnées de mouvement autour de la solution courante. Comme nous le verrons, la liste des expressions valides est dans ce cas constituée des appels de fonction `begin(...)` et `next(...)` pour des arguments qu'il reste à préciser. En des termes informels, ceci revient à dire qu'un Serveur doit permettre d'amorcer l'énumération du voisinage selon des coordonnées initiales (et arbitraires) de mouvement et d'incrémenter si possible ces coordonnées. La fonction `next` doit enfin retourner un booléen indiquant la validité de l'incrémentation. Le concept de Mouvement est quant à lui constitué d'une seule expression valide qui correspond à l'appel de la fonction `move` servant à transformer la Solution courante selon les coordonnées internes du Mouvement. La Solution transformée correspond enfin également à un concept dans lequel est représenté la réponse affectée au problème.

Mentionnons que la distinction entre concepts de Mouvement et de Serveur marque une différence nette avec les cadres d'application pour lesquels toutes les fonctionalités correspondantes sont réunies au sein de la seule composante `Exploration`. Nous verrons que cette distinction s'avère très utile lorsqu'on souhaite définir différents types d'énumération tout en gardant la même structure de mouvement. Notons aussi que contrairement aux cadres d'application, la fonction qui calcule le coût d'un Mouvement n'appartient ni à la définition d'un Mouvement ni à celle d'un Serveur. D'un point de vue conceptuel, un Mouvement et un Serveur sont en effet des opérateurs : ils incrémentent l'état de la recherche, l'un au niveau de la trajectoire des Solutions,

et l'autre, au niveau de l'exploration des Mouvements du voisinage. En revanche, le coût d'un Mouvement est une observation associée à un Mouvement, et à ce titre, appartient au concept d'Attribut. La notion d'attribut utilisée par la méthode taboue et le coût de passage entre deux solutions sont donc ici présentés sous les mêmes traits conceptuels. Cette interprétation nous paraît davantage correspondre à la nature informatique qu'incarnent ces deux entités : une mesure associée à un Mouvement, ce qui conduit à un modèle de recherche locale plus adéquat et donc plus expressif. Il devient par exemple possible d'utiliser un même Mouvement pour des objectifs aussi différents qu'une stratégie de descente ou une stratégie de jonction entre solutions élites. Également, il devient possible de conserver le graphe de recherche pour des problèmes similaires dont la seule différence concerne la fonction objectif à optimiser. C'est le cas par exemple de la version de PVC dépendant du temps, notée $PVCDT$, où le coût de chaque arête formant la solution dépend à la fois des sommets reliés et de la position de l'arête dans la tournée. Nous verrons que pour résoudre un $PVCDT$, il suffit de réutiliser la Solution, le Mouvement et un des Serveurs programmés pour résoudre le PVC, seul l'Attribut de gain devant être redéfini.

```
template <class Dst> struct TspGain : ScanWith<Twex<Dst>, Dst>
{
  int value;
  void scan(const Twex<Dst>& mvt, const Dst& dst)
  {  value = dst(mvt.c[0][0],mvt.c[0][1]) + dst(mvt.c[1][0],mvt.c[1][1])
           - dst(mvt.c[0][0],mvt.c[1][0]) - dst(mvt.c[0][1],mvt.c[1][1]);
  }
};
```

Figure 3.2 – Réalisation de l'`Attribut` TspGain

À titre d'illustration, observons le code METALAB de la figure 3.2. La composante `TspGain`, est un Attribut mesurant le coût de passage entre deux Solutions d'un PVC. La méthode `scan` sert à calculer la valeur de l'Attribut et à la mettre de côté. Pour cela, elle prend en argument le Mouvement `mvt` énuméré ainsi que le Problème à

résoudre, la structure de distance `dst`. Notons que `TspGain` est programmée génériquement en fonction du type `Dst`, ce qui permet de travailler sur une variété de Problèmes, à distance fonctionnelle ou matricielle par exemple. Ici, `TspGain<Dst>` compile si l'expression `dst(c1,c2)` retourne la distance entre deux villes `c1` et `c2`. Les coordonnées d'un `Twex` sont constituées d'un tableau `c` à deux lignes et deux colonnes dont la première (resp. deuxième) ligne contient les indices des sommets de la première (resp. deuxième) arête sortante.

```
template <class SVR, class GAIN> struct FirstImprovement : FindWith<SVR, GAIN>
{
  bool find(const GAIN& gain)
  {
     while (gain.value <= 0 && next());
     return (gain.value > 0);
  }
};
```

Figure 3.3 – Réalisation de l'Explorateur FirstImprovement

La composante `FirstImprovement`, présentée à la figure 3.3, constitue un Explorateur et participe à ce titre à la sélection d'un mouvement dans le voisinage. Le résultat de l'Explorateur, le Mouvement retenu, est enregistré automatiquement dans l'Explorateur à la sortie de `find`. En tant que composante gabarit, `FirstImprovement` peut être utilisée dans des contextes autre que la résolution d'un PVC. De manière générale, l'Explorateur peut être compilé pour tout Attribut `GAIN` passé en paramètre dont `value` est une expression valide. En tant qu'Attribut, `GAIN` doit alors fournir une fonction `scan` qui prélève l'information pertinente sur le Mouvement considéré. Pour résoudre un PVC au moyen d'une descente en première amélioration basée sur un 2-*opt*, on utiliserait par exemple l'Explorateur `FirstImprovement<SVR,TspGain<Dst>>` pour un Serveur `SVR` permettant d'énumérer sur des coordonnées de `Twex`.

3.1.3 Types associés à un concept

La liste des expressions valides ne suffit pas toujours à caractériser un concept. Il est en effet parfois nécessaire de mentionner quels types sont associés à ce concept. De même que les spécifications portant sur les expressions valides, la diffusion de ces types participe à la cohésion entre les diverses composantes du programme générique. En guise de préambule, l'annexe A.2 présente un exemple simple de transmission des types associés.

Dans la composante `FirstImprovement`, on remarque ainsi que la méthode d'énumération `next` est appelée sans argument depuis la fonction find. Cet absence d'arguments est importante car elle permet aux programmeurs d'écrire et/ou de réutiliser leur Explorateur indépendamment des spécificités des Serveurs qu'ils utilisent. En interne, un Explorateur doit malgré tout connaître les véritables arguments de `next` dans le Serveur de sorte que l'appel puisse être relayé aux véritables fonctions d'énumération. Il est pour cela nécessaire que le Serveur déclare explicitement la liste des types d'arguments de ses méthodes `next` et `begin`. Cette liste de type devient alors associée au Serveur, et par transitivité, à l'Explorateur. En particulier, le type de Mouvement sur lequel opère un Serveur est un type associé de même qu'implicitement le type de Solution sur lequel opère ce Mouvement. Tous ces types, une fois portés à la connaissance de l'Explorateur, permettent finalement de générer en temps de compilation les mécanismes de fonctionnement de l'Explorateur en fonction du contexte d'utilisation. En pratique, ce travail interne est effectué en dérivant `FirstImprovement` de la classe `FindWith` en fonction des types qui définissent le contexte, ici `SVR` et `GAIN`.

L'utilisation de l'Attribut `GAIN` dans `FirstImprovement` donne un deuxième exemple de type associé. Remarquons à ce propos que `GAIN` est passé en référence constante dans la fonction `find`. Puisque l'accés à `gain` est limité en lecture, son état doit donc

être automatiquement maintenu en phase avec le Mouvement référé par le Serveur. Ce mécanisme est en fait intégré dans la fonction `next` utilisée dans `find` : chaque fois que le Serveur incrémente les coordonnées du Mouvement énuméré, les Attributs utilisés dans l'Explorateur sont automatiquement recalculés. Pour cela, l'Explorateur doit connaître en interne la signature de chaque fonction `scan` qu'il déclenche. De même que pour les fonctions d'énumération d'un Serveur, un Attribut déclare à cet effet explicitement le type de chaque argument que prend sa fonction scan. Dans l'exemple de la figure 3.2, `Twex<Dst>` et `Dst` constituent ainsi la liste des arguments associés à `scan`. En pratique, la déclaration de ces deux types associés s'effectue simplement en dérivant `TspGain` de la classe `ScanWith<Twex<Dst>,Dst>`.

La déclaration explicite des arguments utilisés dans chaque méthode se généralise facilement à l'ensemble des composantes d'une librairie. Au moyen de ce mécanisme, il devient possible de programmer la charpente de l'algorithme à générer en fonction des listes d'arguments que prennent chacune des fonctions impliquées dans le programme. En suivant cette logique, on remarque par exemple qu'en dérivant `FirstImprovement` de `FindWith<SVR, GAIN>`, on indique en fait à METALAB quels Attributs la fonction `find` doit-elle mettre à jour. Pour des Explorateurs plus complexes opérant sur davantage d'Attributs, comme par exemple ceux employés par la méthode taboue, on ajouterait le nombre nécessaire d'Attributs à la suite de `GAIN`. L'exemple de `FirstImprovement` montre par ailleurs comment il est possible de créer soi-même de nouvelles procédures de choix de mouvements sans avoir à rentrer dans les technicités de la librairie. Le concept d'Explorateur est en effet isolé du noyau de la librairie au même titre que les concepts de Mouvement, de Serveur ou d'Attribut.

3.2 Principes de modélisation

En programmation générique, les concepts symbolisent des unités logicielles dont la mise en évidence constitue la première étape de la conception d'un modèle de

programmation. Dans une deuxième étape, on finalise la structuration du modèle en établissant des relations entre les différents concepts. Comme le rappelle la figure 3.4, deux types de relations régissent l'interaction entre concepts.

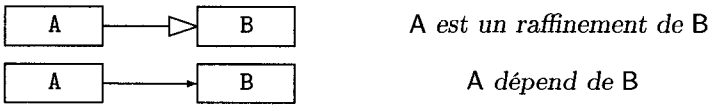


Figure 3.4 – Notation des relations de dépendance et de raffinement

Relations de dépendance. Les relations de dépendance concernent le cas où la définition et/ou le fonctionnement d'un concept repose sur la connaissance et/ou l'utilisation d'un autre concept. Remarquons par exemple que le concept d'Attribut dépend du concept de Mouvement. Dans le cadre de cette relation de dépendance, l'Attribut `TspGain` est ainsi dérivé d'une classe gabarit `ScanWith` qui prend entre autres l'argument `Twex`.

Relations de raffinement. Les relations de raffinement apparaissent lorsqu'un concept est un cas particulier d'un concept plus général : un concept A est un raffinement d'un concept B si l'ensemble des types modélisant A est un sous-ensemble des types modélisant B. Autrement dit, la liste des spécifications de A contient la liste des spécifications de B. Nous verrons dans la prochaine section qu'une Variable désigne dans notre modèle une composante logicielle responsable de la prise d'information concernant le cours de la recherche. Dans ce contexte, le concept d'Attribut constitue le raffinement d'une Variable pour des observations faites autour d'un Mouvement. De même, nous définirons le concept de Caractéristique comme le raffinement d'une Variable pour des observations faites autour d'une Solution. Une Caractéristique dispose à cet égard d'un constructeur permettant d'initialiser la composante ainsi que d'une fonction `track` permettant de maintenir son état chaque fois qu'un Mouvement est exécuté. La valeur de la fonction objectif à optimiser constitue un exemple de Caractéristique.

Tableau 3.1 – Résumé des concepts de la recherche locale

Concept	Synopsis	Fonctions conceptuelles
Problème	Regroupe les données paramétriques de l'instance à résoudre	Aucune
Solution	Initialise et représente la réponse apportée au Problème	Constructeur
Mouvement	Transforme la Solution courante selon des coordonnées internes	`move`
Attribut	Calcule une information devant être en phase avec le Mouvement examiné ou exécuté	`scan`
Caractéristique	Initialise et maintient une information devant être mise à jour à chaque exécution de Mouvement	Constructeur et `track`
Serveur	Initialise puis modifie les coordonnées du Mouvement examiné	`begin` et `next`
Explorateur	Visite le voisinage de la Solution courante pour sélectionner un Mouvement	`find`

Le tableau 3.1 ci-dessus résume l'ensemble des concepts de la recherche locale que nous avons mis en évidence en ce début de chapitre. Nous proposons pour la suite de dresser le schéma des relations autour duquel ces concepts s'organisent. Nous avançons pour cela trois principes. Dans un premier temps (section 3.3), nous montrons l'intérêt de découpler les relations de dépendance entre concepts au moyen de concepts médiateurs appelés Variables. Dans un deuxième temps (section 3.4), nous classifions l'ensemble des Variables selon la portée de l'information qu'elles transmettent. Cette hiérarchisation des Variables permet d'opérer la synthèse d'algorithme la plus efficace possible en fonction des hypothèses les plus fines qu'on peut faire sur les composantes de recherche locale entrant effectivement en interaction pour une application donnée. Finalement, dans un troisième temps (section 3.5), nous introduisons le diagramme de composition des Variables, qui permet de programmer les cas où des Variables élaborent des informations complexes en fonction d'autres Variables.

3.3 Découplage des parties logicielles

Nous avons vu que les associations de type sont parfois transitives. Par exemple, l'Explorateur `FirstImprovement<SVR, TspGain<Dst>>` dépend indirectement du type de Mouvement que l'Attribut `TspGain` est en mesure de sonder. Le concept à travers lequel transite le type associé, ici l'Attribut, joue un rôle de médiateur : il autorise la communication entre deux concepts, l'Explorateur et le Mouvement, tout en évitant que ceux-ci n'interragissent directement. Une méthode de conception importante, qu'on retrouve en programmation générique, consiste à découpler les entités informatiques de sorte à minimiser leur interdépendance. Cette approche conduit généralement à faire apparaître des concepts qui jouent ce rôle de médiateur. Ces derniers, qui relayent l'interaction entre concepts fortement typés, assurent une flexibilité et une réusabilité maximale des pièces logicielles du programme car les hypothèses faites sur la compatibilité entre parties communicantes est réduite au minimum.

La conception de la STL en C++ offre un exemple bien connu de découplage de concepts au moyen de concepts médiateurs appelés Itérateurs. Dans cette librairie, un Itérateur, comme par exemple un pointeur, représente une entité permettant d'accéder et de traverser des Conteneurs tels que des listes ou des tableaux. Un Itérateur permet dans ce cas de découpler l'interaction entre un Conteneur et un Algorithme, par exemple une fonction de tri, qui lui serait appliqué. En effet, l'Algorithme est décrit seulement au moyen de fonctionnalités (déréférencement, parcours des éléments des Conteneurs) déléguées à l'Itérateur de sorte qu'un Algorithme ne fait pas d'hypothèses sur le Conteneur auquel il s'applique mais seulement sur l'Itérateur qu'il utilise.

Comme l'illustre la figure 3.5, nous proposons de suivre une approche analogue pour décrire la recherche locale. En effet, de la même manière que des Algorithmes opérent sur des Conteneurs, la recherche locale se caractérise par l'interaction entre, d'un

côté, des Représentations de problème, et de l'autre, des formes de Contrôle où sont prises les décisions concernant la trajectoire de recherche. Le rôle des médiateurs est joué dans notre modèle par les Variables qui détiennent l'information en fonction de laquelle les composantes de Contrôle dirigent la recherche.

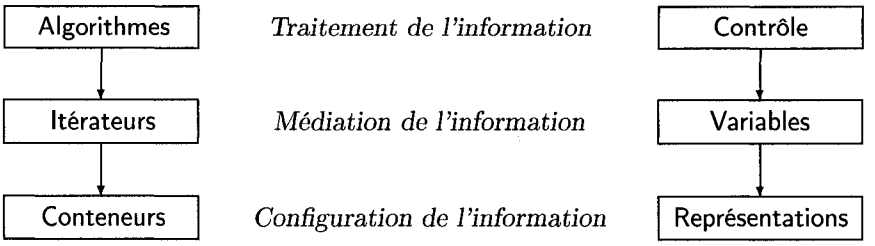


Figure 3.5 – Découplage en algorithmique et en recherche locale

Concepts de Représentation. Responsables de la configuration de l'information, les composantes de Représentation désignent les structures qui permettent de formuler un problème de décision et de définir l'état de la recherche dans le cadre de la recherche locale. Ces composantes incluent les entités logicielles entrant dans la composition du graphe de recherche sous forme de Solutions et de Mouvements. Font également partie des formes de Représentation les composantes, appelées Problèmes, qui contiennent les données paramétriques permettant de caractériser une instance de problème. La réalisation des concepts de Représentation dépend fortement du problème à formuler. Comme nous avons vu au chapitre 2, les cadres d'application intègrent déjà ces différentes formes de Représentations.

Concepts de Contrôle. Responsables du traitement de l'information, les composantes de Contrôle désignent les structures permettant de programmer la sélection des Mouvements et donc d'orienter la trajectoire des Solutions visitées. Ces composantes comprennent les entités logicielles responsables du mode de sélection d'un mouvement (concept d'Explorateur) ainsi que de l'énumération du voisinage (concept

de Serveur). Généralement, la programmation d'un Explorateur ne dépend pas du problème auquel il est appliqué. Un Explorateur dépend en revanche des types de Variables en fonction desquelles sont calculées les décisions de trajectoire. La programmation d'un Serveur dépend généralement du Mouvement dont il énumère les coordonnées. Comme nous verrons, un Serveur peut dépendre également de Caractéristiques calculées autour de la Solution courante telles que, par exemple, une composante qui maintiendrait l'ensemble des mouvements trié en fonction de leur gain.

Concepts de Variables. Responsables de la médiation entre les deux parties logicielles précédentes, les Variables rassemblent les structures de données responsables des prises d'information effectuées en cours de recherche. Souvent, leur fonctionnement interne dépend du type de problème qu'on souhaite résoudre, comme par exemple lorsqu'il s'agit de programmer le coût de passage entre deux Solutions ou encore le coût d'une Solution. Il existe aussi des cas où la programmation d'une Variable ne dépend pas directement des Représentations mais plutôt d'autres Variables passées en argument. Une Variable comme un registre de fréquences ne repose ainsi que sur la Variable qu'elle répertorie. Ces types de Variable sont alors réutilisables : ils peuvent être conjugués à différents schémas de Mouvement.

L'Explorateur `FirstImprovement` que nous avons présenté en préalable illustre le principe de découplage. En effet, cette composante de Contrôle accède seulement indirectement aux Mouvements visités à travers l'Attribut `GAIN`. L'écriture de l'Explorateur ne dépend pas des formes de Représentation d'un problème mais seulement des propriétés d'une Variable médiatrice. L'Explorateur peut ainsi être utilisé pour tout problème pour lequel on dispose d'un Serveur ainsi que d'un Attribut ayant un champ `valeur`. En particulier, `FirstImprovement` peut être appliqué à résoudre un PVC en conjugaison avec l'Attribut `TspGain`. Comme le montre ce cas de figure, le principe de découplage contribue à la généralité et la réusabilité de chaque pièce logicielle. Il permet de séparer au maximum la programmation algorithmique de la programmation des structures de problèmes.

3.4 Modulation des canaux de programme

De même que la première étape de conception visait la réusabilité de chaque unité logicielle, cette seconde étape vise l'efficacité des programmes obtenus. Les relations de raffinement sont ici fort utiles. Elles permettent en effet de sélectionner des canaux de programmes qui sont le plus efficaces possible compte tenu des hypothèses les plus fortes qu'on peut faire sur les entités logicielles concertantes. Cette propriété permet de maintenir la généralité du schéma de découplage ainsi que la réusabilité du code tout en assurant une efficacité maximale pour chaque situation particulière. Puisque les concepts médiateurs sont disposés au centre de l'interaction entre les différentes parties logicielles, les relations de raffinement portent principalement sur cette famille de concepts. Dans un premier temps, nous proposons d'illustrer l'intérêt de cette méthode de conception sur un exemple simplifié tiré de la STL. Dans un deuxième temps, nous mettons en évidence les relations de raffinement rencontrées en recherche locale.

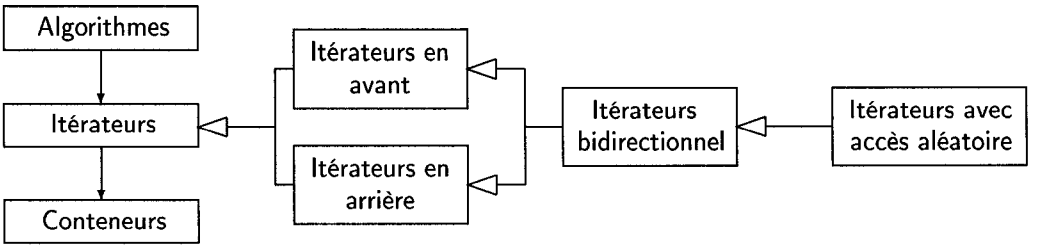


Figure 3.6 – Relations de raffinement entre Itérateurs

La conception de la STL fournit un exemple simple de programmation générique basée sur la hiérarchisation des médiateurs. Comme l'illustre la figure 3.6, il existe en effet des versions pauvres d'Itérateurs ne permettant que de traverser un Conteneur de manière incrémentale, soit en avant (Itérateur en Avant), soit en arrière (Itérateur en Arrière), soit les deux (Itérateur Bidirectionnel). Certains Itérateurs, appelé Iterateurs

avec accès aléatoire, offrent en revanche l'opportunité d'accéder directement à un élément quelconque et constituent à ce titre la version d'Itérateur la plus raffinée. Dans ce contexte, si un Algorithme s'appliquant à un Conteneur peut être réalisé sous les propriétés faibles d'un Itérateur en Avant, il existe parfois une implantation plus performante utilisant les propriétés plus fortes d'un Itérateur avec accès aléatoire. Il est alors intéressant de maintenir le schéma de découplage, tout en autorisant d'avoir recours à des implantations efficaces compte tenu des types informatiques entrant effectivement en interaction.

Le C++ ne supporte pas directement les relations de raffinement, mais une combinaison de deux techniques permet d'atteindre les mêmes objectifs : le polymorphisme statique et les techniques d'étiquetage. La première consiste à redéfinir des versions de fonctions ou de classes gabarits pour des types particuliers pertinents passés en paramètre (cf. annexe A.3). La deuxième associe un type étiquette dans le but de déclarer explicitement à quel concept le plus raffiné d'une certaine famille, ici les Itérateurs, appartient une entité logicielle (cf. annexe A.4). Lorsque par exemple un Algorithme et un Conteneur entrent en interaction, l'étiquette de l'Itérateur permet de brancher à la compilation sur l'implantation la plus efficace qui ait été prévue dans la librairie générique.

La partie gauche de la figure 3.7 transpose le schéma de raffinement à l'ensemble des Variables. Comme nous l'avions anticipé, les Attributs désignent des observations faites autour d'un Mouvement, et les Caractéristiques, des observations faites autour d'une Solution. Notons que selon cette définition, ce qui différencie précisément un Attribut d'une Caractéristique est le moment auquel l'information doit être mise à jour : un Attribut doit être en phase avec le Mouvement auquel il se réfère et une Caractéristique avec la Solution à laquelle elle se réfère. Bien que ce soit le cas en général, cette définition n'implique pas forcément qu'un Attribut (resp. une Caractéristique) prélève une information concernant le Mouvement lui-même (resp. la Solution elle-même). Par

exemple, une Variable qui ordonne dans un conteneur l'ensemble des coordonnées de mouvement du voisinage courant selon leur gain, constitue une Caractéristique car elle doit être maintenue chaque fois qu'une nouvelle solution est atteinte. Au contraire, une Variable qui calcule la valeur que prendrait la solution courante si un mouvement lui était appliqué, constitue un Attribut. *C'est donc la portée et non l'objet de l'information calculée qui distingue les* Attributs *des* Caractéristiques. En fait, un modèle de recherche locale, a priori plus naturel, qui distinguerait les Variables selon l'objet de l'information qu'elles calculent ne couvrirait pas les deux exemples que nous venons de donner. En pratique, le raffinement des Variables tel que défini permet à la librairie, d'un côté, de savoir en temps de compilation à quels moments doivent être mises à jour chaque composante, et de l'autre, de passer indifféremment les Variables en argument de fonctions qu'il s'agisse en fait d'Attributs ou de Caractéristiques.

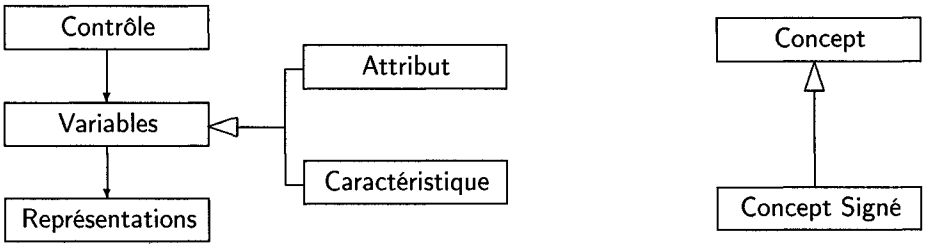


Figure 3.7 – Raffinement par Variables et par signature

La partie droite de la figure 3.7 indique que chaque concept de la recherche locale que nous avons introduit peut donner lieu à autant de raffinements possibles qu'il existe de signatures pour ses fonctions internes. Un Attribut `A1` dont la fonction `scan` prend en argument le Mouvement auquel il se réfère, et un Attribut `A2` dont la fonction `scan` prend en argument le même Mouvement ainsi que la Solution courante, constituent en ce sens deux raffinements d'Attribut distincts. Pour chaque concept, ce deuxième axe de hiérarchisation permet de définir des fonctions qui n'utilisent que des arguments dont elles ont vraiment besoin. Remarquons par exemple que la

fonction `next` d'un Serveur repose forcément sur la Solution courante dans le cas d'un *PVC*. Connaître la Solution courante est en revanche inutile lorsqu'on incrémente les coordonnées d'un Mouvement `Flip` pour un problème *SAT*. Si l'on souhaitait préserver la généralité du modèle sans le mécanisme de raffinement par signature de fonction, toutes les méthodes `next` devraient alors fournir en argument la Solution courante, ce qui alourdirait l'exécutable dans le cas du problème *SAT*. En raffinant chaque concept selon la signature de leurs fonctions, la définition et l'utilisation de chaque composante peuvent être faites sur mesure, et ce, à tous les niveaux de l'application.

3.5 Composition des Variables

Les parallèles avec la conception de la STL ont offert jusqu'ici des analogies pertinentes. Le schéma relationnel des Variables s'avère cependant plus complexe que celui des Itérateurs. En plus des relations de raffinement, il faut en effet prendre en compte des relations de dépendance entre Variables. Considérant qu'une Variable est une fonction associant une valeur descriptive à un objet de la recherche, on parlera, par analogie à la composition entre fonctions, de composition de Variables. Nous avons jusqu'ici privilégié la réusabilité ainsi que l'efficacité des parties logicielles. Dans cette dernière étape de conception, la possibilité de composer les Variables offre enfin une expressivité remarquable au modèle de recherche locale. Nous mettons ici en évidence l'intérêt de la composition en examinant trois cas de figure fréquemment rencontrés : la mémorisation de la recherche, l'accélération du calcul et l'écriture de formules.

Mémorisation de la recherche. Un premier cas typique de composition concerne le cas de Variables qui enregistrent en mémoire la valeur prise par d'autres Variables. Remarquons à cet effet que les formes de fréquence ou de récence, utilisées notamment dans la méthode taboue, reposent rarement directement sur les Mouvements ou

Solutions du graphe de recherche, mais plutôt sur des Attributs ou des Caractéristiques qui leur sont associés. Ces Variables constituent donc des mémoires associatives dont l'indexation s'effectue à travers les valeurs prises par les Variables qu'elles composent.

Figure 3.8 – Exemples de composition pour des Variables mémorisantes

La figure 3.8 présentent deux exemples de ce type de composition. À gauche, l'Attribut `FrAdj<Dst>`, qui prélève les destructions d'adjacence opérées sur une tournée suite à l'exécution d'un Mouvement `Twex<Dst>`, est utilisé par la Caractéristique `Recency`, qui maintient à chaque nouvelle itération les dernières valeurs prises par `FrAdj<Dst>` selon le Mouvement effectué. À droite, la Variable `MinValue` retient la plus petite valeur rencontrée d'une Caractéristique particulière. Elle est ici employée dans le contexte d'un problème SAT en conjugaison avec la Caractéristique `SatVal` qui maintient le nombre de clauses satisfaites dans la solution courante ainsi que pour chaque clause, le nombre de littéraux vérifiés. La composante `MinValue<SatVal>` peut par exemple servir à la réalisation d'un critère d'aspiration pour une méthode taboue. À noter que la réalisation de `Recency` et de `MinValue` sont indépendantes des Représentations de problème car elles reposent uniquement sur des Variables. Comme nous l'illustrerons avec la librairie METALAB, ce type d'indépendance permet par exemple de programmer des listes taboues indépendamment du problème à résoudre. Seules les Variables composées servant à indexer la liste, comme FrAdj<Dst> dans le cas d'un PVC, dépendent du problème.

Accélération du calcul. Un deuxième cas typique de composition intervient lorsqu'on souhaite optimiser le calcul des Variables. Par exemple, si une Caractéristique

peut être réinitialisée à chaque itération, il est souvent bien plus efficace de mettre à jour sa valeur en fonction du Mouvement effectué. C'est pour cette raison que toute Caractéristique présente la fonction `track`, qui répercute l'effet du Mouvement sur son état interne. Dans le cas général, la fonction `track` peut prendre comme arguments le `Mouvement` effectué, la Solution courante, et/ou toute Variable pertinente. Dans l'autre sens, notons qu'il est aussi parfois envisageable d'accélérer la fonction `scan` d'un Attribut au moyen d'une information annexe maintenue par une Caractéristique.

Figure 3.9 – Exemples de composition entre Caractéristiques et Attributs

Dans l'exemple de la figure 3.9, à gauche, la fonction `track` de `TspVal` maintient la longueur de la tournée courante en fonction du gain de mouvement `TspGain`. À droite de la figure 3.9, l'Attribut `SatGain` peut calculer rapidement la différence entre le nombre de clauses nouvellement satisfaites et le nombre de clauses anciennement satisfaites en accèdant pour chaque clause au nombre de littéraux vérifiés que comptabilise la Caractéristique `SatVal`.

Écriture de formules. Le troisième cas typique de composition qu'on rencontre concerne l'élaboration de formules d'utilité complexes pour évaluer les Mouvements. L'Attribut `PenalizedTspGain` présenté à gauche de la figure 3.10 constitue un exemple pour une stratégie de diversification où l'on souhaiterait pénaliser des mouvements trop fréquemment utilisés. Dans ce cas, `PenalizedTspGain` calcule l'utilité d'un Mouvement en ajoutant à la valeur de l'Attribut `TspGain` une pénalité proportionelle aux valeurs que prennent les champs de `Frequency<FrAdj>` pour le Mouvement considéré. L'Attribut `Metroplis` à droite de la même figure programme quant à lui l'utilité d'un Mouvement au sens du recuit-simulé. Ayant tiré λ aléatoirement, il compose

la formule $e^{-\frac{\delta(m)}{T}-\lambda}$, vue au chapitre 2, en fonction d'une température maintenue dans un schéma de refroidissement géométrique GeoTemp et de l'Attribut TspGain. Par exemple, l'Attribut Metropolis<TspGain<Dst>,GeoTemp> pourrait être intégré à FirstImprovement afin de résoudre un PVC au moyen d'un algorithme de recuit-simulé. Notons que la composante GeoTemp est ici une Caractéristique puisqu'elle doit être mise à jour chaque fois qu'une nouvelle Solution est obtenue. Remarquons aussi que Metropolis est présenté comme un Attribut gabarit. Il suffirait donc de lui passer en argument un autre Attribut que TspGain pour l'appliquer à un autre problème. De même, il serait possible d'appliquer de nouveaux schémas de refroidissement en remplaçeant GeoTemp par une autre Caractéristique responsable de l'évolution de la température. Par composition, on pourrait même programmer un schéma de refroidissement qui tienne compte à son tour de l'état de la recherche par le biais de certaines Variables.

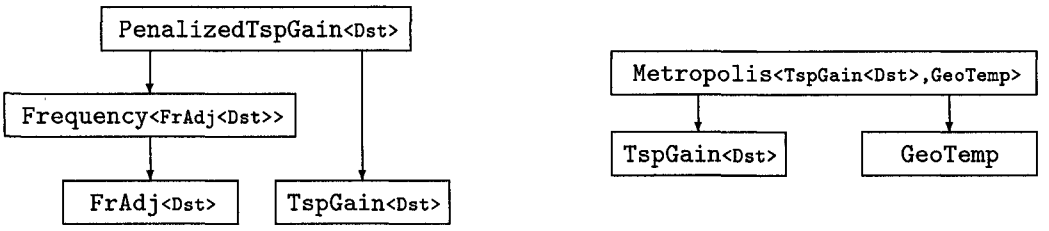


Figure 3.10 – Exemples de composition entre Caractéristiques et Attributs

3.6 Conclusion

Dans ce chapitre, nous avons proposé un modèle informatique pour décrire les méthodes de recherche locale. Nous avons pour cela prolongé l'approche suivie par les cadres d'application mais en remplaçant les patrons d'algorithmes par un ensemble de protocoles autour desquels se compose modulairement la recherche locale.

Pour cela, nous avons d'abord distingué trois catégories de composantes informatiques plutôt que de conjuguer directement les Représentations de problème au formes de Contrôle. Cette première étape, qui introduit les concepts de Variables, constitue une approche originale où la notion de coût de mouvement est par exemple mise au même plan conceptuel que les attributs de la méthode taboue. Elle constitue toutefois une grille de lecture très logique si l'on considère que la recherche locale est constituée d'une infrastructure (les Solutions et Mouvements) sur laquelle est prélevé un certain nombre d'informations (les Attributs et Caractéristiques) en fonction desquelles des mécanismes heuristiques (les Serveurs et Explorateurs) décident de la trajectoire à suivre.

Nous avons ensuite enrichi le modèle en raffinant progressivement cet ensemble de concepts. Nous avons ainsi fait la distinction entre les Variables observées autour de Solutions et celles obervées autour de Mouvements. Dans le modèle, un Attribut et une Caractéristique se distinguent par leur portée et non par l'objet de l'information qu'ils calculent. Nous avons également proposé de raffiner chaque concept en fonction des arguments que prennent leurs méthodes. Ce choix offre la perspective de générer l'ossature de chaque algorithme comme si leur patron avait été conçu sur mesure.

La mémorisation de la recherche, l'accélération du calcul et l'écriture de formules complexes ont enfin conduit à considérer la possibilité de composer des Variables entre elles. À noter que la composition entre Variables se généralise naturellement autour de la notion de *diagramme de composition.* Dans ce diagramme, les nœuds symbolisent les différentes composantes utilisées par un algorithme de recherche locale, et les arcs, les relations de dépendance qu'occasionnent leur composition. Cette notion est au centre du prochain chapitre. Nous y établirons en effet les règles qui coordonnent automatiquement la maintenance de chaque composante d'un algorithme en fonction du concept qu'elle modélise ainsi que de leur localisation dans un diagramme de composition. De cette manière, le développement d'une recherche locale

au moyen de la programmation générique se résume à coder pour chaque composante les fonctions qui calculent leur état interne sans se soucier si les informations qu'elles prennent en argument sont dans un état cohérent, et ce, *quel que soit le contexte où cette composante pourrait être intégrée ultérieurement.* L'orchestration de l'algorithme finalisé, où se décide à quel moment quelles composantes doivent être calculées, sera toujours du ressort de la librairie générique.

CHAPITRE 4 : INTERPRÉTATION DYNAMIQUE DU MODÈLE

Le modèle de recherche locale conçu au précédent chapitre présente un double potentiel. D'une part, en interprétant le modèle dynamiquement, il devient possible d'établir des mécanismes d'enchaînement qui coordonnent automatiquement l'entretien de chaque composante participant à une application donnée. D'autre part, en se conformant aux concepts mis en évidence par le modèle, il devient possible d'encadrer l'activité de programmation autour d'une méthodologie systématique. C'est en réalisant ces deux objectifs que l'utilitaire METALAB offre la possibilité de programmer chaque pièce logicielle de manière isolée et indépendamment de ses contextes d'utilisation ultérieurs. Dans ce chapitre, on propose d'automatiser la maintenance des différents concepts de la recherche locale en fonction des trois événements - initialisation, énumération et transition - autour desquels se déroule la recherche locale. Comme nous l'illustrons en début de chapitre, la maintenance automatique de la recherche ainsi mise au point permet de simplifier grandement l'assemblage des différents objets composés par l'usager.

4.1 Objectifs

Le code METALAB présenté à la figure 4.1 donne un aperçu des possibilités offertes par les mécanismes de maintenance automatique qu'on établit dans ce chapitre. Après avoir chargé un *PVC* à distance matricelle depuis un fichier d'entrée, le programme compose et initialise l'environnement de recherche relatif à une méthode de recuit-simulé. L'énumération du voisinage repose sur le Serveur `CircularTwxSvr` qui parcourt les coordonnées de Mouvement en circulant séquentiellement sur les arcs qui

sortiraient de la tournée. Le recuit-simulé est formé selon le schéma présenté à la figure 3.10. Notons que toutes les pièces logicielles et la manière dont elles interagissent est implicitement connu de METALAB par transitivité des types associés à travers les deux pièces logicielles que compose `FirstImprovement`. Cette connaissance est interprétée en temps de compilation par la librairie pour programmer la classe elle-même de l'objet `explorer`. Les fonctions `init`, permettant d'initialiser les paramètres de la recherche, la fonction `move`, permettant d'itérer la recherche, ainsi que les fonctions d'accés aux éléments de la recherche depuis le programme principal, sont toutes des fonctions automatiquement générées par le logiciel à la compilation (cf. Annexes A.5 et A.6 pour un aperçu). La programmation du recuit-simulé se borne en somme à réitérer la recherche jusqu'à ce qu'une température infinitésimale soit atteinte.

```
main()
{
  MtxDst dst("./att532");
  FirstImprovement<CircularTwxSvr,Metropolis<TspGain<MtxDst>,GeoTemp>> explorer;
  explorer.init<Tour<Dst>,GeoTemp>(dst, random, 1000, 0.999);
  do { explorer.move(); } while (explorer<GeoTemp>().value > 0.0001);
};
```

Figure 4.1 – Exemples de composition finale d'un algorithme de recherche

Le programme composé manuellement à la figure 4.2 déroule une recherche exactement équivalente au code précédent. Outre les qualités évidentes de concision du code généré par METALAB, remarquons que les principaux intérêts liés à la maintenance automatique de la recherche sont la sécurité et la modularité. D'une part en effet, l'utilisation de METALAB prévient tout oubli de mise à jour ou toute modification malencontreuse des objets de la recherche à l'intérieur des boucles de recherche. D'autre part, si par exemple on souhaitait modifier le schéma de refroidissement, l'usager de METALAB n'aurait qu'à passer une nouvelle composante en argument à la place de `GeoTemp` alors que manuellement, il lui faudrait vérifier en particulier si l'appel à `track` prend les mêmes arguments avec la nouvelle composante, ce qui exige

```
main()
{
  MtxDst dst("./att532");
  Tour tour(dst, random);
  Twex<MtxDst> twex;
  CircularTwxSvr<MtxDst> server;
  TspGain<MtxDst> gain;
  GeoTemp temperature(1000,0.999);
  Metropolis<TspGain<MtxDst>, GeoTemp> criteria;

  do {
     server.begin(twex, solution);
     gain.scan(twex);
     criteria.scan(gain, temperature);
     while (criteria.value <= 0 && server.next(twex, solution))
     {
        gain.scan(twex);
        criteria.scan(gain, temperature);
     }
     if (criteria.value > 0)
     {
        temperature.track();
        twex.move(solution);
     }
  }
  while (temperature.value > 0.0001);
};
```

Figure 4.2 – Composition explicite du même algorithme

de rentrer dans le programme. À l'exécution, les deux programmes ont théoriquement les mêmes propriétés de temps de calcul. En pratique, comme l'illustre le tableau 4.1 pour des problèmes allant de 101 à 2392 nœuds, le programme METALAB précédent reste en moyenne 30% plus lent que la version explicite. Les temps mentionnés aux deux dernière lignes du tableau sont comptabilisés en millisecondes sur un total de 1000 recherches locales partant de solutions initiales générées aléatoirement.

Tableau 4.1 – Comparaison des temps de calcul

Problème et taille	eil101	ch150	gr202	pcb442	gr666	pr1002	pr2392
Temps METALAB	3669	3761	3438	3635	3380	3395	3535
Temps explicite	2495	2561	2672	2756	2951	3115	3102

4.2 Usage des diagrammes de composition

Sur le plan technique, l'assemblage et la maintenance automatique de la recherche procède en parcourant des diagrammes de composition qui schématisent les dépendances entre composantes devant être mises à jour. La maintenance de l'ensemble des composantes ne s'effectuant pas de la même façon lors de l'initialisation de la recherche, lors de l'énumération d'un nouveau mouvement, et lors de la transition vers une nouvelle solution, elle s'appuie en fait sur trois diagrammes de composition différents : le diagramme d'initialisation, le diagramme d'énumération et le diagramme de transition. Dans les trois sections suivantes, on propose d'établir d'une part les règles de composition et d'autre part les mécanismes d'enchaînement relatifs à chacun de ces diagrammes et à partir desquels est propagé l'état de la recherche.

Règles de composition. Ces règles spécifient *quelles* composantes peuvent intégrer *quel* diagramme de composition et dans *quel* état. Les règles de composition se formulent en terme de composabilité et de norme de composition. Dans le premier cas, on détermine pour chaque fonction conceptuelle de la recherche locale (pour une Caractéristique : son constructeur et la méthode `track` ; pour un Serveur : son constructeur, les méthodes `begin` et `next`, ...) quelles catégories de composantes peuvent être prises en argument. Dans le deuxième cas, il faut spécifier dans quel état se situe chaque composante prise en argument au moment de l'appel d'une telle fonction. En pratique, les règles de composition contraignent le programmeur à suivre une certaine logique dans son activité de développement. Elle lui assure en contre-partie la compatibilité de ses pièces logicielles quel que soit leur emploi potentiel.

Mécanismes d'enchaînement. Ces mécanismes déterminent *quand* l'utilitaire met à jour les composantes de l'application générée afin de respecter les normes de composition portant sur leur état. En pratique, l'enchaînement s'appuie sur un parcours

dans les diagrammes de composition et constituent le moteur de l'utilitaire. Modifier les mécanismes d'enchaînement ne change pas le résultat de la recherche locale obtenue mais selon le contexte peut par contre améliorer ou détériorer significativement les temps de calcul. Les normes de composition étant définies en amont, on peut définir dans une librairie plusieurs versions d'enchaînement sans affecter la compatibilité des pièces logicielles existantes.

4.3 Phase d'initialisation

Scénario. Au cours de la phase d'initialisation, tous les objets de la recherche sont construits en fonction des données du problème, d'autres objets de la recherche et/ou de paramètres de réglage. Les fonctions conceptuelles initialisantes sont les constructeurs C++ de chaque composante intégrée dans l'application. Au départ, la Solution est soit initialisée selon un algorithme déclenché par son constructeur, soit passée en argument et récupérée par la recherche si elle a été calculée par ailleurs. Les Caractéristiques intégrées dans la recherche sont ensuite progressivement initialisées autour de cette Solution. Pour ce qui a trait au Serveur, au Mouvement et aux Attributs, les constructeurs sont seulement conçus pour formater l'objet construit en mémoire mais non pour initialiser ses champs puisque son état n'a pas encore de signification. Le tableau 4.2 résume les règles de composition suivantes.

Composabilité. On considère que le formatage du Serveur, du Mouvement et des Attributs ne dépend pas de l'état de la recherche. Leurs constructeurs ne peuvent donc composer aucun objet endogène de la recherche. Ils peuvent par contre prendre en argument les données du problème et/ou des paramètres. Un constructeur de Solution peut prendre en argument les données du problèmes et/ou des paramètres de construction comme par exemple la version d'algorithme qui calcule la Solution

Tableau 4.2 – Résumé des règles de composition en phase d'initialisation

Constructeur...	Composables endogènes (en phase avec la Solution initiale)	Composables exogènes
... de Solution	Aucun	Problème, paramètres
... de Caractéristique	Solution, Caractéristiques	Problème, paramètres
... des autres concepts	Aucun	Problème, paramètres

initiale. Les constructeurs des Caractéristiques peuvent composer la Solution et/ou d'autres Caractéristiques. Elles peuvent prendre également en argument des paramètres ainsi que les données du problème. Solution et Caractéristiques sont les concepts qui dessinent le diagramme d'initialisation. Les sommets correspondent à la Solution et aux Caractéristiques utilisées par la recherche. Les arcs relient les Caractéristiques à la Solution et/ou aux Caractéristiques que leurs constructeurs prennent en argument.

Normes de composition. Lorsqu'une Caractéristique est initialisée, la Solution et/ou les Caractéristiques prises en argument par son constructeur ont été au préalable initialisées. Pour respecter cette règle, le diagramme d'initialisation doit être acyclique.

Mécanisme d'enchaînement. Le diagramme d'initialisation étant acyclique, les mécanismes d'enchaînement consistent simplement à construire successivement la Solution et l'ensemble des Caractéristiques selon un ordre topologique ascendant, en remontant des feuilles jusqu'aux racines du diagramme. Mentionnons que l'algorithme qui implante cet enchaînement est statique, c'est-à-dire que le calcul de l'ordre de construction des objets en fonction d'un quelconque diagramme d'initialisation est effectué dès la compilation du programme sans aucune surchage en temps d'exécution. L'exemple présenté à la figure 4.3 montre dans quel ordre METALAB construirait les objets d'une recherche taboue avec critère d'aspiration pour résoudre un PVC. La Caractéristique `NstNgbTwxHshTbl` qui est introduite est une table de hachage dans

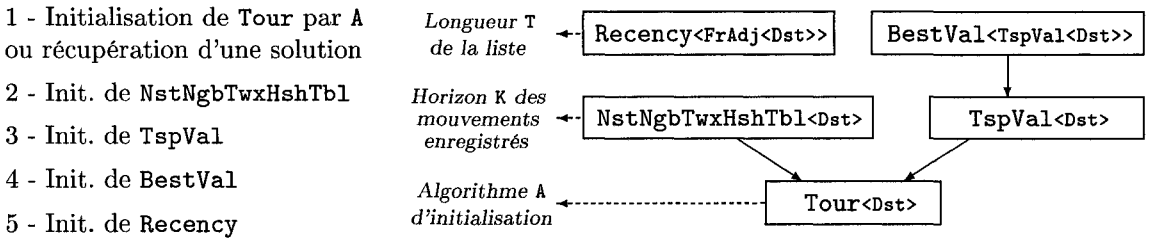


Figure 4.3 – Enchaînement d'un diagramme d'initialisation

laquelle sont rangés selon leur gain tous les Mouvements `Twex` du voisinage courant dont un des deux sommets d'un des deux arcs entrants est un des `K` plus proches voisins de son nouveau sommet relié. Cette structure de donnée qui accélère significativement la fouille du voisinage sera étudiée en détail au chapitre 7. La paramétrisation des composantes (ici, `NstNgbTwxHshTbl` par l'horizon `K`, `Recency<FrAdj>` par la longueur `T` et `Tour` par la version `A` de l'algorithme de construction) sera abordée dans le prochain chapitre puisqu'elle n'interfère pas avec les règles de composition et d'enchaînement. Signalons que seuls les constructeurs peuvent prendre en argument des paramètres : toutes les autres fonctions, intervenant en phase énumérative et/ou transitoire, ne peuvent prendre en argument que des objets de la recherche. Notons que les contructeurs du Serveur, du Mouvement et des Attributs peuvent enfin être appelés dans n'importe quel ordre puisqu'ils fonctionnent chacun de manière isolée.

4.4 Phase d'énumération

La mise en place d'un mécanisme de maintenance en phase d'énumération est une tâche délicate. En effet, cette phase est succeptible d'occuper une large part du temps de calcul où le mécanisme d'enchaînement est exécuté un grand nombre de fois. Les fonctions `scan` peuvent de plus correspondre à d'infimes opérations à côté desquelles le calcul de l'ordre d'enchaînement lui-même n'est pas forcément négligeable.

Scénario. Au cours de la phase d'énumération, qui est orchestrée par la fonction `find` d'un Explorateur, le Serveur et l'Explorateur collaborent pour sélectionner un Mouvement. Ces deux composantes œuvrent pour cela sur un même objet Mouvement qu'on appelle le Mouvement actif. Le Serveur initialise les coordonnées du Mouvement actif au moyen de sa méthode `begin` à l'entrée dans la fonction `find`. L'Explorateur peut alors lui demander de modifier le Mouvement actif au moyen de la méthode `next` afin d'énumérer le voisinage. L'appel à `next` retourne un booléen indiquant la validité de l'opération. En cas d'échec, le Mouvement actif est dans un état invalide et le Serveur n'est plus opérationnel. Pour sélectionner le Mouvement à appliquer, la fonction `find` peut examiner tout objet de la recherche que compose l'Explorateur. Aux moments jugés opportuns, l'Explorateur peut déclencher la mémorisation du Mouvement actif dans des structures tampons proposées par la librairie. Lors d'une telle mémorisation, tous les `Attributs` déjà calculés sur le Mouvement actif sont parallèlement copiés dans le tampon. Lorsque l'Explorateur récupère un Mouvement enregistré dans un tampon, ce dernier devient le Mouvement actif et les Attributs auparavant déjà calculés sont récupérés de sorte que l'état de l'exploration du voisinage est restitué tel quel au moment de l'enregistrement. Un booléen est retourné indiquant un succés ou un échec de la phase d'énumération. En cas de réussite, le Mouvement actif est appliqué. L'Explorateur `FirstImprovement` donné en exemple au chapitre précédent constitue une réalisation simple du scénario d'énumération.

Composabilité. Les deux méthodes du Serveur doivent prendre chacune les mêmes arguments parmi, les données du problème, la Solution courante, le Mouvement actif, et/ou, des Caractéristiques. La fonction `find` ne peut quant à elle composer que des Attributs et/ou des Caractéristiques. Cette contrainte encourage l'écriture d'Explorateurs génériques qui peuvent compiler quelles que soient les formes de Représentation du graphe de recherche. En plus de l'Explorateur qui correspond à sa racine, les sommets du diagramme d'énumération correspondent aux Attributs utilisés directement par

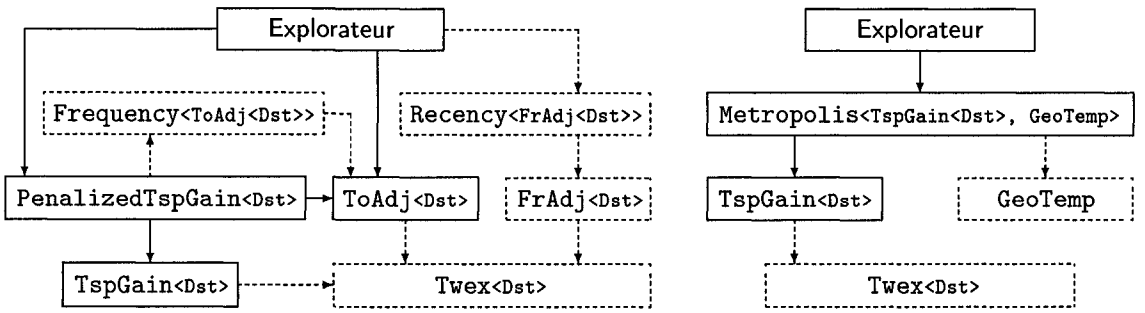


Figure 4.4 – Exemples de diagrammes d'énumération

l'Explorateur ou indirectement à travers d'autres Attributs. Chaque arête Explorateur → Attribut du diagramme correspond à l'utilisation d'un Attribut dans la fonction `find` et chaque arête Attribut → Attribut à la composition d'un Attribut dans la fonction `scan` d'un autre Attribut appartenant déjà au diagramme. La figure 4.4 présente deux diagrammes d'énumération pour la résolution du PVC. Le diagramme de gauche correspond à une meilleure amélioration sur le gain avec pénalité proportionnelle à la fréquence de l'Attribut `ToAdj` parmi les Mouvements déjà effectués et avec critère tabou sur la création d'une adjacence récemment détruite. Le diagramme de droite correspond au recuit-simulé présenté en fin de chapitre précédent. L'ensemble des relations de composition a été reporté sur le schéma mais seuls les traits pleins correspondent aux diagrammes d'énumération. Notons que les Attributs qui sont utilisés seulement en phase de transition pour incrémenter des Caractéristiques ne doivent pas être considérés. L'Attribut `FrAdj` ne fait ainsi pas partie du diagramme d'énumération de la méthode taboue. Mentionnons aussi que pour les deux algorithmes, l'Attribut `TspGain` n'est pas directement examiné dans l'Explorateur mais fait partie du diagramme par transitivité depuis l'Attribut, `PenalizedTspGain` ou `Metropolis`, qui le compose depuis l'Explorateur. À gauche, l'Attribut `ToAdj` participe au diagramme à deux titres, il est directement utilisé par l'Explorateur pour vérifier s'il est intercepté par `Recency<FrAdj>`, et indirectement utilisé par le biais de `PenalizedGain` pour calculer l'utilité du Mouvement.

Normes de composition. La Solution et les Caractéristiques composées directement ou indirectement par l'Explorateur et/ou le Serveur sont des objets constants. Au moment où l'Explorateur examine un Attribut, celui-ci doit avoir été mis en phase avec le Mouvement actif au moyen de la méthode `scan`. Si une méthode `scan` compose elle-même d'autres Attributs, ceux-ci doivent à leur tour avoir été mis en phase avec le Mouvement actif avant d'être utilisés. Pour respecter cette norme, on impose que le diagramme d'énumération soit acyclique. Le tableau 4.3 résume les règles de composition en phase d'énumération.

Tableau 4.3 – Résumé des règles de composition en phase d'énumération

Fonction conceptuelle	Composables endogènes (en phase avec le Mouvement actif et la Solution courante)	Composables exogènes
`next` et `begin`	Mouvement (à modifier), Solution, Caractéristiques	Problème
`find`	Caractéristiques, Attributs	Problème
`scan`	Mouvement, Solution, Caractéristiques, Attributs	Problème

Cadre statique. Une première approche d'enchaînement possible est de s'appuyer uniquement sur la connaissance en temps de compilation de la structure du diagramme d'énumération. À chaque appel de `next`, les fonctions `scan` de tous les Attributs du diagramme sont exécutées selon un ordre topologique ascendant. Cette approche a le mérite d'ordonner le calcul sans aucune surchage en temps d'exécution. Elle convient particulièrement aux applications dont les Attributs effectuent des calculs minimes. Ce cadre d'enchaînement est par exemple idéal pour le programme présenté à la figure 4.1. Il génère un code exactement équivalent à celui déroulé manuellement à la figure 4.2.

Cadre dynamique. Dans la version dynamique, on associe à chaque Attribut un indicateur booléen de maintenance. Lorsqu'un Attribut est examiné soit par l'Explorateur,

soit à l'intérieur d'une fonction `scan`, on vérifie l'indicateur de maintenance et le cas échéant on met à jour l'Attribut composé. L'intérêt de la version dynamique est qu'elle permet d'éviter le calcul des Attributs qui ne sont pas systématiquement utilisés. Dans le cas où deux Attributs (un rapide mais approximatif et un lent mais précis) sont utilisés par l'Explorateur pour évaluer le gain des Mouvements, le calcul de l'Attribut précis peut être ainsi seulement déclenché après considération du calcul préalable de l'Attribut approximatif. L'approche dynamique est également adéquate dans le cas où plusieurs traversées de voisinage sont effectuées, une première qui enregistre certains Mouvements en mémoire à l'examen d'un premier Attribut, et une seconde qui retraverse le registre et sélectionne le Mouvement à appliquer à l'examen d'un second Attribut. Pour ces deux exemples, le calcul systématique de tous les Attributs serait malencontreusement déclenché dans le cadre statique. Par contre, l'enchaînement dynamique peut engendrer une surcharge de calcul non négligeable car il faut systématiquement vérifier l'état de maintenance.

```
template <class SVR, class GAIN> struct FirstImprovement : FindWith<SVR, If<GAIN>>
{
  bool find(If<GAIN>& gain)
  {
     while (gain().value <= 0 && next());
     return (gain.value > 0);
  }
};
```

Figure 4.5 – Version dynamique de l'Explorateur `FirstImprovement`

Composition statique et dynamique d'Attributs. Afin de tirer au mieux parti de chaque situation, la librairie METALAB propose un mécanisme d'enchaînement programmable. Pour cela, lorsque l'usager définit un Explorateur ou un Attribut, il indique pour chaque Attribut composé si le calcul doit en être déclenché statiquement avant de poursuivre les calculs ou dynamiquement au besoin. À titre d'illustration, la composante `FirstImprovement`, reprogrammé à la figure 4.5, signale par exemple

que l'Attribut `GAIN` ne doit être calculé que sur demande car il est préfixé du mot-clé `If`. Dans ce cas, l'Attribut est passé en référence non constante dans la fonction `find` et chaque fois que l'expression `gain()` est rencontrée, l'Attribut est mis à jour si son indicateur de maintenance l'exige. Notons que le deuxième accès à `gain` n'utilise pas l'expression `gain()` car on est sûr à ce point du code que l'Attribut a déjà été calculé. Une arête du diagramme d'énumération correspondant à une composition dynamique signalée par `If` est appelée arête dynamique. Dans le cas contraire, qui est le mode de composition par défaut, on dit que l'arête est statique. Notons qu'un même Attribut peut être à la fois pointé statiquement et dynamiquement. Le mécanisme d'enchaînement qu'on propose est programmé en fonction du diagramme d'énumération, du statut de chaque arête et des indicateurs de maintenance.

Points d'enchaînement. Un point d'enchaînement correspond à un point du programme où doit être calculé un ordre de maintenance. L'appel de `next` dans `find` se termine par exemple par un point d'enchaînement car avant de retourner dans la fonction `find`, il faut déterminer quels Attributs doivent être maintenus statiquement et dans quel ordre. Les seuls autres points d'enchaînement correspondent au déclenchement explicite du calcul d'un Attribut passé en argument dynamique, par exemple lorsqu'est rencontrée l'expression `gain().value` à la figure 4.3. Notons qu'un point d'enchaînement explicite peut intervenir dans la fonction `find` mais aussi dans une fonction `scan` composant dynamiquement un autre Attribut.

Sous-diagramme et diagramme résiduel. À chaque point d'enchaînement est défini un sous-diagramme et un diagramme résiduel comme suit. Lorsque le point d'enchaînement est explicite, la racine du sous-diagramme est l'Attribut depuis lequel est déclenché la maintenance. Lorsque le point d'enchaînement est celui à la sortie de `next`, la racine du sous-diagramme est l'Explorateur. Le sous-diagramme est alors défini comme le sous-graphe inscrit dans le diagramme d'énumération et partant de la racine. On définit ensuite les arêtes désactivantes comme les arêtes dynamiques

et/ou pointant sur un Attribut déjà calculé par un point d'enchaînement précédent. On définit également les chemins désactivés comme les chemins commençant par une arête désactivante et terminant soit sur une feuille soit sur un Attribut pointé par d'autres chemins sur le sous-diagramme. Le diagramme résiduel est enfin défini en enlevant les arcs de tous les chemins désactivés. Chaque fois qu'un sommet ou un groupe de sommets se retouve déconnecté de la racine du sous-diagramme suite à la suppression de telles arêtes, ce sommet ou ce groupe de sommets est également supprimé.

Mécanisme d'enchaînement. Chaque fois qu'un point d'enchaînement est rencontré, les Attributs du diagramme résiduel sont calculés dans l'ordre topologique ascendant. Notons que le cadre de maintenance statique (resp. dynamique) présenté en préalable correspond au cas où toutes les compositions sont statiques (resp. dynamiques). Malgré le calcul du graphe résiduel que nécessite cette approche flexible, le mécanisme d'enchaînement peut en fait être implanté sans qu'il engendre pratiquement aucune surcharge de calcul en temps d'exécution. Pour cela, on s'appuie sur le fait qu'un diagramme d'énumération comporte en réalité peu d'Attributs. Étant donné ce nombre restreint, on peut d'une part aggréger l'ensemble des booléens indicateurs de maintenance dans un seul nombre entier considéré en base 2. À l'appel de `next`, l'entier est initialialisé à 0, ce qui signifie qu'aucun Attribut n'est en phase. D'autre part, la comparaison entre la structure du sous-diagramme et cet entier peut être faite dès la compilation pour l'ensemble des cas de figure, c'est-à-dire pour l'ensemble des sous-diagrammes et des valeurs d'entier possibles. Pour les diagrammes de la figure 4.4, on obtient par exemple 3 sous-diagrammes possibles et 2^3 valeurs de maintenance, ce qui exige 24 calculs de diagrammes résiduels à la compilation. Dans le contexte, déjà très rare, où un diagramme d'énumération comporterait dix Attributs, la phase de compilation énumèrerait dans le pire cas $10 \times 2^{10} = 10240$ graphes de 10 sommets au plus, ce qui est encore acceptable. En énumérant explicitement tous les diagrammes résiduels possibles à la compilation, le calcul de l'ordre

de maintenance en temps d'exécution se résume pour chaque point d'enchaînement à un branchement (en C++, un `switch`) sur l'entier et à sa mise à jour après maintenance. Le mécanisme d'enchaînement est donc programmable mais ne génère en pratique aucune surcharge de calcul par rapport au véritable travail des fonctions `scan`.

4.5 Phase de transition

Scénario. Lors de la phase de transition, les fonctions `track` des Caractéristiques mettent d'abord à jour leur état interne, généralement au moyen d'Attributs pris en argument. La fonction `move` du Mouvement actif met ensuite à jour la Solution courante. Notons que la mise à jour en phase de transition s'effectue avec la logique inverse des phases d'initialisation et d'énumération : chaque étape de maintenance s'appuie sur des informations associées à la Solution non encore mise à jour. Cette logique a été retenue car dans la phase de transition, on renouvelle l'information en dérivant l'état de la recherche associée à l'itération précédente. Le tableau 4.4 résume les règles de composition en phase de transition.

Tableau 4.4 – Résumé des règles de composition en phase de transition

Fonction conceptuelle	Composables endogènes (en phase avec le Mouvement actif et la Solution avant transformation)	Composables exogènes
`scan`	Mouvement, Solution, Caractéristiques, Attributs	Problème
`track`	Mouvement, Solution, Caractéristiques, Attributs	Problème
`move`	Solution (à modifier), Attributs	Problème

Composabilité. Les fonctions `track` peuvent prendre en argument la Solution, le Mouvement actif, d'autres Caractéristiques et/ou des Attributs. La fonction `move` doit

prendre en argument la Solution et éventuellement des Attributs. Les deux fonctions peuvent prendre également en argument le Problème à résoudre. Une raison pour laquelle on interdit à la fonction `move` de prendre en argument des Caractéristiques est que le corps de cette fonction constitue un point de basculement de l'algorithme entre l'ancienne Solution et la nouvelle alors que les Caractéristiques sont attachées à un point fixe de la trajectoire de recherche. À noter que pour des raisons similaires, les fonctions `begin` et `next` ne pouvaient pas composer d'Attributs. Le mécanisme de composition dynamique n'est pas défini en phase de transition ; les Attributs ne peuvent donc être composés par les Caractéristiques que de manière statique.

Normes de composition. L'état d'une Caractéristique ou de la Solution composée pas une fonction `track` est celui avant transition. L'état d'un Attribut composé par `track` ou `move` doit être en phase avec le Mouvement actif. Il est ici utile de définir deux diagrammes de transition distincts, un pour les Attributs et l'autre pour les Caractéristiques. Les Attributs utilisés en phase de transition directement ou indirectement par le biais d'autres Attributs forment les sommets du diagramme de transition des Attributs. Les arcs correspondent à la composition d'un Attribut par un autre Attribut. Le diagramme de transition des Caractéristiques est défini par l'ensemble des compositions faites entre Caractéristiques de la recherche par les fonctions `track`. Les normes de composition contraignent ces deux diagrammes à être acycliques.

Mécanisme d'enchaînement. Les Attributs non calculés en phase d'énumération mais utilisés en phase de transition doivent être préalablement calculés. L'enchaînement des mises à jour est déterminé en comparant l'entier indicateur de maintenance utilisé en phase d'énumération avec le diagramme de transition des Attributs. Sachant que toutes les compositions sont ici statiques, on met à jour les Attributs selon un ordre topologique ascendant sur le diagramme résiduel obtenu. La mise à jour des Caractéristiques est alors enchaînée statiquement dans l'ordre topologique descendant sur le diagramme de transition des Caractéristiques.

4.6 Conclusion

Dans ce chapitre, nous avons établi les mécanismes d'enchaînement qui permettent de dérouler automatiquement la recherche en fonction des composantes entrant en interaction. Ces mécanismes, qui constituent le moteur de la librairie METALAB présentée au prochain chapitre, permettent de simplifier la composition des algorithmes à partir des différents objets de recherche utilisés par le programmeur. Un survol des techniques de programmation génériques qui sont utilisées afin d'implanter ces mécanismes est proposé en annexes A.5 et A.6.

En phase d'initialisation et de transition, la maintenance automatique a été réalisée au moyen de parcours statiques dans les diagrammes d'initialisation et de transition. Cette approche est efficace car elle ne génère aucune surcharge de calcul en temps d'exécution. Elle repose par contre sur l'hypothèse qu'un objet composé est systématiquement utilisé : lorsque cet objet n'est en fait pas utilisé, il est quand même calculé.

Comme en phase d'énumération il arrive fréquemment que des objets composés ne soient pas systématiquement utilisés, nous avons proposé un mécanisme d'enchaînement programmable. Lorsqu'un Attribut est composé dynamiquement au moyen du mot-clé `If`, son calcul est ainsi retardé à son utilisation effective. Dans ce cas, la maintenance des composantes s'effectue couche par couche à chaque point d'enchaînement. Par défaut, le mode de composition d'un Attribut est statique, c'est-à-dire, similaire aux phases d'initialisation et de transition. Lorsque tous les Attributs sont composés statiquement le mécanisme d'enchaînement coïncide avec un parcours topologique dans le diagramme d'énumération.

Troisième partie

Études expérimentales

La troisième partie de l'ouvrage, expérimentale, se consacre à valider le modèle de recherche locale établi en partie théorique. Cet objectif comporte trois volets. Dans un premier temps, nous présentons le logiciel METALAB qui réalise les mécanismes de maintenance automatique et donne un cadre de réalisation pour chaque concept de la recherche locale dans le cadre du C++. Ce tutoriel fait l'objet du chapitre 5. Dans un deuxième temps, un catalogue de problèmes résolvables au moyen de la librairie METALAB est passé en revue dans le chapitre 6. Dans un troisième temps, nous illustrons l'intérêt d'utiliser METALAB pour la conduite de projets avancés qui nécessitent des composantes de recherche sophistiquées. À ce titre, une étude de cas, où sont développées de nouvelles méthodes avec critère tabou pour le voyageur de commerce, est présentée dans le chapitre 7.

Dans le chapitre 5, la présentation du logiciel METALAB s'appuie sur les cadres de résolution relatifs au voyageur de commerce et au problème SAT. La mise en parallèle de ces deux contextes permet d'illustrer la librairie dans le cadre de l'optimisation ainsi que de la satisfaction de contraintes. Le chapitre montre également comment réaliser des listes taboues génériques ou des critères d'acceptation comme ceux utilisés dans le recuit-simulé au moyen de Variables.

Dans le chapitre 6, nous commençons par montrer comment étendre les composantes utilisées dans le chapitre 5 au cas du voyageur de commerce où le coût des arêtes

dépend du temps. Des composantes de résolution pour le problème de la p-médiane, le problème de coloration de graphe, et enfin, le problème d'affectation quadratique sont ensuite présentées.

Dans le chapitre 7, le comportement de la méthode taboue pour le voyageur de commerce est étudié au moyen de Serveurs permettant d'accélérer très significativement l'énumération du voisinage. Ces expérimentations montrent clairement que le diamètre des sous-séquences qu'inversent les 2-*opts* choisis par la méthode taboue reste très petit comparativement au nombre de villes. Une technique de recherche adaptée aux phases d'intensification est alors développée. À cet effet, seuls les mouvements de petit diamètre ou seuls les mouvements faisant rentrer une arête dont une des deux extrémités est un des K plus proches voisins de l'autre extrémité sont examinés. Une technique d'anticipation est alors utilisée pour renforcer localement la couverture du graphe de recherche. Pour cela, on propose de sélectionner à chaque itération le mouvement qui se situe sur le chemin conduisant à la meilleure tournée accessible depuis la solution courante et en un nombre fixé de mouvements.

CHAPITRE 5 : PRÉSENTATION DU LOGICIEL METALAB

Nous donnons dans ce chapitre la liste des spécifications C++ de chaque concept utilisé dans METALAB. La programmation des composantes participant à la réalisation d'une méthode taboue ainsi que d'un recuit-simulé pour résoudre les problèmes *SAT* et *PVC* illustre l'exposé. Des composantes de résolution plus efficaces seront proposées dans le chapitre 7.

5.1 Concept de Problème

Pour une application donnée, un programme de recherche locale s'exerce en fonction de données paramétriques qui caractérisent l'instance à solutionner. Le concept de Problème impose de regrouper dans une même classe l'ensemble de ces données paramétriques. Un Problème ne présente aucune fonction conceptuelle. En pratique, il est souvent constitué d'une interface d'accès facilitant la consultation des données et d'un constructeur permettant de charger l'instance dans le programme. Dans METALAB, un Problème est toujours passé en argument comme référence constante.

La classe `SAT` présentée à la figure 5.1 réalise le Problème *SAT*. Elle est constituée d'un tableau `var` de `N Variable` et d'un tableau `cls` de `M Clause`. Les variables (resp. les clauses) sont indexées par `i` (resp. par `j`) de sorte que `var[i].j[k]` désigne par exemple la $\texttt{k}^{\text{ième}}$ clause où apparaît la variable d'indice `i`. La classe `SAT` définit par ailleurs un constructeur permettant de charger le Problème depuis un fichier d'entrées. L'implantation proposée se limite ici aux versions de 3-*SAT*.

```
struct SAT
{ int N, M, W;                    Nombre N de variables, nombre M de clauses et
                                  nombre maximal W d'apparition d'une même variable
  struct Variable
  { int m;                        Nombre de clauses où apparaît la variable
    int *j;                       j[0] ... j[m-1] : indices des clauses contenant la variable
    int *r;                       r[k] : rang de la variable dans la clause j[k]
    bool *s;                      s[k] : signe de la variable dans la clause j[k]
  } *var;

  struct Clause
  { int i[3];                     i[r] : indice de la variable qui apparaît dans le r^ième littéral
    bool s[3];                    s[r] : signe du r^ième littéral
  } *cls;

  SAT(const std::string&);        Chargement d'un problème depuis un fichier d'entrées
};
```

Figure 5.1 – Programmation du Problème SAT

5.2 Concept de Solution

On représente dans une Solution l'état d'une réponse pour le Problème à résoudre. La seule fonction conceptuelle d'une Solution est son constructeur qui sert à initialiser le point de départ de la recherche si aucune Solution déjà calculée n'est fournie à l'algorithme. Ce constructeur prend en argument le Problème à traiter suivi éventuellement d'une liste de paramètres. Le Problème est transmis en référence constante et les paramètres sont copiés par valeur.

La classe `Assignment` présentée à la figure 5.2 constitue un exemple de Solution pour un Problème SAT. Comme illustré, on doit indiquer à la librairie le type de Problème traité suivi des types des paramètres de construction avec le mot-clé `BuildWith`. La classe `Assignment` dérive ainsi de `BuildWith<SAT,int>`. La liste des types d'arguments indiqués dans `BuildWith` doit suivre exactement le même ordre que les arguments dans le constructeur.

```
struct Assignment : BuildWith<SAT, int>
{ Array<bool> x;   x[i] : valeur affectée à la variable i

  Le constructeur initialise chaque variable à false, true ou aléatoirement
  Assignment(const SAT& pbm, int mode) : x(pbm.N)
  {
    switch (mode)
    {
      case 0  : for (short i=0; i<pbm.N; i++) x[i] = false;
      case 1  : for (short i=0; i<pbm.N; i++) x[i] = true;
      default : for (short i=0; i<pbm.N; i++) x[i] = rand() % 2;
    }
  }
};
```

Figure 5.2 – Programmation de la Solution `Assignment`

Pour des situations plus complexes que l'exemple d'`Assignment`, il est parfois pratique qu'une Solution définisse des transformations internes à partir desquelles sont programmées les Mouvements. Cette technique permet de factoriser l'écriture de sections de code fastidieuses. Par exemple, Fredman, Johnson, McGeoch et Ostheimer (1995) notent que pour le PVC, l'ensemble des Mouvements *r-opt* peut être programmé en fonction des opérations `flip` (renversement d'une sous-séquence), `between` (test sur la présence d'un élément dans une sous-séquence), `next` et `pred` (élément suivant et précédent d'un élément dans la tournée).

La réalisation de la Solution `Tour` pour le PVC présentée à la figure 5.3 suit cette logique. Dans `Tour`, une tournée est représentée par une permutation encodée au moyen des tableaux `city` et `rank` où `city[k]` désigne la $k^{\text{ième}}$ ville de la tournée et `rank[c]` la position de la ville d'indice `c` dans la tournée. Notons que le constructeur de `Tour` prend ici seulement en argument les données du problème, une structure de distance `Dst` à travers laquelle on accéde au nombre N de sommets du PVC à résoudre. La composante `Tour` dérive à ce titre de `BuildWith<Dst>`. Notons que la classe `Tour` est définie en fonction du type abstrait `Dst`. Comme dans le cas de l'Attribut `TspGain` présenté au préalable au chapitre 3, cette technique permet de

```
template <class Dst> struct Tour : BuildWith<Dst>
{
  const int N;
  Array<short> city, rank;

  Tour(const Dst& dst) : N(dst.N), city(dst.N), rank(dst.N)
  {
  } for (short c=0; c<dst.N; c++) city[rank[c] = c] = c;

  short succ(short c) const { return city[(rank[c]==N-1) ? 0 : rank[c]+1]; }
  short pred(short c) const { return city[(rank[c]==0) ? N-1 : rank[c]-1]; }
  short succ(short c, short k) const {
  } return city[(rank[c]+k<N) ? rank[c]+k : rank[c]+k-N];

  bool between(short c1, short c2, short c3) const { ... }

  void flip(short c1, short c2)
  {
     short l = rank[c1], r = rank[c2];
     short ray = (r>l) ? (r-l+1)/2 : (r-l+N+1)/2;

     while (ray-- > 0)
     {
        short city_r = city[r];
        rank[city[r] = city[l]] = r--;
        rank[city[l] = city_r ] = l++;
        if (r < 0) { r = N-1; } else if (l == N) { l = 0; }
     }
  }
};
```

Figure 5.3 – Programmation de la Solution Tour

travailler indifféremment sur une variété de structures de distance. À part le constructeur et la méthode `flip` qui opèrent en $O(N)$, toutes les opérations s'effectuent en temps constant. Toujours selon Fredman, Johnson, McGeoch et Ostheimer (1995), la représentation d'une tournée par tableau n'est en pratique dominée par les représentations par arbres qu'à partir de problèmes contenant quelques milliers de sommets. Nous verrons dans le chapitre 7 que dans le cas de la méthode taboue basée sur un 2-*opt*, la représentation par tableau reste en fait encore très efficace au delà de cette limite.

5.3 Concept de Caractéristique

Une Caractéristique est constituée d'une méthode `track`, d'un constructeur et d'un certain nombre de champs. La méthode `track` calcule une information qui doit être mise à jour chaque fois qu'un Mouvement est effectué. Les champs de la Caractéristique servent à stocker l'information calculée. Le constructeur est utilisé pour formater et initialiser l'ensemble des champs autour de la Solution initiale.

Selon les règles de composition, le constructeur peut prendre en argument le Problème, la Solution et/ou d'autres Caractéristiques, qui à leur réception, sont déjà initialisées. Un constructeur de Caractéritique peut également prendre en argument des paramètres. De même que pour les constructeurs de Solution, ces paramètres sont alors transmis par valeur et doivent apparaître en fin de liste. Qu'elle soit vide ou non, on doit signaler la liste ordonnée de tous les types pris en arguments par le constructeur en dérivant `BuildWith` de la même manière que pour une Solution.

La méthode `track` peut prendre en argument la Solution, le Mouvement à appliquer, des Attributs et/ou des Caractéristiques. Rappelons que seuls les constructeurs peuvent prendre en argument des paramètres. Toutes les fonctions conceptuelles, comme ici `track`, ne peuvent prendre en arguments que d'autres objets de la recherche. Qu'elle soit vide ou non, on doit signaler la liste ordonnée des types composés en dérivant la classe `TrackWith` selon les mêmes conventions que pour `BuildWith`. Les normes de composition assurent qu'à l'appel de `track`, toute Solution ou Caractéristique composée est en phase avec la Solution avant transformation, et tout Attribut composé est en phase avec le Mouvement sur le point d'être appliqué.

La Caractéristique `SatVal` présentée à la figure 5.4 retient dans le champ `value` le nombre de clauses vérifiées par la Solution courante. Dans le tableau `nbtl`, le champ `nbtl[j]`, pour `j` allant de `0` à `pbm.M`, tient à jour le nombre de littéraux vérifiés par

la clause j. Le constructeur compose le Problème SAT à résoudre ainsi que la Solution initiale. La méthode track utilise en plus le Mouvement Flip à effectuer. Rappelons qu'à l'intérieur de track, la Solution sol n'est pas encore transformée.

```
struct SatVal : BuildWith<SAT, Assignment>, TrackWith<SAT, Assignment, Flip>
{
  Array<short> nbtl;
  short value;

  SatVal(const SAT& pbm, const Assignment& sol) : nbtl(pbm.M), value(0)
  {
    for (short j=0; j<pbm.M; j++) nbtl[j] = 0;
    for (short i=0; i<pbm.N; i++)
      for (short k=0; k<pbm.var[i].m; k++)
        if (sol.x[i] == pbm.var[i].s[k]) nbtl[pbm.var[i].j[k]]++;
  }

  void track(const SAT& pbm, const Assignment& sol, const Flip& mvt)
  {
    for (short k=0; k<pbm.var[i].m; k++)
    {
      if (sol.x[mvt.i] == pbm.var[mvt.i].s[k])
        value -= (nbtl[pbm.var[mvt.i].j[k]]-- == 1);
      else
        value += (nbtl[pbm.var[mvt.i].j[k]]++ == 0);
    }
  }
};
```

Figure 5.4 – Programmation de la caractéristique SatVal

```
template <class Dst> struct TspVal
  : BuildWith<Tour<Dst>,Dst>, TrackWith<TspGain<Dst>>
{
  int value;

  TspVal(const Tour<Dst>& sol, const Dst& dst) : value(0)
  {
    for (short c=0; c<dst.N; c++) value += dst(c,sol.succ(c));
  }

  void track(const TspGain<Dst>& gain)
  {
    value -= gain.value;
  }
};
```

Figure 5.5 – Programmation de la caractéristique TspVal

La Caractéristique `TspVal` présentée à la figure 5.5 est constituée du seul champ `value` dans lequel on maintient la longueur de la tournée d'un PVC. Elle est initialisée en fonction de la structure de distance abstraite `Dst` et de la tournée initiale. La méthode `track` prend en argument l'Attribut `TspGain` du chapitre 3.

Contrairement aux deux Caractéristiques `SatVal` et `TspVal` précédentes, les composantes `GeoTemp`, `FlpHshTbl` et `Recency` suivantes ne prélèvent pas d'information qualifiant directement la Solution à laquelle elles sont attachées. Elles n'en demeurent pas moins des Caractéristiques car elles doivent être initialisées en début de recherche et maintenues à chaque exécution de Mouvement.

```
struct GeoTemp : BuildWith<double, double>, TrackWith<>
{
  double value, rate;

  GeoTemp(double v, double r) : value(v), rate(r) { }

  void track()
  {
    value /= rate;
  }
};
```

Figure 5.6 – Programmation de la caractéristique `GeoTemp`

La Caractéristique `GeoTemp` présentée à la figure 5.6 implante un schéma de refroidissement géométrique associable au recuit-simulé. La structure est construite selon les deux paramètres de température initiale et de taux de refroidissement.

La Caractéristique `FlpHshTbl` présentée à la figure 5.7 conserve toutes les coordonnées de `Flip` ordonnées autour de la Solution courante selon leur gain. Chaque indice `i` de variable est enregistré dans une ligne de hachage `tab[k]` correspondant à sa clé. La clé d'un `Flip` correspond à sa valeur de gain décalé de `pbm.W` vers le haut. Rappelons que `pbm.W` est le nombre maximum d'apparitions d'une même variable dans le système et constitue donc une borne sur les gains possibles de Mouvements. De cette façon,

```
struct FlpHshTbl
  : BuildWith<SAT, Assignment, SatVal>, TrackWith<SAT, Assignment, SatVal>
{
  int uk, wg;          meilleure clé enregistrée et gain plancher
  Array<int*> tab;     tab[k][t] : indice de la variable que permute le tième Flip de clé k
  Array<int> len;      len[k] : nombre de Flip associés à la valeur de clé k
  Array<int> key;      key[i] : valeur de clé du Flip qui permute la variable i
  Array<int> adr;      adr[i] : position du Flip qui permute i dans la ligne tab[key[i]]

  FlpHshTbl(const SAT& pbm, const Assignment& sol, const SatVal& val)
   : uk(0), wg(-pbm.W), tab(2*pbm.W+1), len(2*pbm.W+1), adr(pbm.N), key(pbm.N)
  {
     for (int k=0; k<(2*pbm.W+1); k++)
     {  tab[k] = new short[pbm.N];
        len[k] = 0;
     }

     for (int i=0; i<pbm.N; i++)
     {  key[i] = -wg;
        for (short t=0 ; t<pbm.var[i].m ; t++)
          if (sol.x[i]==pbm.var[i].s[t]) key[i] -= (val.nbtl[pbm.var[i].j[t]]==1);
          else                           key[i] += (val.nbtl[pbm.var[i].j[t]]==0);
        if (key[i] > uk) uk = key[i];
        tab[k][adr[i] = len[k]++] = i;
     }
  }

  ~FlpHshTbl() { for (int k=0; k<-2*wg+1; k++) delete [] tab[k]; }

  void track(const SAT&, const Assignment&, const SatVal&, const Flip&) { ... }
};
```

Figure 5.7 – Programmation de la caractéristique `FlpHshTbl`

`tab[0]` (resp. `tab[2*pbm.W]`) est la ligne d'enregistrement du pire (resp. du meilleur) Mouvement possible. La méthode `track`, non représentée sur la figure, corrige les clés en circulant sur l'ensemble des Mouvements qui partagent une même clause que le Mouvement à appliquer. À noter que les Mouvements qui ne partagent pas de clause avec celui appliqué ne change pas de valeur de clé. Pour des clauses à λ variables, une variable apparaît en moyenne dans $\lambda\frac{M}{N}$ clauses. Le nombre de réajustements de clés est donc en moyenne égal à $(\lambda-1)\lambda\frac{M}{N}$. Pour $\lambda = 3$, on obtient en moyenne $6\frac{M}{N}$ réajustements de clés par itération prenant chacun $O(1)$. Sachant que les problèmes difficiles à résoudre sont obtenus pour $\frac{M}{N} \approx 4$ (Hayes, 1997), cette complexité est

avantageuse comparée à la détermination explicite à chaque itération du meilleur Mouvement parmi les N possibles. Notons que comme le constructeur alloue de la mémoire dynamique, un destructeur doit être programmé pour libérer celle-ci. Les objets METALAB suivent à ce titre les conventions habituelles du C++.

```
template <class ATB> struct Recency : BuildWith<int,int>, TrackWith<ATB>
{
  Array<bool> dejavu;
  Array<int> queue;
  int capacity, cursor;

  Recency(int sz, int cap) : dejavu(sz), capacity(cap), queue(cap), cursor(0)
  { for (int i=0; i<capacity; i++) dejavu[queue[i] = i] = true;
    for (int i=capacity; i<sz; i++) dejavu[i] = false;
  }

  void track(const ATB& atb)
  { for (short d=0; d<atb.dim; d++)
    { dejavu[queue[cursor]] = false;
      dejavu[queue[cursor] = atb.value[d]] = true;
      if (++cursor == capacity) cursor = 0;
    }
  }

  template <class BTA> bool intercept(const BTA& bta)
  { for (short d=0; d<bta.dim; d++)
      if (dejavu[bta.value[d]]) return false;
    return true;
  }
};
```

Figure 5.8 – Programmation de la caractéristique Recency

La classe gabarit Recency présentée à la figure 5.8 suit également un modèle de Caractéristique. Elle est compilable pour tout Attribut ATB définissant un tableau d'entiers value de dimension dim dans lequel sont stockées les valeurs de champs de l'Attribut. La réalisation de Recency s'appuie sur un tableau dejavu pour lequel dejavu[v] est affecté à true si la valeur d'Attribut v a été récemment enregistrée. Le tableau queue liste explicitement les valeurs récemment enregistrées. La capacité capacity de Recency correspond au nombre de valeurs retenues, de sorte qu'une

valeur d'Attribut enregistrée reste en mémoire $\frac{\texttt{capacity}}{\texttt{dim}}$ itérations. À l'initialisation, le constructeur de `Recency` enregistre arbitrairement les valeurs d'Attributs allant de `0` à `capacity`. La conception de `Recency` suppose qu'il est possible d'élever un tableau dont la taille correspond au nombre possible de valeurs d'Attributs différentes. Dans ce cas, la mise à jour et la consultation de la mémoire s'effectue en temps constant. Lorsque le tableau `dejavu` devient trop grand, il est possible de programmer d'autres structures de récence mais dans ce cas, le temps de consultation est proportionnel à la longueur de la liste. Il est à noter que la fonction `intercept` ne fait pas partie des spécifications d'une Caractéristique. Elle est définie ici en interne de sorte que le statut tabou d'un mouvement puisse être consulté de manière standardisée quelle que soit l'implantation de la récence. Pour être fonctionnelle, `intercept` doit être appelée avec un Attribut `BTA` inverse d'`ATB`.

5.4 Concept de Mouvement

Un Mouvement est constitué d'une méthode `move` et d'un jeu de coordonnées. La méthode `move` transforme la Solution qui lui est associée en fonction du jeu de coordonnées. Les constructeurs vide et par copie doivent être définis. Le premier permet de découpler la déclaration d'un Mouvement de son initialisation dont les Serveurs ont la responsabilité. Il doit seulement formater l'instance créée. Le second doit effectuer une copie profonde du jeu de coordonnées. Les constructeurs par défaut du C++ remplissent généralement ces objectifs. À noter que la dérivation de `BuildWith` ne doit pas intervenir ici car on connaît à l'avance la signature des deux constructeurs.

La méthode `move` doit prendre en argument la Solution et éventuellement des Attributs. La liste ordonnée des types composés doit être signalée en dérivant `MoveWith`. Le premier objet composé doit obligatoirement être la Solution modifiée et elle est

passée en argument comme référence non constante. Les autres arguments sont passés en référence constante. Selon les normes de composition, à l'appel de `move`, les Attributs composés ont au préalable été calculés en phase avec le Mouvement effectué.

```
struct Flip : MoveWith<Assignment>
{
  short i;   index de la variable booléenne à inverser

  void move(Assignment& sol) { sol.x[i] =!sol.x[i]; }
};
```

Figure 5.9 – Programmation du Mouvement `Flip`

La fonction `move` du Mouvement `Flip` présenté à la figure 5.9 prend seulement en argument la Solution `Assigment` dont elle inverse la variable indexée par `i`.

```
template <class Dst> struct Twex : MoveWith<Tour<Dst>>
{
  short c[2][2];     c[0] et c[1] doivent suivre la même direction

  void move(Tour<Dst>& sol)
  { short dir = (sol.succ(c[0][0]) == c[0][1]);
    short len = sol.rank[c[dir][1-dir]] - sol.rank[c[1-dir][dir]];

    On inverse la plus petite des deux sous-séquences possibles
    if (2+2*len<((len<0)?-sol.N:sol.N)) sol.flip(c[1-dir][dir], c[dir][1-dir]);
    else                                sol.flip(c[dir][dir], c[1-dir][1-dir]);
  }
};
```

Figure 5.10 – Programmation du Mouvement `Twex`

La fonction `move` du Mouvement `Twex` présenté à la figure 5.10 prend en argument la tournée dont il faut inverser une sous-séquence. Le jeu de coordonnées d'un Mouvement `Twex` est constitué des deux arêtes sortantes $\{$`c[0][0],c[0][1]`$\}$ et $\{$`c[1][0],c[1][1]`$\}$ dont les écritures doivent être orientées dans la même direction relativement à la tournée courante. La fonction `move` opère sur la plus petite

des deux sous-séquences qu'on peut inverser pour effectuer le 2-*opt* spécifié par le jeu de coordonnées. De même que `Tour` et `TspGain`, `Twex` est une classe générique définie en fonction du type `Dst`. Notons que dans les deux exemples, les constructeurs n'ont pas été explicitement programmés car les versions par défaut du C++ réalisent correctement les spécifications de Mouvement.

5.5 Concept d'Attribut

Un Attribut est constitué d'une méthode `scan` et d'un certain nombre de champs. La méthode `scan` calcule une information qui doit être mise à jour chaque fois qu'un nouveau Mouvement est en cours d'examen. Les champs de l'Attribut servent à stocker l'information calculée.

Les spécifications relatives aux constructeurs d'un Attribut sont similaires à celles d'un Mouvement. Le constructeur vide sert seulement à formater l'`Attribut`, l'initialisation des champs variables étant par la suite du ressort de la méthode `scan`. Le constructeur par copie doit effectuer une copie profonde des champs de l'Attribut. Dans la plupart des cas, les constructeurs que propose par défaut le C++ remplissent convenablement ces deux charges. Comme dans le cas des Mouvements, la dérivation de `BuildWith` n'a pas ici lieu d'être.

La méthode `scan` peut prendre en argument des composantes de Solution, de Mouvement, d'Attribut et/ou de Caractéristique. Dans ce cas, la liste ordonnée des types composés doit être signalée en dérivant `ScanWith`. Tous les objets composés sont passés en argument en référence constante. Selon les normes de composition, l'état de ces objets à l'appel de `scan` dépend du contexte. Lorsqu'un Attribut est calculé au moment de l'application d'un Mouvement pour mettre à jour certaines composantes, les Caractéristiques composées sont en phase avec la Solution avant transformation

et les Attributs composés sont en phase avec le Mouvement à appliquer. Lorsqu'un Attribut est calculé pour mettre en oeuvre la sélection du Mouvement lors de l'énumération préalable du voisinage, les Caractéristiques composées sont en phase avec la Solution courante et les Attributs composés sont en phase avec le Mouvement examiné.

```
struct SatGain : ScanWith<SAT, Assignment, Flip, SatVal>
{
  short value;
  void scan(const SAT& pbm, const Assignment& sol, const Flip& mvt, const SatVal& val)
  {
     value = 0;
     for (short k=0; k<pbm.var[mvt.i].m; k++)
     {
        if (sol.x[mvt.i] == pbm.var[mvt.i].s[k])
           value -= (val.nbtl[pbm.var[mvt.i].j[k]] == 1);
        else
           value += (val.nbtl[pbm.var[mvt.i].j[k]] == 0);
     }
  }
};
```

Figure 5.11 – Programmation de l'Attribut SatGain

L'Attribut SatGain présenté à la figure 5.11 est constitué d'un champ valeur qui représente le gain du Mouvement Flip examiné. La méthode scan prend en argument le Problème SAT, la Solution Assignment, le Mouvement Flip et la Caractéristique SatVal. Cette dernière permet d'accélérer le calcul du gain puisque pour chaque clause où apparaît la variable dont on inverserait la valeur, le gain local peut être évalué en $O(1)$ en s'appuyant sur le nombre de littéraux vérifiés que comptabilise SatVal. Sans ce mécanisme, le temps de calcul du gain local associé à une clause serait proportionnel au nombre de littéraux par clause. Au total, la complexité de scan est en $O(\texttt{pbm.var[i].m})$. Les deux constructeurs par défaut du C++ réalisent ici les spécifications d'Attribut.

La méthode scan de l'Attribut gabarit ToAdj, présenté à la figure 5.12, détermine les arcs qui entreraient dans la tournée si le Mouvement Twex passé en argument

```
template <class Dst> struct ToAdj : ScanWith<Twex<Dst>>
{
  const short dim;
  Array<int> value;
  ToAdj() : dim(2), edge(2) {}

  void scan(const Twex<Dst>& mvt)
  {
     if (mvt.c[0][0] < mvt.c[1][0])
        value[0] = (mvt.c[1][0]*(mvt.c[1][0]-1))/2 + mvt.c[0][0];
     else
        value[0] = (mvt.c[0][0]*(mvt.c[0][0]-1))/2 + mvt.c[1][0];

     if (mvt.c[0][1] < mvt.c[1][1])
        value[1] = (mvt.c[1][1]*(mvt.c[1][1]-1))/2 + mvt.c[0][1];
     else
        value[1] = (mvt.c[0][1]*(mvt.c[0][1]-1))/2 + mvt.c[1][1];
  }
};
```

Figure 5.12 – Programmation de l'Attribut `ToAdj`

était exécuté. On considère pour cela une numérotation des arêtes selon la formule $\frac{c_2*(c_2-1)}{2} + c_1$ où $0 \leq c_1 < c_2 < N$ sont les indices des deux sommets de l'arête considérée. Les numéros des deux arêtes entrantes sont enregistrés dans le tableau `value`. Notons que le constructeur par défaut est ici corrigé afin de formater la taille de `value` et de garder en mémoire sa dimension `dim`, c'est-à-dire le nombre d'arêtes entrantes, ici égal à deux. L'adaptation du constructeur par copie est en revanche superflue car la copie d'un `Array` est profonde. Remarquons que la présence de la dimension `dim` permet de compiler la Caractéristique `Recency<ToAdj>`, par exemple, pour réaliser une méthode taboue. La programmation d'un `Attribut FrAdj` servant à déterminer les arêtes sortantes est similaire au cas de `ToAdj`.

L'Attribut `Metropolis` présenté à la figure 5.13 implante le critère de Métropolis selon le schéma de composition suggéré au chapitre 3. La composante est générique relativement aux types `GAIN` et `TEMP`. Elle compile si l'Attribut `GAIN` et la Caractéristique `TEMP` définissent chacun un champ `value` correspondant respectivement au gain du Mouvement examiné et à la température du schéma de refroidissement.

```
template <class GAIN, class TEMP> struct Metropolis : ScanWith<GAIN, TEMP>
{
  short value;
  void scan(const GAIN& gain, const TEMP& temp)
  {
    if (gain.value >= 0) value = 1;
    else value = exp(-gain.value / temp.value) - rand();
  }
};
```

Figure 5.13 – Programmation du Attribut `Metropolis`

5.6 Concept de Serveur

Un Serveur est constitué des méthodes **begin** et **next**, qui doivent avoir la même liste d'arguments dont le premier correspond au Mouvement devant être modifié, passé en référence non constante. Le reste des arguments peut comprendre des Caractéristiques et/ou la Solution ainsi que le Problème. Selon les normes de composition, les Caractéristiques composées sont en phase avec la Solution courante. Selon le contexte, un Serveur contient un certain nombre de champs utiles pour développer l'énumération.

Le constructeur du Serveur ne sert qu'à formater l'objet créé en mémoire. Il ne peut pas prendre d'objets de la recherche mais peut prendre des paramètres en arguments. La méthode **begin** initialise les coordonnées de Mouvement que la méthode **next** modifie ultérieurement. La méthode **next** retourne un booléen : **true** si le Serveur a pu fournir de nouvelles coordonnées de Mouvement, et **false** sinon. Dans ce dernier cas, la méthode **next** ne doit plus être utilisée par l'Explorateur et le Mouvement actif est éventuellement invalide. Même si elle est vide, on doit toujours indiquer la liste des arguments du constructeur (resp. des méthodes **begin** et **next**) en dérivant `BuildWith` (resp. `EnumerateWith`) selon les conventions habituelles.

Le Serveur `ByMidBndDiaTwxSvr` présenté à la figure 5.14 énumère le voisinage d'une tournée pour le Mouvement `Twex`. Le Serveur traverse seulement les Mouvements qui

```
template <class Dst> struct ByMidBndDiaTwxSvr
  : BuildWith<short, short>, EnumerateWith<Twex<Dst>, Tour<Dst>>
{
  short dia, dmin, dmax;
  ByMidBndDiaTwxSvr(short dmn, short dmx) : dmin(dmn), dmax(dmx) {}

  void begin(Twex<Dst>& mvt, const Tour<Dst>& sol)
  {
    mvt.c[0][0] = sol.pred(mvt.c[0][1] = sol.city[0]);
    mvt.c[1][1] = sol.succ(mvt.c[1][0] = sol.succ(mvt.c[0][1], dmin-1));
    dia = dmin;
  }

  bool next(Twex<Dst>& mvt, const Tour<Dst>& sol)
  {
    if (++dia <= dmax)
    {
      if ((dia-dmin)%2) mvt.c[0][0] = sol.pred(mvt.c[0][1] = mvt.c[0][0]);
      else mvt.c[1][1] = sol.succ(mvt.c[1][0] = mvt.c[1][1]);
    }
    else
    {
      mvt.c[0][0] = sol.pred[mvt.c[0][1] = sol.succ(mvt.c[0][1],1+(dia-dmin)/2)];
      if (mvt.c[0][1] == sol.city[0]) return false;
      mvt.c[1][1] = sol.succ(mvt.c[1][0] = sol.succ(mvt.c[0][1], dmin-1));
      dia = dmin;
    }
    return true;
  }
};
```

Figure 5.14 – Programmation du Serveur ByMidBndDiaTwxSvr

inversent une sous-séquence de longueur comprise entre les paramètres `dmin` et `dmax`. À chaque appel de `next`, le centre de la sous-séquence à inverser reste inchangé mais le rayon est augmenté tantôt à gauche tantôt à droite d'une unité. Dans le cas où le diamètre de la sous-séquence deviendrait supérieur à `dmax`, le centre de la sous-séquence est plutôt incrémenté d'une unité et le diamètre est réinitialisé.

Le Serveur `ByIdxFlpSvr` présenté à la figure 5.15 énumère le voisinage relatif au Mouvement `Flip`. La méthode `begin` initialise l'indice de la variable à permuter à 0 et la méthode `next` incrémente cet indice. La méthode `next` vérifie par ailleurs si l'indice de variable est toujours cohérent, c'est-à-dire inférieur au nombre de variables.

```
ByIdxFlpSvr : BuildWith<>, EnumerateWith<Flip, SAT>
{
  void begin(Flip& mvt, const SAT& pbm) { mvt.i = 0; }
  bool next(Flip& mvt, const SAT& pbm) { return (++mvt.i < pbm.N); }
};
```

Figure 5.15 – Programmation du Serveur ByIdxFlpSvr

Le Serveur ByGanFlpSvr présenté à la figure 5.16 énumère le même voisinage que ByIdxFlpSvr en s'appuyant sur la Caractéristique FlpHshTbl. Le premier Flip énuméré correspond donc au meilleur gain possible et l'énumération des Flip se poursuit selon une suite un gain décroissante. Les coordonnées du Mouvement pointé dans la table de hashage fht composée sont déduites des champs adr et key.

```
ByGanFlpSvr : BuildWith<>, EnumerateWith<Flip, FlpHshTbl>
{
  short key, adr;

  void begin(Flip& mvt, const FlpHshTbl& fht)
  {
    mvt.i = fht.tab[key = fht.uk][adr = 0];
  }

  bool next(Flip& mvt, const FlpHshTbl& fht)
  {
    if (++adr == fht.len[key])
    {
      adr = 0;
      do { if (key == 0) return false; }
      while (fht.len[--key] == 0);
    }
    mvt.i = fht.tab[key][adr];
    return true;
  }
};
```

Figure 5.16 – Programmation du Serveur ByGanFlpSvr

5.7 Concept d'Explorateur

La fonction find d'un Explorateur visite le voisinage de la solution courante pour déterminer un mouvement à exécuter si elle retourne true. Cette fonction est intégrée

dans une classe dérivée de `FindWith` dont le premier paramètre gabarit indique le Serveur à utiliser et les suivants les arguments de la fonction `Find`. Les composantes déjà présentées aux figures 3.3 et 4.5 sont deux exemples d'Explorateurs. Rappelons que la fonction `find` peut seulement composer des Caractéristiques ou des Attributs : les Caractéristiques sont des objets constants et les Attributs peuvent être composés de manière statique ou dynamique. Dans le premier cas, ces derniers sont systématiquement recalculés à chaque modification du Mouvement actif. Dans le deuxième cas, ils sont seulement calculés sur demande selon la syntaxe vue au chapitre précédent. Lorsque cela est utile, un Explorateur peut contenir un certain nombre de champs. Un constructeur initialise dans ce cas ces champs en fonction de paramètres appropriés.

```
template <class SVR, class GAIN> struct BestImprovement : FindWith<SVR, GAIN>
{
  bool find(const GAIN& gain)
  {
     Buffer best;
     best.import();
     while (next()) if (gain.value > best.get<GAIN>().value) best.import();
     best.export();
     return (gain.value > 0);
  }
};
```

Figure 5.17 – Programmation de l'Explorateur `BestImprovement`

La programmation de `find` nécessite parfois de retenir en mémoire des Mouvements visités. C'est par exemple le cas d'une recherche en meilleure amélioration où chaque fois qu'un nouveau meilleur Mouvement est examiné, il doit être retenu à des fins de comparaison. À cet effet, on dispose dans METALAB de composantes tampons appelées `Buffer`. Lorsque la fonction `import` d'un tampon est appelée, le Mouvement actif ainsi que les Attributs ayant déjà été calculés sur ce Mouvement sont enregistrés. Lorsque la fonction `export` est appelée, les composantes enregistrées dans le tampon sont réaffectées au mouvement actif et à ces dépendances de sorte que la recherche retrouve son état préalable. Comme l'illustre l'Explorateur `BestImprovement` de la

figure 5.17, on peut accéder facilement aux différents Attributs d'un tampon. On utilise dans ce cas la syntaxe `get<C>()` où `C` est l'Attribut consulté.

```
template <class SVR, class FAST_GAIN, class SLOW_GAIN>
struct BoostBestImprovement : FindWith<SVR, FAST_GAIN, If<SLOW_GAIN>>
{
  float espsilon;
  BoostBestImprovement(float eps) : epsilon(eps) {}

  bool find(const FAST_GAIN& fast_gain, If<SLOW_GAIN>& slow_gain)
  {
    Buffer best;
    slow_gain();
    best.import();
    while (next())
      if (fast_gain.value > (1 - epsilon) * best.get<SLOW_GAIN>().value)
        if (slow_gain().value > best.get<SLOW_GAIN>().value) import();
    best.export();
    return (slow_gain.value > 0);
  }
};
```

Figure 5.18 – Programmation de l'Explorateur `BoostBestImprovement`

L'Explorateur `BoostBestImprovement`, présenté à la figure 5.18, combine les fonctionnalités de tampon et de composition dynamique afin d'accélérer l'évaluation des Mouvements pour une recherche en meilleure amélioration. On suppose ici qu'on dispose d'un premier Attribut de gain de mouvement `FAST_GAIN`, facile à calculer mais approximatif et d'un second, `SLOW_GAIN`, exact mais coûteux. Cet exemple montre également comment il est possible de paramétrer un Explorateur. Ici, le champ `epsilon` contrôle le seuil à partir duquel `FAST_GAIN` doit être calculé. À noter que la dérivation de `BuildWith` n'est pas nécessaire dans le cas des Explorateurs.

L'Explorateur `BiphasicBestImprovement`, présenté à la figure 5.19, implante la recherche d'un meilleur `Mouvement` en deux phases. D'abord, le Serveur énumère le voisinage afin de retenir les `dim` meilleurs `Mouvements` relativement à l'Attribut `ATB1`. L'ensemble `best1` de ces Mouvements élites est ensuite retraversé pour finalement

sélectionner le meilleur d'entre eux relativement à l'Attribut ATB2. Si par exemple on souhaitait implanter une stratégie de jonction entre solutions élites, ATB1 mesurerait le rapprochement vis-à-vis de la solution cible et ATB2 le gain traditionnel de Mouvement. À noter que l'utilisation du tableau best1 et du paramètre dim conduit à définir un constructeur et un destructeur selon les conventions du C++ habituelles.

```
template <class SVR, class ATB1, class ATB2>
struct BiphasicBestImprovement : FindWith<SVR, ATB1, If<ATB2>>
{
  Array<Buffer*> best1;
  short dim;

  BiphasicBestImprovement(int d) : dim(d), best1(d)
  { for (short i=0; i<d; i++) best1[i] = new Buffer; }

  ~BiphasicBestImprovement() { for (short d=0; d<dim; d++) delete best1[d]; }

  bool find(const ATB1& atb1, If<ATB2>& atb2)
  {
    for (short i=0; i<dim; i++)
    {
      short j=i-1;
      while (j>0 && best1[j-1]->get<ATB1>() < atb1) best1[j] = best[--j];
      best1[j]->import();
      next();
    }

    do {
      short j=dim;
      while (j>0 && best1[j-1]->get<ATB1>() < atb1) best1[j] = best[--j];
      if (j < dim) best1[j]->import();
    } while (next());

    best1[0]->export(); atb2();
    Buffer best2; best2.import();

    for (short i=1; i<dim; i++)
    {
      best1[i]->export();
      if (atb2().value > best2.get<ATB2>().value) best2.import();
    }

    best2.export();
    return true;
  }
};
```

Figure 5.19 – Programmation de l'Explorateur BiphasicBestImprovement

L'Explorateur `BestNonTabu`, présenté à la figure 5.20, implante enfin une recherche taboue. L'Attribut `BTA` doit être un Attribut inverse de celui composé par la Caractéristique `RECENCY`. À noter que syntaxiquement, toute Caractéristique peut jouer le rôle de `RECENCY` à partir du moment où elle définit une fonction `intercept`.

```
template <class SVR, class GAIN, class BTA, class RECENCY>
struct BestNonTabu : FindWith<SVR, GAIN, If<BTA>, RECENCY>
{
  bool find(const GAIN& gain, If<BTA>& bta, const RECENCY& recency)
  {
    Buffer best;
    while (recency.intercept(bta()) next();
    best.import();

    while (next())
    {
      if (gain.value <= best.get<GAIN>().value) continue;
      if (recency.intercept(bta()) continue;
      best.import();
    }

    best.export();
    return true;
  }
};
```

Figure 5.20 – Programmation de l'Explorateur `BestNonTabu`

5.8 Finalisation de l'algorithme

Le code présenté à la figure 4.1 au chapitre précédent donne un exemple de programme principal reposant sur la librairie METALAB. Lorsqu'on finalise un programme, on doit successivement déclarer, initialiser, et enfin itérer la recherche.

Déclaration de la recherche. La première étape à suivre lorsqu'on finalise un algorithme consiste à déclarer un objet Explorateur. Par exemple, l'instruction suivante :

```
FirstImprovement<LinearTwexer,Metropolis<TspGain<MtxDst>,GeoTemp>> explorer;
```

crée un objet `explorer` correspondant à une recherche en meilleure amélioration.

Lorsque l'Explorateur utilisé présente un certain nombre de paramètres de construction, on doit passer ces paramètres lors de l'étape de déclaration. Ainsi, l'instruction :

```
BoostBestImprovement<SomeServer,SomeFastGain,If<SomeSlowGain>> explorer(0.1);
```

crée un objet `explorer` dont l'Attribut `SomeSlowGain` n'est calculé que lorsque la valeur de `SomeFastGain` ne dépasse pas de plus de 10% la valeur record de `SomeSlowGain`.

Initialisation de la recherche. Avant de procéder à la recherche, on doit initialiser l'Explorateur déclaré, c'est-à-dire construire une Solution de départ ainsi qu'initialiser toutes les Caractéristiques attachées à cette Solution. Ainsi, l'instruction suivante :

```
explorer.init<Tour<Dst>,GeoTemp>(dst, random, 1000, 0.999);
```

initialise la tournée d'un PVC selon le problème à résoudre `dst` et le paramètre `random` (algorithme de construction utilisé) puis initialise le schéma de refroidissement selon les paramètres `1000` (température intiale) et `0.999` (taux de décroissance géométrique). Par convention, la liste des arguments de la fonction `init` doit être définie par groupes de paramètres. Chaque groupe correspond à la liste ordonnée des paramètres de construction d'une composante donnée et l'ensemble des groupes doit suivre le même ordre que la liste des types de composantes fournies entre chevrons. Dans l'exemple précédent, comme on utilise la fonction `init<Tour<Dst>,GeoTemp>`, le groupe des paramètres d'initialisation de `Tour<Dst>` précède le groupe des paramètres de `GeoTemp`. À noter que seuls les paramètres d'initialisation et non les objets de la recherche composés par les constructeurs doivent être ici passés en arguments.

Itération de la recherche. Pour dérouler la recherche, on itère sur la fonction `move()` tant qu'un critère de terminaison arbitraire n'est pas vérifié. Ainsi, la ligne :

```
do { explorer.move(); } while (explorer<GeoTemp>().value > 0.0001);
```

réitère la recherche tant que la température du schéma de refroidissement n'atteint pas une valeur seuil infinitésimale. À noter que pour exprimer les critères d'arrêt, on

dispose de fonctions d'accès à l'état de la recherche sous la syntaxe `explorer<C>()` où `C` représente la composante de l'algorithme à consulter. Enfin, soulignons que les Explorateurs s'intègrent sans contraintes et très naturellement au C++ : on peut allouer dynamiquement des Explorateurs, construire des tableaux d'Explorateurs, etc.

5.9 Conclusion

La présentation du logiciel METALAB faite dans ce chapitre s'est appuyée sur les contextes du voyageur de commerce et du problème *SAT*. À ce titre, nous avons tenté d'illustrer que METALAB permet, d'une part, d'exprimer des constructions de recherche locale géométriques propres à l'optimisation (cf. *PVC*) et que prennent en charge les cadres d'application, et d'autre part, de couvrir des mécanismes d'incrémentation de la recherche souvent utilisés par les formulations analytiques de recherche locale propres à la satisfaction de contraintes (cf. *SAT*) et sur lesquels se concentrent les langages déclaratifs. Ce résultat tient de ce que ni l'hypothèse d'une réduction à une famille de problèmes remarquables (comme les problèmes de satisfaction de contraintes) ni l'hypothèse d'une réduction à une famille de métaheuristiques classiques (comme les collections d'algorithmes pris en charge par les cadres d'application) n'a été faite ici. Nous avons aussi montré que la librairie permet de construire très facilement de nouveaux schémas algorithmiques comme en témoigne par exemple les Explorateurs `BoostBestImprovement` et `BiphasicBestImprovement`.

Notons enfin que la présentation des différentes composantes amène à faire la distinction entre spécifications *conceptuelles* et *contextuelles*. Lors de la réalisation d'une composante, les spécifications conceptuelles portent sur le rôle qu'elle joue dans le modèle informatique établi aux chapitres 3 et 4. Par exemple, la composante `Recency`, qui conserve une information devant être mise à jour à chaque exécution de mouvement, est une Caractéristique, et à ce titre, dispose d'une fonction `track`. Mais

en tant que Liste taboue, cette composante dispose également d'une fonction contextuelle, `intercept`, qui permet de vérifier si les valeurs d'un Attribut inverse recoupent les champs enregistrés dans la liste. Dans ce cas, les spécifications contextuelles assurent que l'Explorateur `BestNonTabu` fonctionne quelle que soit l'implantation de Liste taboue qui lui est passée en argument. De manière similaire, on remarque que la Liste taboue `Recency` fait l'hypothèse que l'Attribut `ATB` qu'elle compose est Tabulaire, c'est-à-dire qu'il contient un tableau `value` de dimension `dim`. Les expressions `atb.dim` et `atb.value[.]` sont à ce titre des expressions valides correspondant aux spécifications contextuelles d'Attribut Tabulaire. Similairement, les Attributs `TspGain`, `SatGain` et `Metropolis`, qui contiennent chacun un champ `value`, appartiennent à la catégorie des Attributs Singletons. La figure 5.21 suivante schématise l'ensemble des types de Variables ainsi mis en évidence.

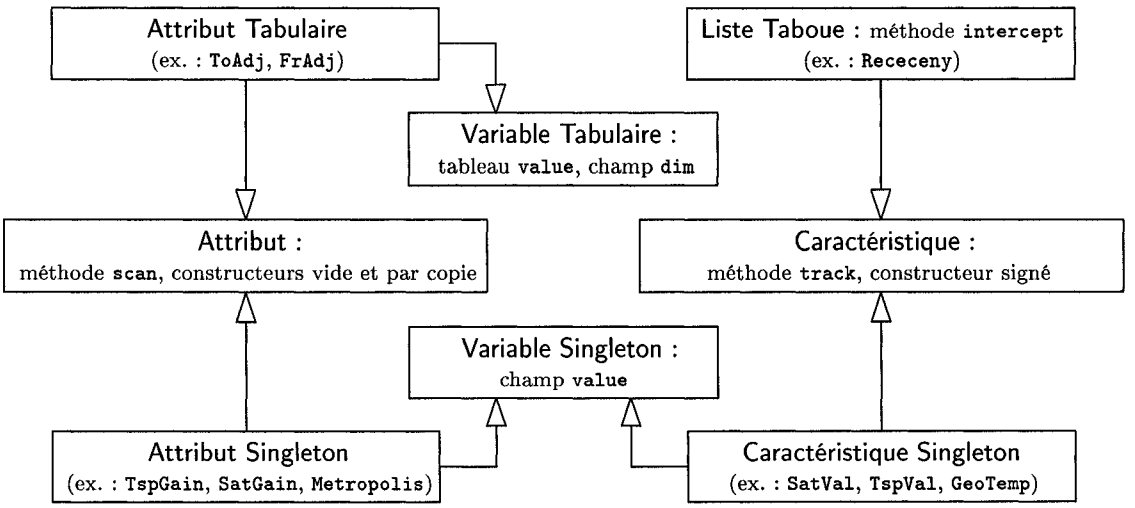


Figure 5.21 – Spécifications conceptuelles et contextuelles des Variables

Pour déterminer la validité d'une composante, il suffit alors de spécifier à quels concepts doivent appartenir les composantes prises en argument. Par exemple, on notera que l'Explorateur `BestNonTabu` est valide si et seulement si `GAIN` est un Attribut Singleton, `BTA` un Attribut Tabulaire et `RECENCY` une Liste Taboue.

CHAPITRE 6 : REVUE DE PROBLÈMES

Alors que dans le prochain chapitre nous chercherons à illustrer l'utilité de la librairie pour la conduite de projets avancés, notre objectif est ici de montrer que METALAB est applicable à de nombreuses formes de recherche locale. Pour cela, nous revisitons un certain nombre de problèmes de la littérature et dans chaque cas indiquons de quelle manière les composantes peuvent être implantées au moyen du logiciel.

6.1 Le PVC avec coûts d'arêtes dépendant du temps

Dans le problème du voyageur de commerce avec coûts d'arêtes dépendant du temps (Van Der Wiel et Sahinidis, 1995), noté $PVCDT$, le coût d'une arête dépend non seulement des deux sommets reliés mais également de la position de l'arête dans la tournée. Par exemple, même si les deux permutations 1-2-3-1 et 2-3-1-2 sont équivalentes dans le cas du PVC, elles n'ont pas forcément ici la même longueur puisque l'arête 1-2 intervient en début de tournée dans le premier cas et en fin de tournée dans le second. À noter qu'un cas particulier du $PVCDT$, appelé problème du voyageur de commerce cumulatif (Lucena, 1990, Fischetti, Laporte et Martello, 1992), concerne le cas où le coût de la $k^{\text{ième}}$ arête de la tournée vaut k fois le coût de l'arête dans la version du PVC. Dans ce cas, le $PVCDT$ revient à chercher la tournée dont le temps moyen d'arrivée à chaque sommet (ou de manière équivalente le temps cumulé d'arrivée) est minimisé.

Bien que la résolution d'un $PVCDT$ au moyen de la recherche locale présente de nombreuses similarités avec le cas du PVC, deux nouveaux aspects apparaissent principalement. Premièrement, il faut noter que l'utilisation exclusive d'un mouvement de type *r-opt* peut s'avérer limitée. Par exemple, remarquons que de tels

mouvements peinent à relier deux tournées pourtant identiques à une translation près, comme le sont les deux exemples présentés au préalable. À cet effet, il peut être utile de considérer plusieurs voisinages, certains travaillant sur les aspects de routage (par exemple un *r-opt*) et d'autres sur les aspects de temps (par exemple un mouvement décalant la tournée sans en changer l'ordre). Deuxièmement, le calcul du coût d'un *r-opt* s'avère plus complexe que pour un PVC. Par exemple, notons que comme un 2-*opt* revient à inverser une sous-séquence de la tournée courante, le calcul du coût doit non seulement tenir compte des différences entre les arêtes entrantes et sortantes comme habituellement, mais aussi de la nouvelle position que prennent les arêtes situées dans la sous-séquence inversée. En effectuant ce calcul sans précaution, l'examen d'un 2-*opt* s'effectue alors en $O(N)$ dans le pire cas. L'étude de cas que nous présentons dans cette section se limite à la programmation du seul voisinage 2-*opt*. Pour celui-ci, nous montrons comment réutiliser les composantes mises en place pour le PVC au chapitre 5 à l'exception de l'Attribut de gain pour lequel nous proposons une fonction `scan` adaptée, fonctionnant en temps constant.

Pour ce faire, supposons que l'on connaisse le gain $\texttt{value}_{s_i \ldots s_j}$ du mouvement inversant la sous-séquence s_i-...-s_j, où s_k dénote le $k^{\text{ième}}$ sommet de la tournée. En notant $d^t_{s,s'}$ le coût de l'arête reliant les sommets s et s' en $t^{\text{ième}}$ position de la tournée, le gain $\texttt{value}_{s_{i-1} \ldots s_{j+1}}$ du mouvement inversant s_{i-1}-...-s_{j+1} peut alors directement être obtenu par la formule :

$$\texttt{value}_{s_{i-1} \ldots s_{j+1}} =$$

$$\texttt{value}_{s_i \ldots s_j} + d^{i-2}_{s_{i-2},s_{i-1}} - d^{i-2}_{s_{i-2},s_{j+1}} + d^{i-1}_{s_{i-1},s_j} - d^{i-1}_{s_{j+1},s_j} + d^{j}_{s_i,s_{j+1}} - d^{j}_{s_i,s_{i-1}} + d^{j+1}_{s_{j+1},s_{j+2}} - d^{j+1}_{s_{i-1},s_{j+2}}.$$

Afin d'exploiter cette formule, notons que le voisinage 2-*opt* doit être énuméré judicieusement, c'est-à-dire en gardant constant tant que possible le centre de la sous-séquence inversée, autrement dit, en partant d'un diamètre 2 ou 3, et en allongeant chaque fois la sous-séquence à inverser d'une unité à gauche et d'une unité à droite.

Lorsque le diamètre atteint une valeur seuil, on doit alors renouveler la procédure, soit en passant à des mouvements inversant des sous-séquences paires à des mouvements inversant des sous-séquences impaires, soit en revenant à des mouvements inversant des sous-séquences paires, mais en incrémentant le centre de la sous-séquence.

```
template <class Dst> struct TdTspGain : ScanWith<Twex<Dst>, Tour<Dst>, Dst>
{
  int value, alternative_value;

  void scan(const Twex<Dst>& mvt, const Tour<Dst>& sol, const Dst& dst)
  {
    Permuter value et alternative_value
    Si mvt a un diamètre 2 ou 3, réinitialiser value
    Sinon réajuster value selon la formule value_{s_{i-1}...s_{j+1}} = value_{s_i...s_j} + ...
  }
};
```

Figure 6.1 – Programmation de l'Attribut TdTspGain

Un schéma d'énumération pertinent et déjà présenté au chapitre 5, correspond au Serveur ByMidBndDiaTwxSvr à cela près que ByMidBndDiaTwxSvr rallonge la sous-séquence à inverser tantôt à gauche, tantôt à droite. À cet égard, l'Attribut TdTspGain qu'on propose doit simplement gérer parallèlement le cas des diamètres pairs et impairs. Le pseudo-code de la fonction scan obtenue est indiqué à la figure 6.1. À noter que TdTspGain doit composer la tournée afin de pouvoir calculer le diamètre du mouvement examiné.

En pratique, la seule autre pièce logicielle à reprogrammer est la composante de Problème qui doit ici proposer un opérateur de distance prenant trois arguments : les deux indices des deux sommets ainsi que le rang de l'arête dans la tournée. On notera à ce propos que même si certains Attributs du chapitre 5, comme FrAdj, sont définis génériquement par rapport au type de Problème, l'opérateur de distance n'est en fait seulement utilisé que dans l'Attribut de gain. Cette propriété assure par exemple que :

```
BestNonTabu<ByMidBndDiaTwxSvr<P>, TdTspGain<P>, FrAdj<P>, Recency<ToAdj<P>>>
```

implante une méthode taboue directement utilisable lorsque P représente une composante de Problème valide correspondant au $PVCDT$. La réalisation d'un recuit-simulé serait également aussi directement applicable.

6.2 Le problème de la p-médiane

Dans le problème de la p-médiane, on se donne un ensemble U de n clients, un ensemble F de m sites potentiels, et une fonction de distance $d(u, f)$ (qu'on notera aussi $d_{u,f}$) à valeurs entières positives, où $u \in U$ et $f \in F$. Étant donné un sous-ensemble S de F, le coût de service d'un client u est défini par $\min_{f \in S} d(u, f)$. L'objectif est de choisir p sites parmi les m de sorte que la somme des coûts de service soit minimisée.

L'étude de cas que nous proposons se limite aux mouvements consistant à remplacer un site, noté f_{out}, inclus dans la solution courante, par un site, noté f_{in}, n'en faisant pas partie. Nous nous limitons également à ne décrire que les composantes intervenant dans le choix d'un meilleur mouvement. Les composantes visant à introduire des formes de mémoire ne sont donc pas traitées ici. L'objectif est de montrer que METALAB permet de programmer progressivement deux implantations possibles du choix du meilleur mouvement. La première correspond à l'approche la plus naturelle mais est coûteuse en calculs. La seconde consiste à récupérer le calcul de chaque gain effectué à une itération donnée pour l'itération suivante. Cette seconde approche présente certaines similarités avec Resende et Werneck (2003) dont les travaux visent à améliorer la factorisation du calcul du gain initialement proposée par Whitaker (1983), et par exemple utilisée par Hansen et Mladenovic (1997) dans une recherche à voisinage variable.

Pour la suite, on suppose qu'on dispose d'une composante de Problème `PmedDst` pour la p-médiane munie d'un opérateur de distance correspondant à la fonction

$d(u, f)$. On suppose également qu'une Solution est représentée par la composante `Subset` dans laquelle deux tableaux d'entiers, `open`, (de dimension p), et `closed`, (de dimension $m - p$), contiennent respectivement les sites ouverts et fermés. Un tableau de pointeurs d'entiers (de dimension m) donne enfin l'adresse de chaque site dans l'un ou l'autre des tableaux précédents de sorte que l'application d'un mouvement s'effectue en $O(1)$. La composante de Mouvement, appelé `Swap`, dont la programmation de la fonction `move` est triviale, est paramétrée par les indices f_{in} et f_{out} correspondant au site à fermer et au site à ouvrir. On suppose également qu'une Caractéristique, notée `Closest`, maintient pour chaque client `u` les deux sites ouverts dans la solution courante qui lui sont les plus proches, notés $\texttt{f}_1\texttt{[u]}$ et $\texttt{f}_2\texttt{[u]}$. Des études expérimentales menées par Resende et Werneck (2002) ont montré que les fonctionnalités associées à ces quatre composantes prennent une part négligeable du temps de calcul total de sorte qu'il n'est pas indispensable d'en proposer des versions optimisées. En particulier, dans la fonction `track` de `Closest`, on peut se contenter, pour chaque client `u` lésé par un mouvement, de recalculer $\texttt{f}_1\texttt{[u]}$ et $\texttt{f}_2\texttt{[u]}$ par une boucle explicite sur l'ensemble des sites ouverts.

Recherche séquentielle du meilleur mouvement. Dans la première approche, on constate simplement que le gain d'un mouvement s'écrit :

$$\texttt{value}_{f_{in},f_{out}} = \sum_{u|f_1[u]\neq f_{out}} [d_{u,f_1[u]} - d_{u,f_{in}}]^+ - \sum_{u|f_1[u]=f_{out}} [min\{d_{u,f_2[u]}, d_{u,f_{in}}\} - d_{u,f_1[u]}].$$

Pour implanter la sélection du meilleur mouvement, il suffit de programmer un Serveur itérant selon un schéma arbitraire sur les coordonnées $f_{in} \leftrightarrow f_{out}$ possibles. Pour cela, les deux fonctions `begin` et `next` composent le Mouvement Swap, la Solution Subset ainsi que le Problème PmedDst. La fonction `scan` de l'Attribut de gain calcule enfin la formule précédente en fonction du Mouvement Swap, de la Caractéristique `Closest` et du Problème `PmedDst`. À noter que `scan` opère en $O(n)$ et que $p(m-p)$ mouvements doivent être visités, ce qui donne une complexité de $O(nmp)$ par itération.

Recherche incrémentale du meilleur mouvement. Dans la deuxième approche, on exploite le fait que la formule de gain précédente est relativement identique d'une itération à l'autre. Dans la Caractéristique `PmedAllGain` présentée à la figure 6.2, on propose donc d'élever un tableau de dimension $(m-p) \times p$ dans lequel sont maintenus tous les gains de mouvement. Comme les lignes et colonnes sont susceptibles de représenter différents sites d'une itération à l'autre, trois tableaux auxiliaires, `in`, `out` et `position`, de dimensions $m-p$, p et m, sont maintenus de sorte que `value[in[i]][out[j]]` représente le gain du mouvement remplaçant le site de la ligne i par celui de la colonne j. Le tableau `position` indique la ligne ou colonne où apparaît chaque site. Dans la fonction `track`, on note $\texttt{f}'_1\texttt{[u]}$ et $\texttt{f}'_2\texttt{[u]}$ les valeurs que prendraient $\texttt{f}_1\texttt{[u]}$ et $\texttt{f}_2\texttt{[u]}$ suite à l'application du mouvement `swap` de coordonnées $f_{in} \leftrightarrow f_{out}$. La fonction `track` comprend quatre étapes. Dans la première, on corrige l'apport des termes fluctuants dans la première somme de la formule de gain en supposant que les bornes des sommes restent inchangées. Dans la deuxième, on tient compte des changements relatifs aux bornes des deux sommes dans la formule de gain. Dans la troisième, on corrige l'apport des termes fluctuants dans la deuxième somme de la formule de gain. Notons qu'aucune des trois étapes précédentes ne touche à la ligne et la colonne correspondant au mouvement $f_{in} \leftrightarrow f_{out}$ à appliquer. En effet, les mouvements qui y sont représentés changent à l'itération suivante de sorte qu'on doit construire et non corriger leur valeur de gain, ce qui est fait dans la quatrième étape. On notera qu'en utilisant des formules de transitivité du type $\texttt{value}^{S'}_{f_{out},k} = \texttt{value}^{S}_{f_{in},k} - \texttt{value}^{S}_{f_{in},f_{out}}$, où S désigne la solution courante et S' la solution après application de `swap`, le calcul de chacun de ces gains peut en fait être réalisée en $O(1)$. Pour finir, la maintenance des trois tableaux auxiliaires consiste simplement à permuter $\texttt{in[position[}f_{in}\texttt{]]}$ et $\texttt{out[position[}f_{out}\texttt{]}$ ainsi que $\texttt{position[}f_{in}\texttt{]}$ et $\texttt{position[}f_{out}\texttt{]}$.

```
struct PmedAllGain : BuildWith<Subset, Closest, PmedDst>,
                     TrackWith<Closest, PmedDst>
{
  int** value, *in, *out, *position;

  PmedAllGain(const Subset&, const Closest&, const PmedDst&) { ... }
  ~PmedAllGain() { ... }

  void track(const Swap& swap, const Closest& closest, const PmedDst& dst)
  {
    Pour toute ligne i, initialiser correction[i] à 0
    Pour tout client u tel que f'_1[u] ≠ f_1[u]
      Pour toute ligne i de site f ≠ f_in tel que d_{u,f} ≤ max(d_{u,f'_1[u]}, d_{f_1[u]})
        correction[i] += [d_{u,f'_1[u]} − d_{u,f}]^+ − [d_{u,f_1[u]} − d_{u,f}]^+
    Pour toute ligne i, ajouter correction[i] en chaque colonne j de site f ≠ f_out

    Pour tout client u tel que f'_1[u] ≠ f_1[u] et sur toute ligne de site f ≠ f_in
    {  Si f_1[u] ≠ f_out
         Ajouter [d_{u,f_1[u]} − d_{u,f}]^+ + min{d_{u,f_2[u]}, d_{u,f}} − d_{u,f_1[u]} à la colonne de site f_1[u]
       Si f'_1[u] ≠ f_out
    }    Enlever [d_{u,f'_1[u]} − d_{u,f}]^+ + min{d_{u,f'_2[u]}, d_{u,f}} − d_{u,f'_1[u]} à la colonne de site f'_1[u]

    Pour tout client u tel que f'_2[u] ≠ f_2[u], f'_1[u] = f_1[u] et f_1[u] ≠ f_out
      Sur toute ligne de site f ≠ f_in tel que d_{u,f} ≤ max(d_{u,f'_2[u]}, d_{f_2[u]})
        Ajouter min{d_{u,f_2[u]}, d_{u,f}} − min{d_{u,f'_2[u]}, d_{u,f}} à la colonne de site f_1[u]

    Mettre à jour la ligne de site f_in et la colonne de site f_out
  }  Mettre à jour les tableaux in, out et position
};
```

Figure 6.2 – Programmation de la Caractéristique `PmedAllGain`

Notons que dans le pire cas, la fonction `track` s'exécute en $O(nm)$, mais que dans une approche métaheuristique où beaucoup de temps est passé près de solutions quasi-optimales, relativement peu de clients `u` vérifient $\mathtt{f}'_1[\mathtt{u}] \neq \mathtt{f}_1[\mathtt{u}]$ et peu de sites `f` vérifient $d_{u,f} \leq max(d_{u,f'_1[u]}, d_{f_1[u]})$. Signalons aussi qu'il serait assez simple de contruire une Caractéristique où les mouvements seraient triés par ordre de gain, par exemple dans une table de hachage, comme nous l'avons déjà fait pour le problème

SAT et comme nous le proposerons pour accélérer la résolution du *PVC* au chapitre 7. L'utilisation d'une matrice de taille $O(mp)$ est en revanche un inconvénient comparée aux techniques s'appuyant sur la factorisation du gain.

6.3 Le problème de coloration de graphe

Étant donné un graphe $G = (V, E)$ non dirigé, une k-coloration des sommets consiste à affecter une parmi k couleurs à chaque sommet de sorte qu'aucun sommet adjacent ne comporte la même couleur. Bien que le problème de coloration de graphe consiste à trouver une k-coloration utilisant le moins de couleurs possibles, la plupart des méthodes de recherche locale proposées dans la littérature travaillent en fait sur la recherche d'une k-coloration, et lorsqu'elle est trouvée, décrémentent d'une unité le nombre de couleurs disponibles puis recommencent la procédure. On s'intéresse ici à l'implantation efficace d'un algorithme tabou fréquemment proposé, initialement dû à Hertz et de Werra (1987). Selon cette approche, la fonction à minimiser correspond au nombre d'arêtes reliant deux sommets de même couleur et un mouvement est défini par la modification de la couleur d'un sommet en conflit, c'est-à-dire partageant une même couleur avec l'un de ses voisins. Si `v` est un sommet en conflit, on notera `(v,i)` les coordonnées du mouvement affectant la couleur `i` à `v`.

On suppose pour la suite qu'on dispose d'une composante de Problème dans laquelle `N`, `M` et `k` donnent le nombre de sommets, d'arêtes et de couleurs du problème, `degree[v]`, le nombre de sommets adjacents à `v`, et `adjacent[v][k]`, le $k^{\text{ième}}$ sommet adjacent à `v`. On suppose aussi que la composante de Solution, notée `Coloring`, contient un tableau `color` de taille `N`, où `color[v]` est la couleur du sommet `v`.

Recherche du meilleur mouvement. Nous proposons ici d'examiner en détail les composantes intervenant dans le choix efficace d'un meilleur mouvement. Remarquons que la recherche naïve d'un meilleur mouvement consisterait, pour chaque

sommet en conflit, à évaluer le gain induit sur la fonction coût pour chacune des k-1 couleurs alternatives, ce qui demanderait chaque fois un temps proportionnel au degré du sommet réaffecté. Afin d'accélérer ce calcul, qui en pratique est le plus coûteux, la littérature propose de maintenir pour chaque sommet `v` le nombre de voisins de `v` de couleur `i`. La Caractéristique `AdjColDeg` de la figure 6.3 implante cette approche. Elle contient une matrice `value` de taille `N` × `k`, où `value[v][i]` indique le nombre de sommets de couleur `i` adjacents à `v`. Elle maintient également le nombre de sommets en conflit, noté `critic_nbr`. Pour chaque sommet adjacent au sommet `mvt.v` recolorié, la fonction `track` modifie simplement les deux seuls champs affectés dans la matrice `value` et lorsque nécessaire met à jour `critic_nbr`.

```
struct AdjColDeg : BuildWith<Coloring, GraphColPbm>,
                   TrackWith<Recoloring, Coloring, GraphColPbm>
{
  int critic_nbr, **value;

  AdjColDeg(const Coloring& sol, const GraphColPbm& pbm) { ... }
  ~AdjColDeg() { ... }

  void track(const Recoloring& mvt, const Coloring& sol, const GraphColPbm& pbm)
  {
    for (int k=0; k<pbm.degree[mvt.v]; k++)
    {
      int w = pbm.adjacent[mvt.v][k];
      if (++value[w][mvt.i]==1 && mvt.i ==sol.color[w]) critic_nbr++;
      if (--value[w][sol.color[mvt.v]]==0 && mvt.i==sol.color[w]) critic_nbr--;
    }
  }
};
```

Figure 6.3 – Programmation de la Caractéristique `AdjColDeg`

```
struct ColGain : ScanWith<Recoloring, Coloring, AdjColDeg>
{
  int value;
  void scan(const Recoloring& mvt, const Coloring& sol, const AdjColDeg& acd)
  {
    value = acd.value[mvt.v][mvt.i] - acd.value[mvt.v][sol.color[mvt.v]];
  }
};
```

Figure 6.4 – Programmation de l'Attribut `ColGain`

Comme le montre la figure 6.4 avec l'Attribut `ColGain`, l'évaluation du gain d'un mouvement se calcule alors en $O(1)$ par la différence entre les degrés de coloration autour de `v` pour les couleurs après et avant mouvement.

Également, comme l'illustre la figure 6.5 avec le Serveur `ConflictNodeSvr`, l'énumération du voisinage revient à détecter dans la matrice de `ColAdjDeg` les sommets `v` pour lesquels `value[v][sol.color[v]]` est non nul. On énumère alors l'ensemble des recolorations possibles pour chacun de ces sommets.

```
struct ConflictNodeSvr
 : BuildWith<>, EnumerateWith<Recoloring, Coloring, AdjColDeg, GraphColPbm>
{
  void begin(Recoloring& mvt, const Coloring& sol,
             const AdjColDeg& acd, const GraphColPbm& pbm)
  {
     for (mvt.v=0; acd.value[mvt.v][sol.color[mvt.v]]==0; v++);
     mvt.i = (sol.color[mvt.v] == 0) ? 1 : 0;
  }

  bool next(Recoloring& mvt, const Coloring& sol,
            const AdjColDeg& acd, const GraphColPbm& pbm)
  {
     if (++mvt.i == sol.color[mvt.v]) mvt.i++;
     if (mvt.i != pbm.k) return true;
     do { mvt.v++; }
     while (mvt.v < pbm.N && acd.value[mvt.v][sol.color[mvt.v]]==0);
     if (mvt.v == pbm.N) return false;
     mvt.i = (sol.color[mvt.v] == 0) ? 1 :0;
     return true;
  }
};
```

Figure 6.5 – Programmation du Serveur `ConflictNodeSvr`

Liste taboue à longueur variable. La résolution par la méthode taboue du problème de coloration de graphe s'appuie souvent sur une liste taboue à longueur variable. Cette longueur est généralement formulée par la somme d'un terme fixe et d'un terme proportionnel au nombre de sommets en conflit. Nous montrons ici comment découpler la programmation d'une Caractéristique implantant une liste taboue

à longueur variable de la programmation d'une Caractéristique calculant la longueur de cette liste. En procédant ainsi, la liste taboue devient réutilisable pour d'autres problèmes dont on souhaiterait faire varier la longueur de la liste taboue. Également, il devient possible de greffer différentes gestions de longueur sur la liste implantée.

```
template <class ATB, class TEN>
struct ByIterRecency : BuildWith<int>, TrackWith<ATB, TEN>
{
  Array<int> last_iter;
  max_iter, cur_iter;

  ByIterRecency(const TEN& ten, int sz)
    : last_iter(sz), max_iter(-ten), cur_iter(0)
  { for (int i=0; i<sz; i++) last_iter[i] = std::numeric_limits<int>::min(); }

  void track(const ATB& atb, const TEN& tenure)
  {
    last_iter[atb.value] = cur_iter++;
    max_iter = cur_iter - tenure.value;
  }

  template <class BTA> bool intercept(const BTA& bta)
  {
    return (last_iter[bta.value] > max_iter);
  }
};
```

Figure 6.6 – Programmation de la Caractéristique ByIterRecency

L'implantation de la liste taboue `ByIterRecency` présentée à la figure 6.6 est similaire à celle du chapitre 5 à ceci-près que les Attributs enregistrés dans cette liste doivent être des Attributs Singletons.

```
struct FrCol : ScanWith<Recoloring, Coloring, PmedDst>
{
  int value;

  void scan(const Recoloring& mvt, const Coloring& sol, const PmedDst& pbm)
  {
    value = pbm.N*mvt.v + sol.color[mvt.v];
  }
};
```

Figure 6.7 – Programmation de l'Attribut `FrCol`

De même qu'au chapitre 5, on suppose que les champs d'Attributs enregistrés ou testés sont entiers. Dans le cas de la coloration de graphe, on introduit à cet effet les deux Attributs `FrCol` et `ToCol` qui transportent les couples (`mvt.v`, `sol.color[mvt.v]`) et (`mvt.v`, `mvt.i`) sur des indices biunivoques dans le tableau `value` de la liste taboue. `FrCol` est par exemple présenté figure 6.7.

```
struct AffinColTenure : BuildWith<AdjColDeg, int, int>, TrackWith<AdjColDeg>
{
  int value, T, lambda;

  AffinColTenure(const AdjColDeg& acd, int t, int l)
     : T(t), lambda(l), value(t + l * acd.critic_nbr)
  {}

  void track(const AdjColDeg& acd) { value = T + lambda * acd.critic_nbr; }
};
```

Figure 6.8 – Programmation de la Caractéristique `AffinColTenure`

On donne finalement à la figure 6.8 la Caractéristique `AffinColTenure` qui maintient la longueur de la liste taboue. Rappelons que la liste taboue de la figure 6.6 est découplée de la maintenance de la longueur de liste. Il serait ici par exemple possible de définir une nouvelle Caractéristique, `AutoColTenure`, dans laquelle on détecterait les cycles sur la trajectoire des solutions en surveillant la fréquence à laquelle apparaissent des valeurs de signature associée à chaque solution. Par exemple, étant donné un poids associé à chaque sommet, la signature d'une solution pourrait être définie comme la somme des produits de poids et de couleurs sur chaque sommet. À noter qu'à chaque itération, cette signature pourrait être maintenue en temps constant. Dans `AutoColTenure`, on maintiendrait alors la fréquence de chaque signature possible depuis le dernier changement de longueur de liste. Lorsqu'une fréquence dépasse un certain seuil, la longueur de la liste est augmentée alors qu'elle est diminuée si chaque fréquence reste sous ce seuil depuis un certain nombre d'itérations.

6.4 Le problème d'affectation quadratique

On se donne un ensemble de n sites et un ensemble de n activités communiquant entre elles. Le flux échangé entre deux activités est défini par une matrice f_{ij} et le coût de communication par unité de flux entre deux sites est défini par une matrice de distance d_{pq}. Le problème d'affectation quadratique consiste à affecter une activité à chaque site de sorte que le coût total de communication entre activités soit minimisé. Parmi toutes les permutations π définies sur $\{1..n\}$, on cherche donc à minimiser $\sum_{i=1}^{n}\sum_{j=1}^{n} f_{ij}d_{\pi(i)\pi(j)}$.

Un mouvement souvent utilisé pour résoudre le problème d'affectation quadratique consiste à permuter les emplacements de deux activités données de la solution courante. La valeur de gain d'un mouvement permutant les emplacements des activités `u` et `v` est alors donnée par la formule $\texttt{value}_{u,v} = 2\sum_{j=1,j\neq u,j\neq v}^{n}(f_{uj} - f_{vj})(d_{\pi(v)\pi(j)} - d_{\pi(u)\pi(j)})$. Pour la suite, on suppose qu'on dispose d'une composante de problème contenant les matrices de flux `f[i][j]` et de distance `d[i][j]`. Également, la composante de solution contient les tableaux `loc` et `elt` où `loc[e]` donne l'emplacement de l'activité `e` dans la solution courante et `elt[l]` donne l'activité située à l'emplacement `l`. Le Mouvement, dont la programmation de `move` est triviale, est paramétré par le jeu de coordonnées `u`, `v` des deux activités dont on permute les emplacements.

Nous nous limitons dans cette section à examiner en détail la programmation efficace du calcul de gain de chaque mouvement. Pour ce faire, remarquons que lorsque le mouvement de coordonnées (`x`, `y`) est exécuté, la valeur de gain du mouvement de coordonnées (`u`, `v`) avec `u` et `v` différents chacun de `x` et `y` doit être incrémentée de :

$$2(\mathrm{f}_{ux} + \mathrm{f}_{vy} - \mathrm{f}_{uy} - \mathrm{f}_{vx})(d_{\pi(v)\pi(y)} + d_{\pi(u)\pi(x)} - d_{\pi(u)\pi(y)} - d_{\pi(v)\pi(x)}).$$

De même que pour le problème de la p-médiane, il est ici naturel de maintenir dans une Caractéristique, notée `QapAllGain` et présentée à la figure 6.9, l'ensemble des

valeurs de gain dans un tableau `value` de dimension n^2. Dans un premier temps, la fonction `track` réajuste selon la formule d'incrémentation les valeurs de gain pour les mouvements dont les coordonnées n'interceptent pas celles de `mvt`, et dans un deuxième temps, réétablit explicitement le gain des mouvements restants. La complexité obtenue est quadratique car $O(n^2)$ valeurs sont corrigés par incrémentation, chacune en $O(1)$, et $O(n)$ valeurs sont recalculées explicitement, chacune en $O(n)$.

```
struct QapAllGain : BuildWith<QapAssignment, QapMatrix>,
                    TrackWith<QapSwap, QapAssignment, QapMatrix>
{
  int** value;

  QapAllGain(const QapAssignment& sol, const QapMatrix& pbm) { ... }
  ~QapAllGain() { ... }

  void track(const QapSwap& mvt, const QapAssignement& sol, const QapMatrix& pbm)
  {
    Pour tout i ≠ mvt.i et mvt.j et pour tout j ≠ mvt.i et mvt.j
      Ajuster value[i][j] selon la formule d'incrément

    Ajuster les valeurs de gain des mouvements partageant au moins une coordonnée avec mvt
  }
};
```

Figure 6.9 – Programmation de la Caractéristique `QapAllGain`

6.5 Conclusion

Dans ce chapitre, nous avons illustré l'expressivité de la librairie METALAB pour différents problèmes de la littérature, et pour chacun d'eux, avons examiné en détail la programmation des composantes relevant le plus d'intérêt. Soulignons que contrairement aux cadres d'application présentés au chapitre 2 et ne permettant que de programmer des versions très standardisées de recherche locale, nous avons chaque fois proposé des implantations parmi les meilleures présentées dans la littérature. La résolution de nouveaux problèmes au moyen de la librairie METALAB, par exemple le problème de confection de tournois (Nemhauser et Trick, 1998) ou le problème de clique de cardinalité maximale (Gendreau, Soriano et Salvail, 1993) sont en projet.

CHAPITRE 7 : ÉTUDE DE CAS, MÉTHODES TABOUES POUR LE *PVC*

Comparativement aux algorithmes en première amélioration basés sur les mouvements 2-*opt*, 3-*opt* ou dérivés de Lin et Kernighan (1973), les méthodes avec critère tabou ne constituent pas à ce jour une approche de recherche locale privilégiée pour la résolution du problème du voyageur de commerce (Johnson, McGeoch, 1997). Comme nous le verrons, une des raisons principales provient de ce que chaque itération exige d'énumérer explicitement le voisinage alors que dans le cas de méthodes basées sur une première amélioration, des techniques spécifiques non applicables au cas d'une descente en meilleure amélioration permettent d'accélérer significativement la recherche d'un mouvement satisfaisant. Après avoir comparé l'implantation d'une descente en première amélioration munie des techniques classiques d'accélération à l'approche en meilleure amélioration standard, nous proposons dans ce chapitre plusieurs avenues de recherche qui permettent de perfectionner la sélection d'un mouvement 2-*opt* dans le strict cadre de la méthode taboue. À notre connaissance, aucune de ces méthodes n'a été présentée dans la littérature. Nous proposons successivement trois améliorations. La première, appelée technique d'incrémentation du voisinage, consiste à ordonner dans une structure l'ensemble du voisinage courant selon le gain de chaque mouvement. Cette approche, qui a été utilisée dans d'autres contextes (en particulier pour le problème *SAT* au chapitre 5), est avantageuse lorsque la maintenance de la structure à chaque itération est moins coûteuse que l'énumération explicite du voisinage. La deuxième technique, appelée segmentation du voisinage, a été motivée par l'observation que la méthode taboue procède en fait la plupart du temps à des mouvements qui inversent des sous-séquences de très petit diamètre comparativement au nombre N de villes constituant le problème. L'idée consiste alors à alterner une recherche se limitant le plus souvent à la sélection du meilleur mouvement parmi ceux correspondant à de petits diamètres et tantôt à forcer le choix d'un

meilleur mouvement parmi ceux présentant un gros diamètre. La dernière technique, appelée technique d'anticipation (en anglais *lookahead*) emprunte à la programmation par contraintes l'idée de tester partiellement l'effet ultérieur d'une itération de l'algorithme sur la trajectoire de recherche vis-à-vis de la fonction à optimiser et/ou des contraintes à vérifier. Dans le cadre de la méthode taboue, plutôt que de sélectionner le meilleur mouvement, elle vise à choisir celui qui se situe sur le chemin menant à la meilleure solution accessible en un petit nombre de mouvements depuis la solution courante. Mentionnons que la réalisation et l'expérimentation des algorithmes que nous présentons a été faite au moyen de la librarie METALAB. Le présent chapitre illustre à ce titre l'intérêt du logiciel pour la conduite de projets non triviaux.

7.1 Contexte

Considérons dans cette section une recherche locale basée sur un 2-*opt* en première amélioration n'utilisant aucun critère métaheuristique, l'algorithme se terminant donc lorsque la tournée courante est 2-*opt* optimale. Rappelons que dans ce contexte, il a été observé empiriquement que le nombre d'itérations exécutées par la recherche locale est de l'ordre de $O(N)$ (Johnson et McGeoch, 1997). Dans le cas d'une implantation triviale où la tournée est représentée sous forme de tableau (comme proposé au chapitre 5 à la figure 5.3) et où les mouvements sont énumérés sans technique particulière, notons que chaque itération comprend dans le pire cas $O(N^2)$ observations de mouvements, chacune fonctionnant en temps constant, et $O(N)$ modifications dans le tableau représentant la tournée. La littérature a proposé à cet égard trois améliorations principales : l'utilisation de listes de proximité, c'est-à-dire ordonnées par plus proches voisins, l'enregistrement du statut de chaque liste, et enfin, la représentation des tournées sous forme d'arbre.

Listes de proximité. La première technique d'accélération, proposée par Steiglitz et Weiner (1968) repose sur le fait qu'un mouvement améliorant impliquant les arêtes

$\{a, b\}$ et $\{c, d\}$ vérifie forcément $dst(a, b) > dst(a, c)$ et/ou $dst(c, d) > dst(d, b)$. Étant donnée une orientation de la tournée courante, la recherche d'un mouvement améliorant peut donc se limiter à examiner pour chacun de ses arcs (x, y) l'ensemble des mouvements qui connecte x à une ville plus proche que ne l'est y. Selon cette procédure, notons que bien qu'il soit possible d'oublier un mouvement satisfaisant lors du traitement de l'arc (a, b), ce même mouvement sera forcément détecté lors du traitement de l'autre arc (c, d) impliqué dans le mouvement. En pratique, l'énumération peut être effectuée efficacement en construisant en début d'algorithme et pour chaque ville une liste des autres villes par ordre de proximité. Pour chaque arc (x, y) de la tournée, il suffit alors de traverser la liste de x jusqu'à ce que la ville y soit rencontrée. Cette stratégie est d'autant plus efficace qu'au cours de l'algorithme, les sommets voisins sur la tournée courante deviennent de moins en moins distants. Des études empiriques ont montré que le temps de sélection d'un mouvement est ainsi réduit en moyenne, sur l'ensemble des itérations, de $O(N^2)$ à $O(N)$ par itération (Johnson et McGeoch, 1997). Si cette accélération n'est pas suffisante, il est enfin raisonnable d'approximer la recherche d'un mouvement améliorant en utilisant des listes tronquées pour des tailles variant entre $0.4N$ à $0.8N$. Dans le contexte de la librairie METALAB, signalons que les listes de proximités seraient programmées dans la structure de Problème passée en argument à l'algorithme. Un Serveur circulerait alors sur les arcs de la tournée, et pour chaque arc traverserait la liste de proximité du sommet origine jusqu'à ce que le sommet destination soit atteint. Pour chaque mouvement induit par ce schéma d'énumération, l'Attribut `TspGain` défini au chapitre précédent serait ensuite passé à l'Explorateur `FirstImprovement` à partir duquel serait généré l'algorithme.

Enregistrement des listes inutiles. La deuxième technique d'accélération, proposée par Bentley (1992), consiste à retenir quelles listes n'ont précédemment pas permis de trouver de mouvements améliorants. En effet, si une liste, associée par

exemple à la ville a n'a pas permis de trouver un mouvement améliorant, il n'est pas rentable de revisiter cette liste tant que les voisins de a restent inchangés. Au moyen de cette seconde technique, il a été observé que le nombre total de mouvements évalués en cumulant toutes les itérations est de l'ordre de $O(N)$ pour une large constante. Dans METALAB, l'enregistrement des listes inutiles serait programmé au moyen d'une Caractéristique comme suit. En phase d'initialisation, toutes les listes sont activées. Chaque fois qu'un mouvement est appliqué, les listes correspondant aux villes changeant de voisin sont réactivées. Également, les listes ayant été traversées sans succès, qu'on peut déduire des coordonnées du mouvement sélectionné et de la solution courante, sont désactivées. Cette Caractéristique est finalement composée par un Serveur similaire au précédent mais sautant les listes de proximité inutiles. Le reste de l'algorithme demeure inchangé.

Représentations arborescentes des tournées. Lorsque les deux techniques précédentes sont exploitées, la modification de la tournée courante selon les mouvements sélectionnés devient finalement l'opération la plus coûteuse. Il a en effet été observé empiriquement qu'au cours de la descente, la plus courte des deux sous-séquences qu'on peut inverser pour réaliser le mouvement sélectionné est en moyenne, sur l'ensemble des itérations, de l'ordre de $O(N^{0.75})$ (Bentley, 1992). Dans ce contexte, trois représentations fondées sur des arbres et permettant respectivement d'inverser une sous-séquence en $O(N \log N)$ (mais avec une très grande constante), $O(N^{0.5})$ et $O(N)$ (mais avec un bon comportement en moyenne) ont alors été proposées (Fredman, Johnson, McGeoch et Ostheimer, 1995). Des études empiriques ont montré que la représentation par tableau est dominée à partir de N compris entre 10^3 et 10^4. Comme nous verrons dans la section 7.3, ces représentations par arbre ne sont en fait pas indispensables dans le contexte des méthodes taboues car le diamètre moyen des sous-séquences qu'inversent les mouvements est beaucoup plus petit que dans la phase de descente pure préliminaire. Notons que si l'on souhaitait rendre compatibles les structures présentées au chapitre précédent pour le PVC aux différentes représentations

de tournées possibles, il faudrait non seulement les définir génériquement par rapport à la structure de distance `Dst`, comme c'était déjà le cas, mais également par rapport à la structure `Tr` implantant la Solution. En admettant que toute tournée `Tr` fournisse les opérations `succ` et `pred`, seule la fonction `move` du Mouvement `Twex<Dst,Tr>` devrait alors être reprogrammée. Les autres structures, comme par exemple `TspGain`, `TspVal` et `ToAdj`, seraient directement compatibles.

Le cas de la méthode taboue. Si l'on compare au contexte algorithmique précédent, la méthode taboue semble à plusieurs titres inefficace. Par exemple, Johnson et McGeoch (1997) reprochent principalement à la méthode taboue, d'une part, un trop grand nombre d'itérations relativement au nombre $O(N)$ d'itérations d'une descente en première amélioration, et d'autre part, l'impossibilité d'accélérer substantiellement la recherche en $O(N^2)$ du meilleur mouvement possible à chaque itération. Il est bien évident que des méthodes taboues oeuvrant alors entre $O(N^4)$ et $O(N^6)$ selon le nombre d'itérations effectués sont de peu d'utilité comparativement aux algorithmes précédents oeuvrant en $O(N)$ pour le cas du mouvement 2-*opt* et que des techniques de relance permettent ensuite de réitérer. Une des tentatives visant à améliorer le sort de la méthode taboue (Troyon, 1988) consiste d'ailleurs à initier chaque itération par la recherche d'un mouvement en première amélioration et lorsqu'elle est infructueuse de finalement rechercher le mouvement le moins dégradant parmi cN nouveaux mouvements énumérés. Cette projection de la méthode taboue sur une descente en première amélioration traduit l'impossibilité d'utiliser telles quelles les techniques de liste de proximité précédentes. Johnson et McGeoch (1997) ont également suggéré d'éviter de revisiter les mouvements invariants, à savoir, ceux dont les arcs sortants se situent tous deux soit à l'intérieur soit à l'extérieur de la sous-séquence inversée par le mouvement courant. Puisque la taille de la plus courte sous-séquence impliquée dans un 2-*opt* est de l'ordre de $O(N^{0.75})$, cela se traduit malheureusement par un gain de l'ordre $O(N^{0.25})$ par rapport à la complexité initiale en $O(N^2)$.

7.2 Incrémentation du voisinage 2-*opt*

Une première amélioration significative pour l'énumération exhaustive du voisinage 2-*opt* consiste à maintenir une structure de données où sont ordonnés les mouvements selon leur coût. Cette technique, qui relève de l'incrémentation du voisinage, constitue une des techniques classiques de recherche locale dont le chapitre 5 a par exemple fourni une implantation pour le problème SAT. Dans le contexte du chapitre 2, ces techniques ont aussi un lien étroit avec la conception des langages déclaratifs. Notons qu'il existe parfois plusieurs niveaux d'incrémentation : pour le problème SAT, par exemple, la maintenance du nombre de littéraux vérifiés pour chaque clause constitue un premier degré d'incrémentation alors que la maintenance de tous les mouvements *Flip* dans une table de hachage selon leur gain constitue un deuxième niveau d'incrémentation. En général, les techniques d'incrémentation sont avantageuses tant que la maintenance à chaque itération des structures qu'on a fait apparaître est moins coûteuse que les calculs explicites qui devraient être faits sans ces structures.

Extention du voisinage 2-*opt*. Dans le cas du mouvement 2-*opt*, l'incrémentation de l'énumération du voisinage n'a sans doute pas été à ce jour bien explorée car plutôt que de considérer les mouvements invariants comme ceux possédant leurs deux arêtes sortantes soit à l'intérieur soit à l'extérieur de la sous-séquence qu'inverse le mouvement courant (comme il était suggéré en fin de section précédente), il est en fait bien plus judicieux de considérer chaque mouvement en double, l'un correspondant au 2-*opt* valide, et l'autre, au mouvement qui créerait deux sous-tours. Pour deux arêtes a_1 et a_2 appartenant à la tournée courante, on note ici $(a_1, a_2)^+$ les coordonnées du mouvement valide et $(a_1, a_2)^-$ celles du mouvement invalide. La figure 7.1 illustre le changement de validité des mouvements suite à l'application d'un 2-*opt*. À gauche, les deux mouvements du couple 2-*opt*$(\{1,2\},\{3,4\})^{\pm}$ ne permutent pas de statut car les deux arêtes sortantes sont situées sur la même sous-séquence relativement aux arêtes

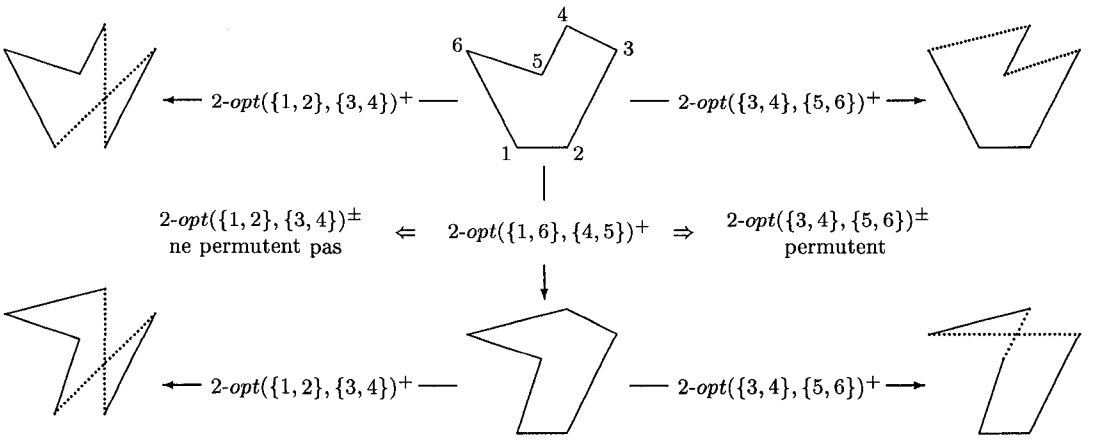


Figure 7.1 – Variation du voisinage 2-*opt*

$\{1,6\}$ et $\{4,5\}$ enlevées par le mouvement 2-*opt*$(\{1,6\},\{4,5\})^+$ qu'on applique. À droite par contre, les deux mouvements du couple 2-*opt*$(\{3,4\},\{5,6\})^\pm$ permutent. Notons que mises à part les permutations concernant la notation $\pm$ des mouvements, la seule variation du voisinage étendu suite à l'application d'un mouvement concerne d'une part l'ajout des mouvements qui enlèvent une des deux arêtes entrantes (ici $\{4,6\}$ et $\{1,5\}$), et d'autre part, la suppression des mouvements qui enlèvent une des deux arêtes sortantes (ici $\{1,6\}$ et $\{4,5\}$).

Maintenance de la table de hachage. La maintenance d'une table de hachage où sont ordonnés tous les mouvements du voisinage étendu devient marginale puisque seuls les couples de mouvements qui utilisaient un des deux arcs sortants doivent être effacés et seuls les couples de mouvements qui utilisent un des deux arcs entrants doivent être introduits. Comme dans une table de hachage chaque opération d'enregistrement et d'effacement peut-être effectuée en temps constant, la maintenance de la table à chaque itération prend un temps proportionnel au nombre $O(N)$ de ces couples de mouvements à supprimer et à introduire. Notons qu'étant donnée une orientation du tour et pour une notation judicieuse des mouvements (cf. Annexe

B), il est enfin possible de vérifier en temps constant le statut valide ou invalide d'un mouvement de sorte qu'il est inutile de maintenir à chaque itération les $O(N^2)$ changements de statut possibles.

Traversée de la table de hachage. La traversée de la table de hachage jusqu'au premier mouvement valide constitue théoriquement le maillon faible de l'approche proposée. Si on note D la distance maximale entre deux sommets quelconque du problème, la table de hachage contient $O(D)$ lignes d'enregistrement qu'il faut potentiellement traverser. Néanmoins, comme le montre le tableau 7.1, le temps passé à traverser la table de hachage reste en pratique largement négligeable par rapport à la maintenance de la structure.

Les deux premières colonnes du tableau indiquent les caractérisques des problèmes considérés. Tous ces problèmes appartiennent à la librairie TSPLIB pour des tailles allant de 101 à 2392 villes. Les temps de calculs sont mesurés en microsecondes ($10^{-6}s$) et constituent chaque fois une moyenne sur un ensemble de 100000 itérations consécutives. Les mesures ont été par ailleurs effectuées pour des solutions initiales différentes obtenues selon l'algorithme de Clarke & Wright (CW), selon l'algorithme des plus proches voisins (PPV) et enfin générées aléatoirement (RD). Les colonnes de maintenance comptabilisent le temps CPU œuvrant à la mise à jour de la table à chaque itération alors que les colonnes de traversées comptabilisent le temps passé à traverser la table jusqu'à l'obtention d'un mouvement valide non tabou. À titre d'indication, le temps CPU passé à chercher le meilleur mouvement explicitement est indiqué dans la troisième série de colonnes.

Deux propriétés expliquent en partie le bon comportement de la table de hachage en phase de traversée. Premièrement, à chaque opération d'effacement et d'enregistrement, il est possible de maintenir en temps constant la meilleure ligne d'enregistrement non vide, de sorte que lors de la traversée de la table de hachage, on parte

Tableau 7.1 – Performances et comportement de la table de hachage

Problème		*Temps CPU par itération (en μs)*									*Dominance des 2-opt$^-$*					
Nom	*Taille*	*Maintenance*			*Traversée*			*Explicite*			*#2-opt$^-$*			$\frac{\#\text{2-opt}^-}{N}$		
		CW	*PPV*	*RD*	*CW*	*PPV*	*RD*	*CW*	*PPV*	*RD*	*CW*	*PPV*	*RD*	*CW*	*PPV*	*RD*
eil	101	91.1	87.1	83.0	3.0	3.1	2.9	349	331	307	16.9	18.5	17.2	0.17	0.18	0.17
ch	150	200	179	185	6.22	5.4	4.6	736	711	707	32.2	28.8	24.4	0.21	0.19	0.16
gr	202	370	388	374	14.6	15.1	24.9	1259	1304	1353	41.4	49.5	45.6	0.21	0.25	0.23
pcb	442	1454	1447	1395	50.0	49.7	53.7	8093	8242	7695	69.0	74.1	77.0	0.16	0.17	0.17
gr	666	2752	2771	2707	104	105	106	1.7e4	1.8e4	1.7e4	126	130	138	0.19	0.19	0.21
pr	1002	4094	4362	4485	149	148	157	4.1e4	4.4e4	4.3e4	195	182	199	0.19	0.18	0.20
pr	2392	1.3e4	1.2e4	1.3e4	436	383	430	3.2e5	3.3e5	3.8e5	436	428	454	0.18	0.18	0.19

directement de la meilleure ligne d'enregistrement non vide. Dans ce contexte, le facteur prépondérant en phase de traversée devient le nombre de mouvements qu'on traverse et qui sont invalides, car ceux-ci, à l'approche de tournées quasi-optimales, ont tendance à dominer les mouvements valides, ce qui force à continuer la traversée. À cet égard, les deux dernières colonnes du tableau montrent avec précision que ce nombre reste en fait empiriquement dans un ordre $O(N)$, ce qui explique que la traversée prend un temps de calcul comparable à la maintenance de la structure mais pour une plus petite constante car elle entreprend moins d'opérations élémentaires. Remarquons enfin que bien que la maintenance s'effectue théoriquement en $O(N)$ par itération, l'expérience montre plutôt un comportement sur-linéaire. En pratique, il y aurait donc lieu d'accélérer la maintenance de la table comme nous le ferons par la suite. En début d'algorithme, notons que chaque couple de mouvement associé à la solution initiale doit être enregistré dans la table de hachage. Cette phase, qui demande un temps de calcul en $O(N^2)$, n'est pas critique puisque le nombre d'itérations qui utilise la table de hachage est toujours en pratique au moins de l'ordre de N. Les composantes METALAB réalisant l'incrémentation du voisinage 2-*opt* étendu tel qu'étudié au tableau 7.1 sont présentées en annexes B.1, B.2 et B.3.

Compression de la table de hachage. Un des désagréments lié à l'utilisation de la table de hachage concerne l'espace mémoire nécessaire. En pratique pourtant, signalons que pour les instances TSPLIB que nous avons traitées, les temps de calcul se sont avérés bien plus contraignants que l'espace mémoire utilisé. Dans le cas où l'espace mémoire deviendrait une ressource précieuse nous mentionnons deux extensions pertinentes. Premièrement, notons qu'il est possible d'enlever pour chaque arête un nombre donné δ des bits de poids le plus faible dans la représentation de leur coût. Ceci revient à diviser par 2^{δ} chaque coût de mouvement et donc par ce même facteur le nombre de lignes d'enregistrement. Lorsque les distances interponctuelles sont grandes et que leur distribution est assez uniforme, il est raisonnable de penser que ce schéma ne sera pas trop approximatif. Deuxièmement, il est possible de remplacer la table de hachage par une approche d'arbre binaire, ce qui évite de devoir préparer une ligne d'enregistrement pour toute valeur de coût possible. Selon cette approche, les opérations d'insertion et de suppression d'un mouvement deviennent toutefois plus coûteuses car elles fonctionnent en $O(logN)$ pour des arbres équilibrés.

Accélération de la table de hachage. Lorsqu'au contraire la mémoire est une ressource non restrictive et qu'on souhaite accélérer la maintenance de la table de hachage, plutôt que d'utiliser des listes chaînées extensibles, il est possible de borner la taille des listes d'enregistrement. Dans ce cas, chaque ligne d'enregistrement, même vide, prend un espace mémoire constant proportionnel à cette borne. Lorsqu'une ligne d'enregistrement est saturée de mouvements, il est alors raisonnable de se contenter de ne garder en mémoire que les mouvements déjà enregistrés. Cette approche est intéressante car elle permet de remplacer les listes chaînées de chaque ligne d'enregistrement par des tableaux, ce qui permet de faire décroître la constante devant l'ordre de complexité. Notons enfin qu'il est possible de ne considérer que les mouvements dont un des deux arcs entrants (x, y) vérifie que x (ou y) est un des K plus proches voisins de y (ou x). On récupère dans ce cas l'idée de tronquer les

listes de proximité pour une taille K raisonnablement petite. À cet effet, la valeur de K peut être ici beaucoup plus petite que les valeurs comprises entre $0.4N$ et $0.8N$ suggérées pour la phase préliminaire de descente pure, car en mode métaheuristique, la tournée est déjà constituée d'arcs reliant des sommets suffisamment proches. Cette dernière extention a l'avantage de réduire très significativement le temps de mise à jour de la table de hachage qui est en pratique la phase de calcul prépondérante. Dans le cadre de la librairie METALAB, la Caractéristique `NstNgbTwxHshTbl` ainsi que le Serveur `ByGanNstNbrTwxSvr` présentés en annexe B.4 implantent les deux composantes nécessaires à la composition de la méthode taboue utilisant une table de hachage avec listes de proximité tronquées. Des tests de comparaison seront présentés en fin de section suivante, après avoir introduit une dernière accélération possible du voisinage 2-*opt* s'appuyant sur la majoration des diamètres des sous-séquences inversées.

7.3 Segmentation du voisinage 2-*opt*

L'incrémentation du voisinage selon le schéma précédent a permis de ramener le temps de calcul de chaque itération à l'ordre $O(N)$ sans poser de problèmes en pratique pour la taille de la mémoire. Cette première approche ne semble toutefois pas être en mesure de faire concurrence aux techniques d'accélération pour la descente en première amélioration puisque dans ce cadre nous avions fait état d'un nombre $O(N)$ d'itérations et d'un nombre cumulé $O(N)$ d'évaluations de mouvement, ce qui donne $O(1)$ examens de mouvements par itération en moyenne au cours de l'algorithme. En fait, la table de hachage précédente est intéressante en pratique car elle permet de faire des mesures exactes sur le comportement de la méthode taboue pour des tailles de problème et un nombre d'itération beaucoup plus grand que ne le permettrait une recherche explicite du meilleur mouvement à chaque itération.

Tableau 7.2 – Mesures portant sur le diamètre des sous-séquences inversées

Problème		*Diamètre moyen et partition (en pourcentage) des diamètres*											
Nom	*Taille*	d_{moy}			$\%\{i\|d_i > 2d_{moy}\}$			$\%\{i\|d_i > 4d_{moy}\}$			$\%\{i\|d_i > 8d_{moy}\}$		
		CW	*PPV*	*RD*	*CW*	*PPV*	*RD*	*CW*	*PPV*	*RD*	*CW*	*PPV*	*RD*
eil	101	5.22	5.55	6.51	13.9	12.9	12.5	0.95	0.59	1.89	0.01	0.00	0.02
ch	150	2.90	3.73	2.63	3.45	6.49	0.77	0.90	0.65	0.34	0.52	0.04	0.11
gr	202	4.09	3.84	5.53	7.60	7.47	8.15	1.47	2.04	3.66	0.74	0.92	2.32
pcb	442	11.2	11.9	17.8	10.3	10.5	11.0	6.27	6.81	8.79	2.84	3.13	3.55
gr	666	9.50	12.0	8.05	10.6	8.15	11.4	6.53	6.67	4.91	1.79	5.71	1.63
pr	1002	3.22	3.06	3.22	3.47	2.61	3.12	2.91	0.97	1.95	1.24	0.44	0.34
pr	2392	5.35	4.92	6.71	5.93	3.45	2.78	4.33	0.97	1.71	1.02	0.69	1.51

Longueur des sous-séquences inversées. Nous avons en particulier utilisé la table de hachage précédente afin de mesurer sur une période de 100000 itérations et pour un assez large spectre de tailles de problème la longueur des sous-séquences qui sont inversées par les mouvements sélectionnés par la méthode taboue. Rappelons que l'application d'un 2-*opt* peut être implantée par l'inversion d'une des deux sous-séquences induites par les deux arcs sortants et que pour les tournées proposées au chapitre 5, cette opération fonctionne en inversant toujours la plus petite de ces deux sous-séquences. La moyenne ainsi que quelques propriétés concernant la distribution de ces longueurs sont mentionnées dans le tableau 7.2. Il est ici très intéressant de remarquer qu'en moyenne la longueur observée est beaucoup plus petite que la taille N du problème de même que la moyenne $O(N^{0.75})$ observée en phase de descente pure. Pour les problèmes considérés, ce diamètre moyen ne semble même pas augmenter avec le nombre N de villes. La deuxième amélioration de la méthode taboue que nous proposons exploite cette propriété.

Segmentation des diamètres. Que la méthode taboue opère des mouvements de petit diamètre n'est pas étonnant. Lorsqu'en mode métaheuristique la trajectoire de recherche avoisine continuement de bonnes solutions, il est en effet raisonnable de

penser que pour une taille de liste taboue bien choisie, la recherche a naturellement tendance à osciller, à l'échelle du mouvement 2-*opt*, entre une phase de diversification et d'intensification. La diversification a lieu lorsque deux sommets voisins en distance mais éloignés sur la tournée sont reliés par le mouvement sélectionné. Cette phase entraîne alors certaines perturbations aux bornes de la sous-séquence inversée que les mouvements suivants auront successivement tendance à réparer. L'oscillation entre ces deux tendances ne suit malheureusement pas un critère rationnel tel qu'implanté habituellement dans des stratégies alternant intensification et diversification. En effet, le choix du mouvement suit le seul critère de gain, ce qui empêche par exemple parfois de visiter plus intensivement qu'il ne faudrait le voisinage d'une solution où deux sommets auparavant distants sur la tournée ont été reliés. La segmentation du voisinage que nous proposons exploite le petit diamètre d'inversion qu'opèrent les mouvements en moyenne, pour d'une part accélérer davantage la sélection d'un mouvement et d'autre part pour contrôler le temps passé à post-optimiser la solution obtenue après diversification. Le principe est de n'autoriser pendant un certain nombre d'itérations que des mouvements à petit diamètre.

Recherche d'un mouvement de petit diamètre. En plus d'autoriser le contrôle de la durée de réparation d'une solution élite, la segmentation du voisinage permet d'accélérer très significativement la sélection d'un mouvement. En effet, puisqu'on borne les diamètres des mouvements sélectionnables à un nombre L négligeable par rapport à la taille du problème, le nombre de mouvements à énumérer est de l'ordre de $O(LN)$. Pour des paramètres `dmin` et `dmax` judicieux, la structure `ByMidBndDiaTwxSvr` présentée à la figure 5.16 au chapitre 5 réalise par exemple une telle énumération. En reprenant l'idée d'une table de hachage, il est toutefois possible de faire beaucoup mieux. Dans ce cas, il est inutile de répertorier chaque mouvement en double comme auparavant puisque seuls $O(L^2)$ mouvements sont touchés (soit qu'ils changent de statut valide/invalide soit qu'ils apparaissent/disparaissent) par

le mouvement qu'on applique entre deux itérations. Avec une notation judicieuse de chaque mouvement, il est alors possible de détecter chaque mouvement touché ainsi que de corriger ses coordonnées et son positionnement dans la table en temps constant. La maintenance de la structure s'effectue alors en $O(L^2)$. De plus, le problème de dominance des mouvements valides par les mouvements invalides disparaît de sorte que la recherche du meilleur mouvement s'effectue en temps constant. Notons enfin que l'initialisation de la structure opère en $O(LN + D)$, ce qui est amorti en pratique si on effectue $O(N/L)$ itérations d'intensification (le terme en D peut-être amorti sur l'ensemble des itérations de l'algorithme puisqu'il provient seulement de l'allocation initiale du tableau qui réfère l'ensemble des lignes d'enregistrement). La Caractéristique `BndDiaTwxHshTbl` fournie en annexe B.5 implante la table de hachage telle que décrite dans cette section pour des mouvements à petit diamètre. Le Serveur `ByGanBndDiaTwxSvr` présenté en annexe B.6 utilise ensuite cette table afin de circuler comme convenu par ordre de gain décroissant sur le voisinage restreint aux petits diamètres.

Le tableau 7.3 compare les temps de calcul ainsi que la valeur de la meilleure solution trouvée par des algorithmes de recherche locale s'appuyant sur les tables de hachages `AllTwxHshTbl` (incrémentation exacte du voisinage étendu), `BndDiaTwxHshTbl` (incrémentation du voisinage classique, mais segmenté à des petits diamètres d'inversion) et `NstNgbTwxHshTbl` (incrémentation accélérée du voisinage étendu par listes de proximité tronquées). Dans chaque cas, la première ligne indique la meilleure valeur trouvée en 100000 itérations, et la deuxième, le temps de calcul écoulé en secondes. Les première et deuxième colonnes indiquent le nom et la taille ainsi que pour référence, la valeur de la tournée optimale de chaque problème. La troisième colonne indique les paramètres des méthodes de résolution. Le paramètre T, dont nous avons cherché les valeurs optimales au cours d'expérimentations préliminaires, correspond à la longueur de la liste taboue (basée sur les Attributs `ToAdj` et `FrAdj`

Tableau 7.3 – Performances des différentes tables de hachage

Contexte			`AllTwxHshTbl`			`BndDiaTwxHshTbl`			`NstNgbTwxHshTbl`		
Pbm.	*Opt.*	*T/L/K*	*CW*	*PPV*	*RD*	*CW*	*PPV*	*RD*	*CW*	*PPV*	*RD*
eil 101	629	20/10/10	**635** 33	639 29	**634** 28	650 27	649 26	642 28	**635** 10	**631** 9	638 9
ch 150	6528	30/10/12	**6575** 45	**6599** 45	6738 44	6888 24	6620 23	6781 23	6756 10	**6599** 10	**6723** 10
gr 202	40160	40/10/15	40524 69	41512 69	41952 71	41683 26	41845 25	42517 27	**40510** 15	**41487** 14	**41920** 15
pcb 442	50778	80/10/20	51447 216	51591 224	**51750** 223	53297 31	53388 29	54511 29	**51384** 21	**51193** 20	51833 21
gr 666	294358	100/10/25	**314378** 407	**312988** 404	**320034** 406	318682 31	315649 30	322773 31	314628 31	313156 31	320035 32
pr 1002	259045	150/10/30	274256 629	**267131** 642	**272020** 659	278202 35	270065 36	282760 33	**273731** 40	267282 40	273107 41
pr 2392	378032	400/10/50	**399254** 1441	**394931** 1381	**401347** 1369	412897 53	401609 55	418827 58	399872 86	395492 86	402457 91

du chapitre précédent). Le paramètre L, qui correspond au diamètre maximal considéré dans le cas de la segmentation du voisinage, est toujours égal à 10, ce qui pour la majorité des problèmes étudiés dans ce chapitre, correspond au double du diamètre moyen inversé. Le paramètre K, enfin, correspond au rang du dernier voisin considéré par la troncature des listes de proximité dans le cas de l'incrémentation accélérée du voisinage étendu. Nous avons choisi de tester des valeurs de $K \approx \sqrt{N}$. Chacune des trois approches est testée depuis un même triplet de solutions initiales obtenu après application d'une descente en meilleure amélioration depuis le résultat de l'algorithme de Clarke et Wright (CW), depuis le résultat de l'algorithme des plus proches voisins (PPV), et enfin, depuis une solution générée au hasard. Avant de poursuivre l'analyse de ces résultats expérimentaux, rappelons que la colonne de référence `AllTwxHshTbl` constitue déjà une amélioration très significative en terme de temps de calcul relativement à l'énumération séquentielle de tous les mouvements à chaque itération.

Bien qu'elle ait permis d'améliorer la solution initiale, remarquons d'abord que la segmentation du voisinage est très nettement moins efficace que les deux autres approches à tendre vers l'optimum. Des trois approches proposées, la segmentation du voisinage est en effet, par construction, celle qui opère seulement des mouvements ne modifiant que très localement la structure de la solution courante. Si cette propriété peut constituer un atout considérable lorsqu'on souhaite contrôler l'intensification de la recherche autour d'une solution élite, elle s'avère ici trop statique pour améliorer globalement des solutions encore trop éloignées de l'optimum. Son usage, à réserver en phase de post-optimisation, bénéfie par ailleurs d'un temps de calcul très proche de $O(1)$ si on considère, à l'analyse du tableau 7.2, que le diamètre moyen des sous-séquences inversées par une méthode taboue ne s'accroit que très légèrement avec la taille du problème.

Tel que nous l'avions anticipé, remarquons que la troncature des listes de proximité peut être effectuée pour des valeurs de K beaucoup plus petites que dans le cas de la phase de descente pure initiale (où K prenait des valeurs entre $0.4N$ et $0.8N$, cf. section 7.1). En effet, pour des valeurs allant de $K = 10$ (pour un PVC de 101 villes) à $K = 50$ (pour un PVC de 2392 villes), les meilleures solutions trouvées sont réparties de manière assez équilibrées entre la méthode taboue s'appuyant sur le voisinage étendu complet (`AllTwxHshTbl`) et la méthode taboue accélérée s'appuyant sur les listes de proximité (`NstNgbTwxHshTbl`). Empiriquement, la troncature du voisinage étendu a donc l'avantage de réduire le temps de calcul de chaque itération en deça d'une fonction linéaire, tout en proposant des valeurs de solutions comparables, sinon meilleures, que l'énumération complète du voisinage.

7.4 Techniques d'anticipation

Les différentes techniques d'accélération de la recherche que nous avons proposées autorisent une troisième amélioration de la méthode taboue. Nous reprenons pour cela

l'idée des techniques d'anticipation qu'on retrouve notamment en programmation par contraintes. Dans ce paradigme, chaque point de branchement de l'algorithme de recherche effectue une dichotomie sur l'ensemble des valeurs encore affectables à une variable donnée. Selon ce seul schéma, l'algorithme opère la recherche de solution avec chainage arrière classique de l'intelligence artificielle. La programmation par contraintes se distingue par l'intégration à chaque point de branchement d'un certain nombre de mécanismes qui visent à réduire le domaine des valeurs affectables aux autres variables en propageant circulairement l'effet de bord qu'induit la restriction d'une variable vis-à-vis des autres variables partageant une même contrainte. Cette approche constitue une technique d'anticipation puisqu'à chaque point de branchement, l'algorithme anticipe l'effet du choix effectué. De cette façon, le sous-arbre de résolution sous-jacent à chaque branchement peut-être grandement réduit, ce qui a pour effet d'accélérer l'algorithme si le calcul d'anticipation ne coûte pas trop cher.

Technique d'anticipation pour la méthode taboue. L'espace de recherche et le déploiement de la trajectoire des solutions étant tout à fait différents dans le contexte de la recherche locale, les techniques d'anticipation prennent ici une forme autrement particulière. Nous proposons à cet effet de ne pas choisir à chaque itération le meilleur mouvement possible non tabou mais plutôt de développer temporairement une recherche locale pour un horizon de H mouvements applicables depuis la solution courante. En première itération temporaire, nous gardons les w_1 meilleurs mouvements non tabous, lesquels permettent d'accéder aux solutions voisines qu'on note $S_i^1, ...S_i^{w_1}$ par ordre croissant relativement à leur fonction coût. À partir de chacune de ces solutions une deuxième itération temporaire est effectuée. Pour chaque solution $S_i^{1..1}, ..., S_i^{w_1..w_k}$ (avec $S_i^{1..1} > ... > S_i^{1..w_k}, ..., S_i^{w_1..1} > S_i^{w_1..w_k}$) obtenue après la $k^{\text{ième}}$ itération temporaire, on répète le même schéma en retenant pour chaque solution atteinte ses w_{k+1} meilleurs mouvements non tabous. Après la dernière itération temporaire, le premier mouvement qui se situe sur le chemin qui a conduit à la

meilleure solution identifiée en phase d'identification est définitivement appliqué. La procédure d'anticipation est alors relancée depuis la nouvelle solution S_{i+1} obtenue.

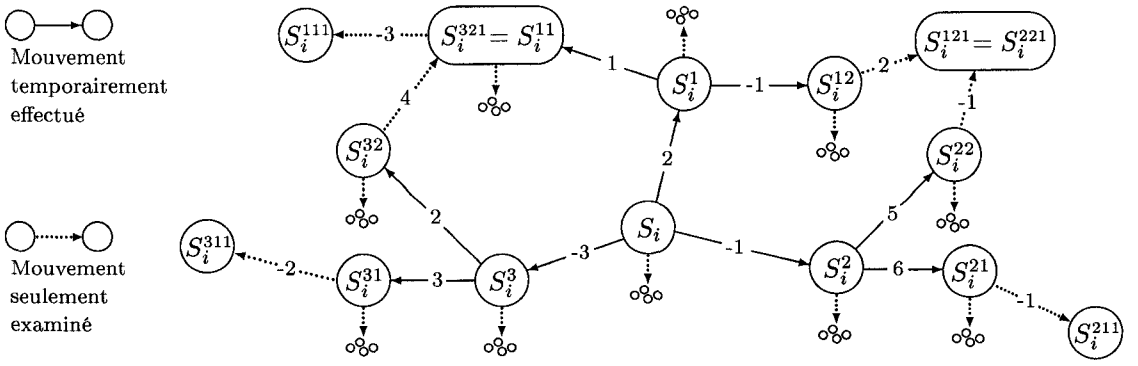


Figure 7.2 – Hyper-voisinage visité par une phase d'anticipation $(3, 2)$

Lors de la phase d'anticipation, un mouvement est tabou s'il conduit a une solution récemment visitée sur la trajectoire de recherche temporaire, c'est-à-dire sur la trajectoire de recherche principale $S_0...S_i$ suivi du chemin partant de S_i et menant à la solution temporaire courante. Selon ce critère, il est donc possible de visiter deux fois la même solution temporaire dans la même phase d'anticipation ou dans deux phases d'anticipation consécutives mais il est impossible qu'une solution temporaire soit visitée deux fois sur une même branche de la phase d'anticipation. La figure 7.2 illustre le schéma de résolution ainsi mis au point. Sur chaque arc, on indique le gain du mouvement correspondant. Pour la phase d'anticipation présentée, le mouvement conduisant à S_i^2 est finalement définitivement appliqué car S_i^{21} est la meilleure solution de l'hyper-voisinage. Ce mouvement alimente alors la liste taboue pour un certain nombre d'itérations. Notons que lors d'une dernière itération temporaire, seul le meilleur mouvement peut conduire à la meilleure solution visitée en phase d'anticipation. On a donc toujours $w_H = 1$. Pour la suite, la notation $(w_1, ..., w_{H-1})$, où

chaque w_k représente la largeur de la $k^{\text{ième}}$ itération temporaire, est utilisée pour désigner les paramètres du schéma d'anticipation. En particulier, la suite vide () désigne ainsi la méthode taboue sans anticipation. Dans le cas général, une phase d'anticipation visite $w_1 + w_1 w_2 + ... + (w_1 ... w_{H-1})$ solutions suite à l'application temporaire des mouvements.

Un exemple de scénario. Le cas de figure présenté à la figure 7.3 montre une situation où une méthode taboue sans technique d'anticipation ignore la solution optimale pour la portion de tournée considérée. Cette solution est par contre identifiée par un mécanisme avec anticipation $(3, 2)$. Notons que sans anticipation, il serait possible que la méthode taboue sélectionne depuis la solution S_i^1 un mouvement 2-*opt* diversifiant mais dégradant ce qui pourrait avoir comme conséquence que la réparation intéressante qu'aurait proposée l'anticipation devienne superflue même si elle est effectuée ultérieurement.

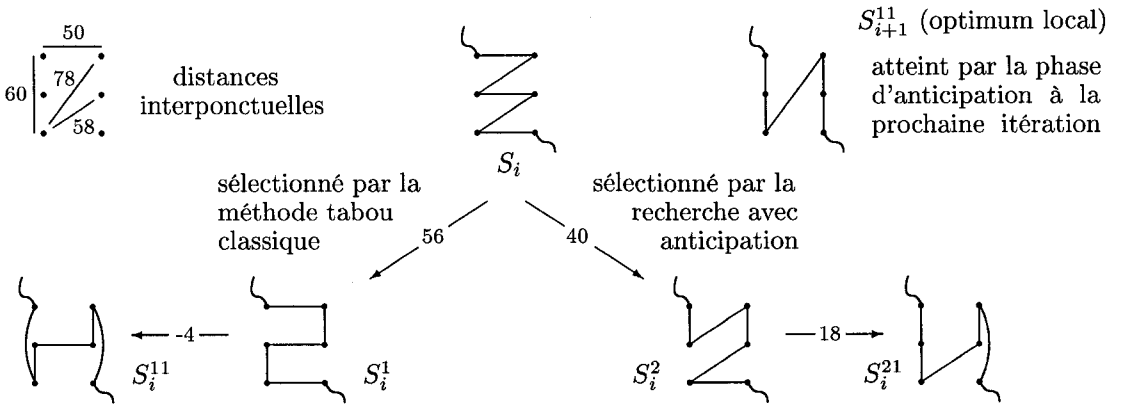


Figure 7.3 – Comparaison de la méthode taboue avec/sans anticipation

Afin de tester l'incidence des techniques d'anticipation en phase d'intensification, nous avons généré pour chaque problème étudié dans ce chapitre un quadruplet de solutions élites. Nous avons dans un premier temps généré un ensemble de 20

Tableau 7.4 – Performances de la recherche avec anticipation

Pbm.	*Opt.*	*T/K*	*CPU*	Valeur objectif et pourcentage à l'optimum (avant et après intensification)							
eil 101	629	20/10	1	630 629	0.16 0.00	630 629	0.16 0.00	630 629	0.16 0.00	631 629	0.16 0.00
ch 150	6528	30/12	2	6545 6528	0.26 0.00	6558 6549	0.45 0.32	6560 6549	0.49 0.32	6562 6549	0.52 0.32
gr 202	40160	40/15	3	40447 40363	0.71 0.51	40464 40391	0.76 0.58	40472 40351	0.78 0.48	40480 40343	0.80 0.46
pcb 442	50778	80/10	7	51101 50888	0.64 0.22	51178 50837	0.79 0.12	51193 51011	0.82 0.64	51277 51094	0.98 0.62
gr 666	294358	100/25	13	306075 305488	3.98 3.78	306664 306138	4.18 4.00	306834 306376	4.24 4.08	307204 306026	4.36 3.96
pr 1002	259045	150/30	14	266228 264672	2.77 2.17	266960 265489	3.06 2.48	267275 265938	3.17 2.66	267545 266012	3.28 2.69
pr 2392	378032	400/50	30	388903 385457	2.88 1.96	391518 388168	3.57 2.68	392397 388271	3.80 2.71	392504 388887	3.82 2.87

solutions initiales (10 par l'algorithme de Clarke et Wright en faisant varier le dépot, et 10 par l'algorithme des plus proches voisins en considérant les sommets dans un ordre différent) pour chacune desquelles 500000 itérations de la méthode taboue ont ensuite été exécutées. Chaque quadruplet a finalement été obtenu en ne retenant que les quatre meilleures solutions, obtenues depuis des solutions initiales distinctes. Ce cadre d'étude est préliminaire puisqu'en pratique, on aurait tout intérêt à chercher de telles solutions d'élite, par exemple, par une variante de l'algorithme de Lin et Kernighan, plus rapide. La raison pour laquelle nous avons choisi un tel protocole est que nous disposions déjà des algorithmes nécessaires. Pour chaque problème, le coût de la solution d'élite à améliorer ainsi que la valeur de la meilleure solution obtenue après 100 itérations d'une anticipation de paramètre (100) sont représentées dans le tableau 7.4. À titre d'indication, nous fournissons dans chaque cas le pourcentage à l'optimum. Pour les problèmes de petites tailles, notons que les solutions d'élite sont assez proches de l'optimum de sorte que la phase d'intensification permet de

diminuer de manière conséquente l'écart relatif à l'optimum. Pour les problèmes de plus grande taille, l'amélioration obtenue sur le pourcentage à l'optimum augmente en valeur absolue mais diminue en valeur relative. Dans chaque cas, l'amélioration est obtenue en quelques secondes.

7.5 Conclusion

Nous avons présenté dans ce chapitre un certain nombre d'accélérations significatives pour l'exploration du voisinage 2-*opt*. Aucune des techniques présentées n'ont à notre connaissance été à ce jour proposées dans la littérature. Nous avons enfin exploité l'incrémentation limitée aux K plus proches voisins du 2-*opt* étendu afin de construire un outil de post-optimisation basé sur des techniques d'anticipation. L'expérience a montré que pour chaque problème, le pourcentage à l'optimum a pu être amélioré de manière significative sur un petit nombre d'itérations et pour des temps de calcul relativement courts. L'objectif de ce chapitre était par ailleurs de montrer, en correspondance avec l'annexe B, comment il est possible de poursuivre un travail de développement efficace au moyen de la librairie METALAB.

Du point de vue de la recherche locale, l'étude du 2-*opt* pour le PVC faite dans ce chapitre constitue un travail préliminaire. Les différents axes de recherche en voie d'être étudiés concernent, d'une part, l'approfondissement de l'étude des techniques d'anticipation, et d'autre part, la mise en place d'un algorithme tabou permettant de générer des solutions élites efficacement. Pour ce dernier point, le remplacement des K plus proches voisins au sens de la distance par les K plus proche voisins au sens du coût d'opportunité proposé par Helsgaun (2000) est en cours d'examen. Étant donné un 1-arbre de poids minimum, le coût d'opportunité d'une arête est définie par le surplus de poids induit sur le 1-arbre si on l'oblige à contenir l'arête considérée.

Sous ce critère de proximité, il a été observé sur la plupart des problèmes TSPLIB que les solutions optimales sont constituées uniquement d'arêtes dont un des deux sommets appartient aux cinq plus proches sommets de l'autre l'extrémité. Pour le problème `pr2392` par exemple, ceci permettrait de limiter K à la valeur 5. D'une part, l'incrémentation du voisinage serait beaucoup plus rapide, et d'autre part, la recherche focaliserait sur les arêtes intéressantes, ce qui devrait diminuer grandement le nombre d'itérations nécessaires pour trouver de bonnes solutions.

Conclusion

Nous avons présenté dans cette thèse un modèle de programmation générique pour la recherche locale. D'une part, l'objectif théorique a consisté à analyser les constructions informatiques factorisables rencontrées dans le domaine de la recherche locale. D'autre part, l'objectif pratique a conduit à développer et à expérimenter la librairie METALAB. Nous rappelons dans un premier temps les contributions majeures qui ont été réalisées en vue de ces objectifs puis donnons quelques axes de recherche pertinents dans le prolongement du travail effectué.

Une première contribution de la thèse a été de prolonger l'approche des cadres d'application proposés dans la littérature. Tout comme ces utilitaires, une librairie générique permet en effet d'utiliser des schémas algorithmiques déjà programmés et auxquels on passe en paramètre les composantes propres à un contexte particulier. Cette propriété assure la réusabilité des composantes algorithmiques et la généricité du logiciel relativement aux différents types de problèmes. Mais contrairement aux cadres d'application, une librairie générique décompose les pièces logicielles en autant de concepts élémentaires que possibles au nombre desquels les Attributs et Caractéristiques servent de relais entre les autres pièces du logiciel. Comme nous l'avons montré au chapitre 6 par un certain nombre d'exemples, il devient ainsi facile d'adapter certaines parties du schéma de recherche locale selon la structure du problème à résoudre. Comme illustré en fin de chapitres 5 et 7, il devient également possible de définir de nouveaux schémas de sélection de mouvements sans rentrer dans les technicités de la librairie. Comparativement aux cadres d'application, une librairie générique supporte donc la programmation de la recherche locale tant au niveau des composantes dépendant du problème que des composantes algorithmiques.

Une deuxième contribution de la thèse a été de proposer des mécanismes calculant l'ordre de maintenance des différentes pièces logicielles participant à une application donnée. Nous nous sommes appuyés pour cela sur des parcours dans des diagrammes de composition statiques représentant l'interaction entre les différentes pièces logicielles. Le seul calcul d'ordre de maintenance dynamique a lieu lorsqu'un Attribut est composé dynamiquement au moyen du mot-clé `If` tel qu'illustré au chapitre 4. Cette construction est utile lorsqu'on souhaite éviter de calculer systématiquement un Attribut qu'il soit effectivement utilisé ou non. À noter que tous les ordres de maintenance sont établis dès la compilation du programme en fonction des composantes fournies dans les en-têtes `BuildWith`, `MoveWith`, `ScanWith`, etc. Cette technique, qui repose sur l'hypothèse que peu d'Attributs sont en pratique intégrés dans une application, a l'avantage de se déployer sans surchage de calcul significative en cours d'exécution. Pour des tests réalisés sur un algorithme en première amélioration résolvant le PVC pour des tailles allant de quelques centaines à plusieurs milliers de noeuds, le code généré par la librarie est resté en moyenne 30% plus lent que le même programme codé explicitement. Il est peut-être important de souligner le lien qui existe ici entre librairies génériques et langages déclaratifs, dont le noyau repose également sur des calculs d'ordre de maintenance. À cet égard, les langages déclaratifs font davantage d'hypothèses sur les modèles de recherche locale, lesquels doivent être essentiellement analytiques, tels que rencontrés en satisfaction de contraintes. Si elle est plus restrictive, cette approche permet en revanche de programmer les fonctions de maintenance (l'équivalent des fonctions `scan` et `track`) dans le noyau du langage de sorte que l'usager s'en tient à déclarer les grandeurs qui doivent être maintenues pour développer sa recherche. Dans l'autre sens, les langages déclaratifs font moins d'hypothèses sur les diagrammes de composition que ne le font les librairies génériques. Dans notre approche, ces diagrammes doivent être acycliques en temps de compilation alors que certains langages déclaratifs résolvent les problèmes de parcours dans les diagrammes en fonction des besoins effectifs de maintenance en temps d'exécution. L'utilisation d'un langage déclaratif peut alors induire une surcharge de calcul non négligeable.

Une troisième contribution de la thèse concerne la réalisation ainsi que l'expérimentation de la librairie METALAB conformément au modèle de programmation établi en partie théorique. Pour cela, nous avons donné l'ensemble des spécifications C++ de chaque concept de la recherche locale et avons documenté leur programmation au moyen d'exemples didactiques. L'utilisation de METALAB a été illustrée sur le problème du voyageur de commerce, le problème 3-SAT, le problème du voyageur de commerce avec coûts d'arêtes dépendant du temps, le problème de la p-médiane, le problème de coloration de graphe, et enfin, le problème d'affectation quadratique. Nous avons également montré comment réaliser des composantes génériques de listes taboues, de longueur fixe ou variable, ainsi que le critère d'acceptation utilisé dans le recuit-simulé. Nous avons aussi montré comment accélérer la recherche au moyen d'Attributs ou de Caractéristques auxiliaires. En particulier, nous avons montré comment composer dans un Serveur des Caractéristiques maintenant l'ensemble des mouvements selon l'ordre des gains induits par ces mouvements. Enfin, nous avons montré comment finaliser la recherche en déclarant et en initialisant un Explorateur, puis en itérant sur les fonctionnalités de recherche que ce dernier propose.

Une quatrième contribution de la thèse concerne enfin l'étude de la méthode taboue pour la résolution du voyageur de commerce. Cette étude visait à corriger le point de vue considérant que la méthode taboue est inefficace pour ce problème en raison du calcul de complexité quadratique nécessaire pour évaluer l'ensemble des mouvements 2-*opt* à chaque itération. Dans un premier temps, nous avons défini un voisinage étendu dont le nombre de mouvements affectés à chaque itération est de l'ordre du nombre N de sommets. Cette première phase, qui a permis d'étudier le comportement de la méthode taboue sur un grand nombre d'itérations et sur un assez large spectre de tailles de problèmes, a montré qu'une fois la phase de descente pure achevée, la méthode taboue ne choisit quasiment que des mouvements inversant des sous-séquences de diamètres négligeables par rapport à N. Nous avons donc défini un

algorithme de recherche se limitant à la version limitée des 2-*opt* de faibles diamètres. Afin d'améliorer la recherche en phase d'intensification, nous avons enfin proposé un schéma de sélection avec anticipation, où à chaque itération, le mouvement choisi est celui situé en premier sur le chemin conduisant à la meilleure solution temporairement visitée au moyen de moins de d mouvements depuis la solution courante. L'étude de la méthode taboue pour le voyageur de commerce a par ailleurs été l'occasion de montrer l'utilité de la librairie METALAB pour la conduite de projets avancés.

Les principaux axes de recherche qu'on pourrait suivre dans le prolongement de cette thèse concernent principalement l'extension de la librairie METALAB. Une première extension intéressante consisterait à définir le concept d'Environnement comme un groupe de Caractéristiques définies à l'extérieur d'un Explorateur. À ce propos, notons que dans la version actuelle de METALAB, l'ensemble des composantes utilisées par un algorithme est défini localement dans l'Explorateur. Avec la notion d'Environnement, il serait possible d'intégrer certaines composantes depuis l'extérieur. Cette possibilité ouvre la voie à des mécanismes de recherche locale parallèle avec gestion de mémoire partagée. Par exemple, une liste taboue intégrée dans un Environnement serait partagée par l'ensemble des Explorateurs qui la composeraient. Une deuxième extension possible de la librairie consiterait à définir des Attributs ayant une fonction `scan` pour plusieurs types de Mouvements. En combinant cette possibilité avec la notion d'Environnement, un ensemble de recherches fondé sur des schémas de mouvement distincts pourrait alors partager ses Caractéristiques, tel qu'une liste taboue. L'hybridation de la programmation à mémoire adaptive et de la recherche à voisinage variable serait ainsi encouragée. La dernière extension de la librarie concerne enfin le but ultimement poursuivi par la réalisation de METALAB, consistant à implanter selon un même protocole un catalogue d'algorithmes de recherche locale efficaces pour des problèmes incontournables, tant du point de vue du chercheur que du practicien. À terme, nous espérons que ce projet de standardisation, amorcé dans cette thèse, contribue à faire progresser le paradigme de la recherche locale.

Bibliographie

AHUJA, R.K., ERGUN, O., ORLIN, J.B., PUNNEN, A.P. (2002). A Survey of Very Large-Scale Neighborhood Search Techniques. *Discrete Applied Mathematics*, 123(1).

ANDREATTA, A.A., CARVALHO, S.E.R., RIBEIRO, C.C. (1998). An Object-Oriented Framework for Local Search Heuristics. Technical Report, Catholic University of Rio de Janeiro.

APPLEGATE, D., COOK, W., ROHE, A. (2003). Chained Lin-Kernighan for large traveling salesman problems. http ://citeseer.nj.nec.com/applegate99chained.html. (page consultée le 30 septembre 2003).

ATHANASIOS M., PARDALOS P., VÄRBRAND P. (1998). *Multilevel Optimization : Algorithms and Applications.* Kluwer Academic Publishers.

AUSTERN, M.H. (1999). GENERIC PROGRAMMING AND THE STL : USING AND EXTENDING THE C++ STANDARD TEMPLATE LIBRARY. Addison-Wesley.

BARTHOLDI, J.J., PLATZMAN, L.K. (1988). Heuristics Based On Spacefilling Curves For Combinatorial Problems In Euclidian Space. *Management Science*, 34(3) : 291–305.

BATTITI, R., TECCHIOLLI, G. (1994). The Reactive Tabu Search. *ORSA Journal on Computing*, 6(2) : 126–140.

BENTLEY, J.L. (1992). Fast Algorithms for Geometric Travelling Salesman Problems. *ORSA Journal on Computing*, 4(4) : 387–411.

BOHLIN, M. (2002). Constraint Satisfaction by Local Search. Technical Report. T2002-07, SICS. http ://citeseer.nj.nec.com/bohlin02constraint.html (Page consultée le 30 septembre 2003).

BROTCORNE, L., FARAND, L., LAPORTE, G., SEMET, F. (1999). Impacts des nouvelles technologies sur la gestion des systèmes de véhicules d'urgence. Rapport Technique. CRT–99–39.

BURKARD, R.E., KARISCH, S., RENDL, F. (1997). QAPLIB : A Quadratic Assignment Problem Library. *Journal of Global Optimization*, 10 : 391-403.

CALEGARI, P., CORAY, G., HERTZ, A., KOBLER, D., KUONEN, P. (1999). A Taxonomy of Evolutionary Algorithms in Combinatorial Optimization. *Journal of Heuristics*, 5 : 145–158.

CRAINIC, T., TOULOUSE, M. (1997). Parallel Metaheuristics. Rapports du CRT, CRT–97–50, Montréal.

CRAINIC, T.G., TOULOUSE, M., GENDREAU, M. (1997). Toward a Taxonomy of Parallel Tabu Search Heuristics. *INFORMS Journal on Computing*, 9(1) : 61–72.

CROUZET, S., SAVARD, G. (2002). Real-time Optimization and Decision Support Systems. *In Proceedings of the IASTED International Conference, Applied Informatics*, Innsbruck, 2002.

DINCBAS, M., VAN HENTENRYCK, P., SIMONIS, H., AGGOUN, A., GRAF, T. (1988). The constraint logic programming language CHIP. *In Fifth Generation Computer Systems*, Tokyo, 1988. Springer.

DI GASPERO, L., SCHAERF, A. (2001a). EasyLocal++ : An Object-Oriented Framework for Flexible Design of Local Search Algorithms. *In proceedings of the 4th Metaheuristics International Conference (MIC2001)*, Porto, 2001.

DI GASPERO, L., SCHAERF, A. (2001b). A case-study for EasyLocal++ : the course Timetabling Problem. Rapport Technique UDMI/13/2001/RR.

FERLAND, J.A., HERTZ, A., LAVOIE, A. (1996). An Object-Oriented Methodology for Solving Assignment Type Problems with Neighborhood Search Techniques. *Operations Research*, 44(2) : 347–359.

FINK, A., VOSS, S., WOODUFF D.L. (1998a). Building Reusable Software Components for Heuristic Search. Extented Abstract of a talk given at OR98, Zurich. http ://citeseer.nj.nec.com/fink98building.html (page consultée le 30 septembre 2003).

FINK, A., VOSS, S., WOODUFF D.L. (1998b). Building Reusable Software Components for Heuristic Search. Tutorial INFORMS/CORS, Montréal, 1998.

FISCHETTI M., LAPORTE G., MARTELLO S. (1993). The Delivery Man Problem and Cumulative Matroïds. *O*perations Research, 41 : 1055–1076.

FOURER, R., GAY, D., KERNIGHAN, B. (1993). *AMPL : A Modeling Langage for Mathematical Programming.* The Scientific Press, San Francisco.

FREDMAN, M.L., JOHNSON, D.S., MCGEOCH, L.A., OSTHEIMER, G. (1995). Data Structures for Travelling Salesmen. *Journal of Algorithms*, 18 : 432–479.

GALINIER, P., HAO, J.K. (2000). A General Approach for Constraint Solving by Local Search. *In Proceedings of the Second International Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems (CP-AI-OR-00)*, Paderborn, 2000.

GALINIER, P., HERTZ, A. (2006). A Survey of Local Search Methods for Graph Coloring. À paraître dans *Computers & Operations Research.*

GAMMA, E., HELM, R., JOHNSON, R., VLISSIDES, J. (1994). *Design Patterns, Elements of Reusable Object-Oriented Software.* Addison-Wesley.

GARVEY, A., LESSER, V. (1994). A Survey of Research in Deliberative Real-Time Artificial Intelligence. *Journal of Real-Time Systems*, 6(3).

GENDREAU, M., GUERTIN, F., POTVIN J.-Y., TAILLARD, É. (1999). Parallel Tabu Search for Real-Time Vehicle Routing and Dispatching. *Transportation Science*, 33(4) : 381–390.

GENDREAU, M., SORIANO, P., SALVAIL, L. (1993). Solving the maximum clique problem using a tabu search approach. *Annals of Operations Research*, 41 : 385–403.

GENDREAU, M., MARCOTTE, P., SAVARD G. (1996). A Hybrid Tabu-Ascent Algorithm for the Linear Bilevel Programming Problem. *Journal of Global Optimization*, 9 : 1–14.

GÉRON, A., TAWBI, F. (1999). *Pour mieux développer avec C++ : design patterns, STL, RTTI et smart pointers.* Dunod, Paris.

GLOVER, F. (1986). Future Paths for Integer Programming and Links to Artificial Intelligence, *Computers and Operations Research*, 13 : 533–549.

GLOVER, F. (1996). Ejection Chains, Reference Structures and Alternating Path Methods for Traveling Salesman Problems, *Discrete Applied Mathematics*, **65**, 223–253.

GLOVER, F., LAGUNA, M. (1997). *Tabu Search.* Kluwer Academic Publishers.

GOLBERG, D. (1989). *Genetic Algorithms in Search Optimization and Machine Learning.* Addison-Wesley Publishing Company.

GU, J. (1992). Efficient local search of very-large satisfiability problems. *SIGART Bulletin*, 3(1) : 8–12.

HANSEN, P. (1986). The Steepest Ascent Mildest Descent Heuristic for Combinatorial programming, *Congress on Numerical Methods in Combinatorial Optimization*, Capri.

HANSEN, P., MLADENOVIC, N. (1997). Variable neighborhood search for the p-median. *Location Science*, 5 : 207–226.

HANSEN P., MLADENOVIC N. (2002). Recherche à voisinage variable. *Optimisation approchée en recherche opérationnelle*, J. Teghem and M. Pirlot, Lavoiser, Hermés Science Publications, 81–100.

HANSEN, P., MLADENOVIC, N. (2003). A Tutorial on Variable Neighborhood Search. Les cahiers du GERAD. G–2003–46.

HAYES, B. (1997). Can't Get No Satisfaction. *(*American Scientist), 85(2) : 108–112. *European Journal of Operations Research*, 12 : 106–130.

HELSGAUN, K. (2000). An effective implementation of the Lin-Kernighan traveling salesman heuristic. *European Journal of Operations Research*, 12 : 106–130.

HERTZ, A. (2005). Les méta-heuristiques : quelques conseils pour en faire bon usage. *Gestion de Production et Ressources Humaines : méthodes de planification dans les systèmes productifs*, Presses Internationales de Polytechnique, 205-222.

HERTZ, A., DE WERRA, D. (1987). Using Tabu Search Techniques for Graph Coloring. *Computing*, 39 : 345–351.

HERTZ, A., TAILLARD, E., DE WERRA, D. (1995). A Tutorial on Tabu Search. *Proc. of Giornate di Lavoro AIRO'95* (Entreprise Systems : Management of Technological and Organizational Changes).

HERTZ, A., WIDMER, M. (2002). Guidelines for the Use of Meta-Heuristics in Combinatorial Optimization. *European Journal of Operational Research*, 151 : 247-252.

HOLLAND, J. (1975). *Adaptation in Natural and Artificial Systems.* University of Michigan Press.

HOOS, H., STÜTZLE, T. (1999). Systematic versus Local Search. *In Burgard, W., Christaller T., Cremers, A. editors, KI-99 : Advances in Artificial Intelligence, volume 1701 in Lecture Notes in Artificial Intelligence*, 289–293, Springer Verlag, Berlin, 1999.

HOOS, H., STUTZLE, T. (2000). Local Search Algorithms for SAT : An Empirical Evaluation. *Journal of Automated Reasoning*, 24(4) : 421–481.

JOHNSON, D.S., MCGEOCH, L.A. (1997). The Travelling Salesman Problem : A Case Study in Local Optimization. *Local Search in Combinatorial Optimization.* Aarts, E.H.L, Lenstra, J.K. London : John Wiley & Sons, 215–310.

JUNGER, M., THIENEL, S. (1998). Introduction to ABACUS - A Branch-and-Cut System. *Operations Research Letters*, 22 : 83–95.

KERNIGHAN, B.W., LIN, S. (1970). An efficient heuristic for partitioning graphs. *Bell System Technology Journal*, 49(2) : 291–307.

KIRKPATRICK, S., GELATT, JR., VECCHI, M.P. (1983). Optimization by Simulated Annealing. *Science*, 220 : 671–680.

LABURTHE, F., CASEAU, Y. (1998). SALSA : A Language for Search Algorithms. *Fourth International Conference on the Principles and Practice of Constraint Programming (CP'98)*, Pisa, 1998.

LAPORTE, G., SEMET, F. (2001). Classical Heuristics for the Vehicle Routing Problem. *The Vehicle Routing Problem*, Toth, P., Vigo, D. : SIAM, 109–128.

LIN AND KERNIGHAN, B.W. (1973). An effective heuristic algorithm for the traveling salesman problem. *Operations Research*, 21(2) : 498–516.

LOURENÇO, H., MARTIN, O., STUTZLE, T. (2002). Iterated Local Search, *In F. Glover and G. Kochenberger, editors, Handbook of Metaheuristics*, 321–353, Kluwer Academic Publishers, Norwell, MA.

LUCENA, A. (1990). Time-Dependant Travelling Salesman Problem - The Deliveryman Case. *Networks*, 20 : 753–763.

LUENBERGER, D.G. (1984). *Linear and Nonlinear Programming.* Addison-Wesley Publishing Company.

MARRIOT, K., STUCKEY P.J. (1998). *Programming with Constraints : An Introduction.* The MIT Press.

MEHLHORN, K., NAHER, S. (1995). LEDA : A Platform for Combinatorial and Geometrical Computing. *Communication of the ACM*, 38(1) : 96–102.

MEYERS, S. (1995). *More Effective C++ : 35 New Ways to Improve Your Programs and Designs.* Addison-Wesley.

MICHEL, L., VAN HENTENRYCK, P. (1998). Localizer. Rapport Technique, CS-98-02, Brown University.

MICHEL, L., VAN HENTENRYCK, P. (1999). Localizer : A Modeling Langage for Local Search. *Informs Journal on Computing*, 11(1) : 1–14.

MICHEL, L., VAN HENTENRYCK, P. (2001). Localizer++ : An Open Library for Local Search. Rapport Technique, CS–01–03, Brown University, 2001.

MICHEL, L., VAN HENTENRYCK, P. (2005). *Constraint-Based Local Search.* MIT Press, Cambridge, London.

MLADENOVIC, N., HANSEN, P. (1997). Variable Neighborhood Search. *Computers & Operations Research*, 34 : 1097–1100.

NEMHAUSER, G.L., WOLSEY, L.A. (1988). *Integer and Combinatorial Optimization.* Wiley-Interscience Publication.

NEMHAUSER, G.L, TRICK, M.A. (1998). Scheduling a Major College Basketball Conference. *Operations Research*, 46(1).

PEARL, J. (1984). *Heuristics : Intelligent Search Strategies for Computer Problem Solving.* Addison-Wesley.

PSARAFTIS, H. (1995). Dynamic Vehicle Routing : Status and Prospects. *Annals of Operations Research*, 61 : 143–164.

REEVES C. (1993). *Modern Heuristic Techniques For Combinational Problems.* Oxford Blackwell Scientific Publications, London.

REINELT, G. (1991). TSPLIB - A Traveling Salesman Problem Library. *ORSA Journal on Computing*, 3 : 376–384.

REINELT, G. (1992). Fast Heuristics for Large Geometric Travelling Salesman Problems. *ORSA Journal of Computing*, 4(2) : 206–217.

RESENDE, M., WERNECK, R. (2003). On the implementation of a swap-based local search procedure for the p-median problem. *In Proceedings of the Fifth Workshop on Algorithm Engineering and Experiments (ALENEX'03).*

ROCHAT, Y., TAILLARD, É. (1995). Probabilistic Diversification and Intensification in Local Search for Vehicle Routing. *Journal of Heuristics*, 1 : 147–167.

ROLI, A. ET MILANO, M. (2001). Metaheuristics : A multiagent perspective. Technical Report DEIS-LIA-01-006, University of Bologna (Italy), LIA Series no. 50.

SAVARÉ, S. (1995). *Proceedings of Phylogeny Workshop*, DIMACS Technical Report 95–48, ed. S. Tavaré.

SÉGUIN, R., POTVIN, J.-Y., GENDREAU, M., CRAINIC, T.,C., MARCOTTE, P. (1997). Real-Time Decision Problems : An Operational Research Perspective. *Journal of the Operational Research Society*, 48 : 162–174.

STEIGLITZ, K., WEINER, P. (1968). Some Improved Algorithms for Computer Solution of the Travelling Salesman Problem. *In Proceedings 6th Ann. Allerton Conf. on Communication, Control and Computing*, Department of Electrical Engineering and the Coordinated Science Laboratory, University of Illinois, Urbana III, 814-821.

STROSNIDER, J., PAUL, C. (1994). A Structured View of Real-Time Problem Solving. *AI Magazine*, 15(2), 45-66.

STROUSTRUP, B. (1997). *The C++ Programming Langage (3rd edition).* Addison-Wesley, Longman, Reading, MA.

TAILLARD, E. D., GAMBARDELLA, L.-M., GENDREAU, M., POTVIN, J.Y. (1998). Adaptive Memory Programming : A Unified View of Meta-Heuristics, *EURO XVI Conference Tutorial and Research Reviews booklet* (semi-plenary sessions), Brussels, 1998.

TALBI, E.G. (2000). Une taxinomie des métaheuristiques hybrides. *Troisième Congrés de la Société Francaise de Recherche Opérationnelle et d'Aide à la Décision ROADEF'2000*, Nantes, France.

TROYON, M. (1988). *Quelques Heuristiques et Résultats Asymptotiques pour trois Problèmes d'Optimisation Combinatoire.* Thèse No. 754, École Polytechnique Fédérale de Lausanne, Suisse, 1988.

VAN DER WIEL R. J., SAHINIDIS N. V. (1995). Heuristic Bounds and Test Problem Generation for the Time-Dependent Travelling Salesman Problem. *Transportation Science*, 29(2) : 167–183.

VAN HENTENRYCK, P., MICHEL, L. (2004). Scheduling Abstractions for Local Search. *First International Conference on the Integration of Constraint Programming, Artificial Intelligence and Operations Research (CPAIOR'04)*, 319–335, Nice.

WHITAKER R. (1983). A Fast Algorithm for the Greedy Interchange of Large-Scale Clustering and Median Location Problems. *I*NFOR, 21 : 95–108.

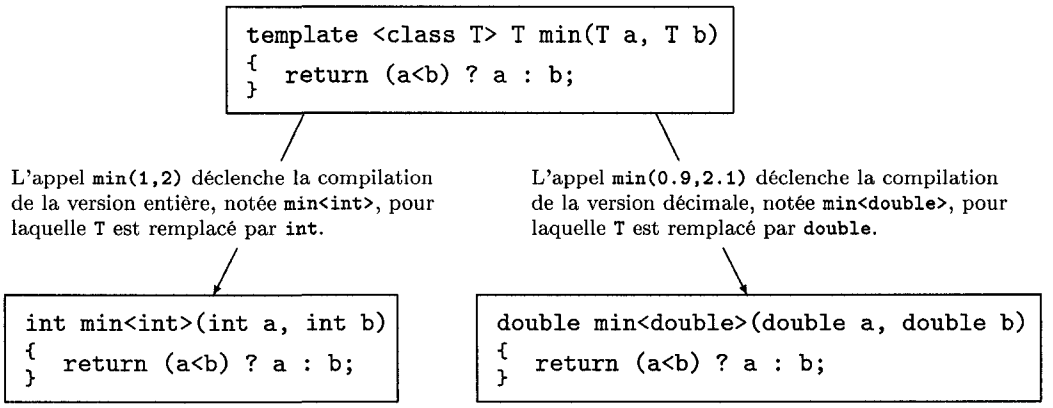
WOODRUFF, D.L., ZEMEL, E. (1993). Hashing Vectors for Tabu Search. *Annals of Operations Research*, 41 : 123–137.

ZILBERSTEIN, S. (1993). *Operational Rationality through Compilation of Anytime Algorithms.* Thèse de Doctorat, University of California at Berkeley.

ANNEXE A : TECHNIQUES GÉNÉRIQUES DE C++

A.1 Composantes gabarits et paramètres-type

Le langage C++ offre la possibilité de paramétrer la programmation de structures et de fonctions au moyen de types abstraits. Le programmeur en signale la construction par le mot-clé `template` suivi de la liste des paramètres-type formels. De telles composantes sont appelées structures ou fonctions gabarits. La programmation d'une composante gabarit permet généralement de factoriser du code pouvant être associé à des structures de données qui, en tant qu'argument de la composante gabarit, restent fonctionnellement similaires. Par exemple, la fonction gabarit `min` suivante peut être utilisée pour tout type dont les valeurs peuvent être comparées deux à deux.



Plusieurs paramètres-type peuvent entrer dans la définition d'une composante gabarit. La structure `Graphe` suivante considère le cas de graphes denses dont chaque sommet présente une demande, et chaque arc, une longueur ainsi qu'une capacité. Elle comporte un premier paramètre pour le type de demande, un second pour le

type de longueur et un troisième pour le type de capacité. Étant donnée une indexation des sommets, `demande[i]` contient la demande associée au sommet i et `longueur[i][j]` (resp. `capacite[i][j]`) pointe sur la longueur (resp. la capacité) associée à l'arc (i, j) si ce dernier appartient au graphe et sur le pointeur nul sinon.

```
template <class D=int, class L=int, class C=int> struct Graphe
{
   int nb_sommets;
   D* demande;
   L*** longueur;
   C*** capacite;
}; [...]
```

Il est possible de spécifier la valeur par défaut d'un paramètre-type. Pour cela, le paramètre apparaît dans la liste `template` suivi de la syntaxe `=T`, où `T` est la valeur de type par défaut. Dans le cas présent, `Graphe<int,int,int>` est le type de `Graphe` par défaut, auquel cas les demandes, longueurs et capacités sont entières.

A.2 Définition et transmission de types

Une composante gabarit est en général reliée à un ensemble de types qui caractérise son fonctionnement interne ainsi que son intégration dans le programme. La structure `Graphe` est par exemple associée à des types de demande, de longueur et de capacité. Lorsqu'une telle composante est passée elle-même en paramètre-type, il est souvent nécessaire de transmettre ses types associés. Par exemple, la fonction gabarit `template <class G> bool somme_demande_nulle(G*)` suivante, vérifiant que la somme des demandes est nulle, nécessite de connaître le type de demande employé. Le C++ permet d'effectuer la transmission des types associés au moyen des mots-clés `typedef`, `::` et `typename`. Le mot-clé `typedef` sert ainsi à créer dans `Graphe`

un synonyme pour chacun de ses types associés. L'opérateur :: permet ensuite aux parties de programme utilisant le graphe d'accéder à ces synonymes. Une composante gabarit qui prend un graphe `G` en paramètre accède ainsi aux types de demande par la syntaxe `typename G::TypeDemande`. Le mot-clé `typename` indique au compilateur que l'expression qui suit immédiatement est une identification de type. L'opérateur :: a une fonction de génitif, `G::TypeDemande` désigne donc le type `TypeDemande` relatif à `G`. La création des synonymes dans `Graphe` est à cet égard indispensable.

```
template <class D, class L, class C> struct Graphe
{
   typedef D TypeDemande;
   typedef L TypeLongueur;
   typedef C TypeCapacite;
   [...]
};
```

```
template <class G> bool somme_demande_nulle(const G& g)
{
   typename G::TypeDemande d_tot=0;
   for (int i=0; i<g.nb_sommets; i++) d_tot += demande[i];
   return (d_tot == 0);
}
```

A.3 Polymorphisme statique

Le polymorphisme statique permet de spécialiser la programmation d'une composante gabarit pour des paramètres-type particuliers. Cette spécialisation est particulièrement intéressante lorsqu'elle permet de coder plus efficacement une composante gabarit au regard de l'utilisation particulière qui en est faite. Dans le cas de la structure `Graphe`, deux simplifications peuvent par exemple se présenter : d'une part, lorsque les capacités sont infinies, et d'autres part lorsque les longueurs d'arcs sont unitaires. Il est possible de traduire ces deux situations en désignant le type `void` pour le paramètre correspondant. Dans ce cas, `Graphe<int,int,void>` (resp.

`Graphe<int,void,int>`) représente un graphe pour lequel les demandes sont entières, les longueurs sont entières (resp. unitaires) et les capacités infinies (resp. entières). Le polymorphisme statique du C++ prend deux formes : la spécialisation totale et la spécialisation partielle. Dans la première, on reprogramme la composante pour un jeu particulier de paramètres-type. Dans la deuxième, on reprogramme la composante pour un sous-ensemble de paramètres-type fixés, les autres restant génériques. Dans l'exemple suivant, on redéfinit la structure de `Graphe` pour le cas où les capacités sont infinies, les types de longueur et de demande restant génériques.

```
template <class D, class L> struct Graphe<D,L,void>
{  int nb_sommets;
   D* demande;
   L*** longueur;
};  [...]
```

Comme pour les structures, on peut spécialiser la définition de fonctions. Dans l'exemple suivant, un algorithme de flot à coût minimum est ainsi défini une première fois de manière générale, puis est amélioré pour les graphes à capacités infinies.

```
template <class D, class L, class C>
void flot_cout_min(Graphe<D,L,C>* g)
{
};  [...] Version générique
```

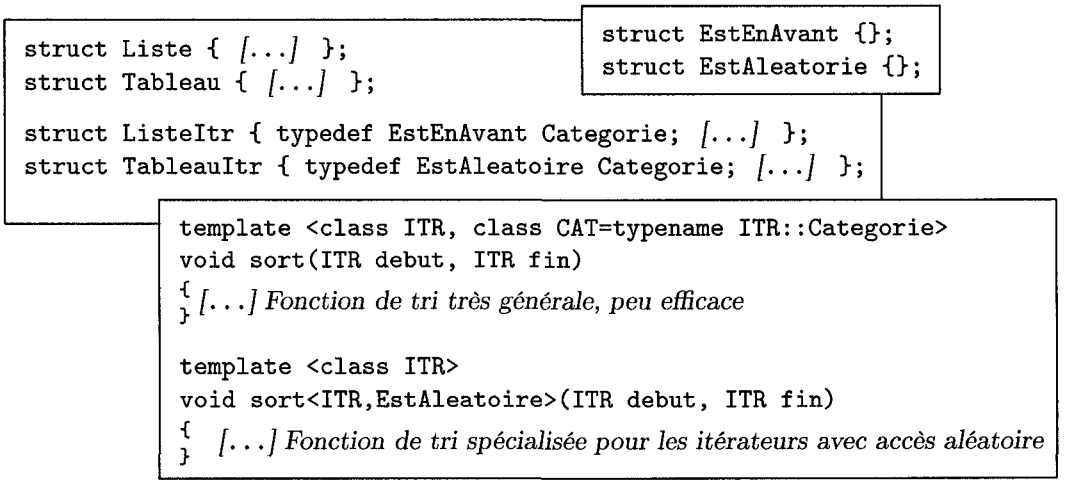
```
template <class D, class L>
flot_cout_min<D,L,void>(Graph<D,L,void>* g)
{
}  [...] Version PCC pour des capacités infinies
```

A.4 Techniques d'étiquetage

Les techniques d'étiquetage consistent à aiguiller explicitement les modes d'intégration d'une composante gabarit par la définition de types-étiquette indicatif de ces modes. Ces techniques sont parfois nécessaires pour prolonger les dispositions géné-

riques du C++. Elles servent par exemple à moduler l'interaction entre composantes, ou encore, à moduler la transmission de valeurs à la périphérie d'une fonction.

Modulation des interactions entre composantes. L'interaction entre algorithmes (recherche, tri, ...) et conteneurs (listes, tableaux, ...) que propose la librairie standard du C++ repose en partie sur des techniques d'étiquetage qu'on applique aux itérateurs. Rappelons qu'un itérateur est une structure qui réunit les opérations de naviguation et d'accès relatives à l'utilisation d'un conteneur. Quel qu'en soit le codage, qui dépend des spécificités du conteneur auquel il s'applique, un itérateur est soit incrémental en avant, en arrière, dans les deux sens, ou bien dispose d'un accès aléatoire. Il est pertinent d'adapter la programmation d'un algorithme donné selon ces quatre catégories. L'exemple suivant montre comment alors distribuer la meilleure version d'algorithme en fonction de la catégorie d'itérateur qu'offre un conteneur.



Deux types-étiquette sont préalablement définies dans le premier bloc : `EstEnAvant` symbolise la catégorie des itérateurs en avant, `EstAleatoire` désigne la catégorie des itérateurs à accès aléatoire. Dans le deuxième bloc, on se donne deux conteneurs, `Liste` et `Tableau`, et dans les deux cas, l'itérateur approprié à l'intérieur duquel un synonyme est employé à désigner la catégorie d'itérateur correspondante. L'itérateur

de liste `ListeItr` est ainsi étiqueté itérateur en avant alors que l'itérateur de tableau `TableauItr` est qualifié d'itérateur offrant un accès aléatoire. Dans le troisième bloc, on définit une version générale d'un algorithme de tri, `sort`, ainsi qu'une version adaptée au cas où le deuxième paramètre-type, qui vaut `ITR::Categorie`, indique un itérateur avec accès aléatoire. Comme l'illustre la fonction principale ci-dessous, cette construction permet de sélectionner la meilleure version d'algorithme en fonction de la catégorie d'itérateur que supporte un conteneur.

```
main()
{  [...] Définition d'une Liste lst et d'un Tableau tbl

   Les méthodes lst.begin et lst.end renvoient des itérateurs en avant
   sort(lst.begin(), lst.end());

   Les méthodes tbl.begin et tbl.end renvoient des itérateurs avec accès aléatoire
   sort(tbl.begin(), tbl.end());
}
```

Dans le second appel de `sort`, le compilateur détecte que les variables sont de type `TableauItr`. Comme `TableauItr::Categorie` est synonyme de `EstAleatoire`, le compilateur développe la version `sort<TableauItr,EstAleatoire>`. À cet égard, rappelons que le C++ distribue la définition gabarit dont les paramètres-type formels couvrent le plus précisément possible les paramètres-type effectifs. Il s'agit ici de la deuxième définition de `sort`, qui correspond comme convenu à la version la plus efficace. Notons que la technique d'étiquetage ne touche pas l'interfaçage entre composantes : les deux appels de `sort` utilisent exactement la même syntaxe.

Modulation des transmissions de valeur. La fonction `min` présentée en début d'annexe peut être spécialisée dans le cadre de paramètres-type correspondant à des objets de grande taille. Celle-ci peut en effet s'avérer inutilisable pour ce genre de type puisque la valeur de ses arguments ainsi que sa valeur de retour sont transmises par copie. Une alternative consiste à transmettre ces valeurs par référence constante. Pourtant, il ne s'agit plus de la même fonction puisque les arguments ne sont plus

les mêmes : d'une part, ceci oblige à écrire la fonction une deuxième fois, et d'autre part, aucune spécification ne permet de savoir quel type utilise quelle version de `min`.

```
template <class T> const T& min(const T& a, const T& b)
{
}  return (a<b) ? a : b;
```

Le code suivant montre comment la technique d'étiquetage permet de programmer directement le type des arguments de `min` en fonction du paramètre-type. Dans le premier bloc, on définit un type-étiquette gabarit `template <class T> Spec` à l'intérieur duquel on spécifie le type de passage en argument d'un objet `T` pour la fonction `min`. La première définition, générique, stipule que par défaut, le passage s'effectue par référence constante. Pour les types qui ne doivent pas suivre ce comportement, on redéfinit la structure. Par exemple, les `int` et les `double` sont passés par valeur. Dans le deuxième bloc, la fonction `min` est finalement écrite d'une seule main. Quel que soit `T`, `Spec<T>::TypePassage` désigne l'argument souhaité.

```
template <class T> struct Spec { typedef const T& TypePassage; };
template <> struct Spec<int> { typedef int TypePassage; };
template <> struct Spec<double> { typedef double TypePassage; };
[...]
```

```
template <class T>
Spec<T>::TypePassage min(Spec<T>::TypePassage a, Spec<T>::TypePassage b)
{
}  return (a<b) ? : b;
```

A.5 Meta-structures et génération de classes

Les méta-structures sont des structures C++ intervenant seulement en phase de compilation. Comme nous verrons dans la section suivante, elles sont passées en argument à des mécanismes qui génèrent ensuite automatiquement des classes qui

autrement devraient être programmées manuellement par l'usager. Au plan sémantique, une méta-structure ne contient généralement que des synonymes de type. La liste de type `TpLst` présentée ci-dessous constitue un des exemples les plus usités de méta-structures. La syntaxe `TpLst<int, TpLst<double, TpLst<char,NUL>>>` fait ainsi référence à la liste de types (`int`, `double`, `char`). En plus de `TpLst`, la librairie METALAB repose par exemple sur des graphes de types qui représentent les différents diagrammes de composition intervenant dans une application de recherche locale.

Structure vide servant de marqueur pour la fin de chaque liste

```
struct NUL {};

template <class T, class U> struct TpLst
{
  typedef T Head;
  typedef U Tail;
};
```

Un exemple : `IntDoubleCharLst` *est synonyme de la liste de types (*`int`, `double`, `char`*)*

```
typedef TpLst<int, TpLst<double, TpLst<char,NUL>>> IntDoubleCharLst;
```

A.6 Programmation en temps de compilation

Les dispositions du C++ pour la programmation générique permettent d'utiliser le compilateur comme interpréteur afin de générer automatiquement en temps de compilation des classes ou des fonctions non triviales. Dans ce contexte, les méta-structures constituent les entrées de l'interpréteur, le polymorphisme statique offre le test conditionnel et la définition d'une classe générique en fonction d'elle-même offre les mécanismes de récursivité. On dispose alors d'une couche de langage s'apparentant aux formes de la programmation fonctionnelle. À titre d'illustration, les deux extraits de programme suivants montrent comment programmer la taille d'une liste de types ou encore comment retourner le type situé à une certaine position dans la liste. Notons que tout ce qui apparaît dans une couche de programme C++ conçu pour

être interprété en temps de compilation est constant. Les entiers sont par exemple représentés de manière fixe par des constantes à l'intérieur d'un `enum`.

Prédéclaration de la structure `Size`

```
template <class> struct Size;
```

Version de `Size` *lorsque la liste est vide*

```
template <> struct Size<NUL> { enum { v = 0 }; };
```

Version générale définie récursivement

```
template <class T, class U> struct Size<TpLst<T,U>> { enum { v=1+Size<U>::v }; };
```

Un exemple : `Size<IntDoubleCharLst>::v` *est une constante égale à* 3

Prédéclaration de la structure `TpAt`

```
template <class, int> struct TpAt;
```

Version de `TpAt` *lorsque la position spécifiée est* 0

```
template <class T, class U> struct TpAt<TpLst<T,U>, 0> { typedef T t; };
```

Version générale définie récursivement

```
template <class T, class U, int i>
struct TpAt<TpLst<T,U>, i> { typedef typename TpAt<U,i-1>::t t; };
```

Un exemple : `TpAt<IntDoubleCharLst,1>::t` *est synonyme de* `double`

Pour conclure, nous faisons à travers deux petits exemples un survol des techniques de programmation en temps de compilation utilisées par METALAB. À cet effet, commençons par remarquer que lors de la déclaration d'une recherche locale, l'ensemble des types de composantes utilisées par l'Explorateur peut être calculé en temps de compilation par transition des types associés et peut être dès lors représenté par une liste de types `ObjLst`. Également, sur la base de techniques similaires au traitement des listes de types, les différents diagrammes de composition ainsi que des fonctions de tri topologiques sur ces diagrammes peuvent être définis. En dérivant un Explorateur de la classe `Holding` suivante avec `ObjLst` en paramètre, celui-ci rassemble automatiquement toutes les composantes nécessaires au fonctionnement de la recherche locale. Notons que chaque composante porte alors le même nom `elt` dans l'Explorateur. Pour distinguer les différents champs, on précède `elt` de l'opérateur génitif comme le montre l'exemple d'accès à l'entier pour `Holding<IntDoubleCharLst>`.

Version simple de la structure `Holding`

```
template <class T> struct Holding { T elt; } ;
```

Version de `Holding` *récursive pour les listes*

```
template <class T, class U>
struct Holding<TpLst<T,U>> : public Holding<U>, public Holding<T> {};
```

Version de `Holding` *pour la liste vide*

```
template <> struct Holding<NUL> { };
```

Un exemple : (dans METALAB, `ObjLst` *remplacerait* `IntDoubleCharLst`)

```
Holding<IntDoubleCharLst> exemple;
exemple.Holding<int>::elt accés à l'entier
```

Supposons maintenant que `AL` représente la liste triée par ordre topologique des Attributs d'un diagramme résiduel d'énumération pour un point d'enchaînement donné. La structure `ScanPropagator` suivante propage alors l'ensemble des mises à jour d'Attributs nécessaires selon l'ordre de maintenance induit par le diagramme et que spécifie `AL`. Les fonctions `call` appelées, qui sont définies de manière générique dans la structure ScanCaller, transmettent finalement l'appel aux véritables fonctions `scan` définies par l'usager en alimentant la liste des arguments nécessaires répertoriés dans l'Explorateur `XPR`. Il est important de noter qu'aucune fonction virtuelle ou pointeur de fonctions ne sont ici utilisés. Il en découle qu'après compilation, la propagation est aussi efficace que si elle avait été conçue sur-mesure.

Spécialisation du foncteur `ScanPropagator` *pour les listes de taille deux*

```
template <class XPR, class AL> struct ScanPropagator<XPR, AttLst, 2>
{
   static void propagate(XPR* xpr)
   {
      ScanCaller<XPR, typename TpAt<AL,0>::t>::call(xpr);
      ScanCaller<XPR, typename TpAt<AL,1>::t>::call(xpr);
   }
};
```

Spécialisation du foncteur `ScanCaller` *pour les attributs composant deux variables*

```
template <class XPR, class ATB> struct ScanCaller<XPR, ATB, 2>
{
   static void call(XPR* xpr)
   {
      xpr->Holding<Atb>.elt.scan (
         xpr->Holding<typename TpAt<typename ATB::VarLst,0>::t>.elt
         xpr->Holding<typename TpAt<typename ATB::VarLst,1>::t>.elt
      )
   }
};
```

ANNEXE B : COMPOSANTES METALAB POUR LE *PVC*

B.1 Une struture préliminaire : TwxHshTbl

Les Caratéristiques `AllTwxHshTbl`, `NstNgbTwxHshTbl` et `BndDiaTwxHshTbl`, présentées par la suite, permettent chacune de regrouper tout ou partie des Mouvements 2-*opt* définis autour de la Solution courante. Ces regroupements, où les Mouvements sont disposés selon leur gain, reposent sur la table de hachage `TwxHshTbl` suivante.

```
struct TwxHshTbl
{ int uk;
  struct Cell { int k; Cell *n, *p; } *tab, **beg;

  TwxHshTbl(int tsz, int bsz) : uk(0), tab(new Cell[tsz]), beg(new Cell*[bsz])
  { for (int k=0; k<bsz; k++) beg[k] = NULL;
  } for (int i=0; i<tsz; i++) tab[i].k = -1;

  ~TwxHshTbl() { delete [] tab; delete [] beg; }

  void insert(Cell& cell, int k)
  { if (uk < (cell.k = k)) uk = k;
    if (cell.n = beg[k]) cell.n->p = &cell;
    beg[k] = &cell, cell.p = NULL;
  }

  void remove(Cell& cell)
  { if (cell.p) cell.p->n = cell.n;
    else if (!(beg[cell.k] = cell.n) && cell.k==uk) while (!beg[--uk]);
    if (cell.n) cell.n->p = cell.p;
    cell.k = -1;
  }
};
```

Figure B.1 – Définition de la table de hachage `TwxHshTbl`

Dans cette table, les mouvements sont triés selon une clé définie par le gain du mouvement majoré de la valeur `2*dst.MAX`, `dst.MAX` étant la distance interponctuelle maximale du problème, de sorte que `0` est la pire clé possible et `4*dst.MAX+1` la meilleure possible. Une ligne d'enregistrement de niveau `k` est implémentée par une liste chaînée contenant l'ensemble des mouvements de clé `k`. La structure `Cell` définie localement dans `TwxHshTbl` constitue chaque maillon de liste. Un maillon contient la clé du mouvement représenté, ainsi que les adresses des cellules suivantes et précédentes dans la ligne d'enregistrement de même clé. Le tableau `beg` amorce les lignes d'enregistrement en pointant dans chaque cas sur la cellule de départ. Si la ligne de niveau `k` est vide, `beg[k]` vaut `NULL`. Le tableau `tab` contient enfin l'ensemble des cellules indépendemment de leur gain. Au moyen d'une convention qui sera propre à `AllTwxHshTbl`, `NstNgbTwxHshTbl` et `BndDiaTwxHshTbl`, l'adresse d'une cellule dans `tab` permettra en fait de retracer les coordonnées du mouvement correspondant. Notons que l'entier `uk` représente la plus grande clé dont la ligne d'enregistrement est non vide. Au départ, toutes les lignes sont vides et `uk` vaut `0`. Les méthodes `insert` et `remove` permettent ensuite, l'une, d'insérer une cellule de `tab` dans la ligne d'enregistrement pour une clé notifiée, et l'autre, de retirer une cellule déjà insérée. Lors de ces deux opérations, la valeur de `uk` est maintenue cohérente.

B.2 La Caractéristique `AllTwxHshTbl`

La Caractéristique `AllTwxHshTbl`, dont les trois extraits de codes suivants donne la définition complète, maintient dans la table de hachage `TwxHshTbl`, dont elle dérive, tous les mouvements 2-*opt* du voisinage étendu défini au chapitre 6. Le tableau `tab`, de dimension N^2, est ici considéré comme une matrice dont l'élément `tab[p*dst.N+q]` correspond aux coordonnées (p, q). Chaque rangée du tableau correspond à une arête de la tournée dont les tableaux `I` et `J` fournissent les extrémités. Le couple de mouvements enregistré dans les cellules de coordonnées (p, q) et (q, p) supprime donc les

arêtes $\{I[p], J[p]\}$ et $\{I[q], J[q]\}$. La partie supérieure (resp. inférieure) de la matrice est réservée aux mouvements qui sont valides (resp. invalides) lorsque $(I[p], J[p])$ et $(I[q], J[q])$ ont la même orientation sur la tournée. À noter que pour chaque ligne ou colonne `r` de `tab`, trois cellules (une sur la diagonale et deux partageant un sommet avec $\{I[r], J[r]\}$ doivent rester déconnectées des lignes d'enregistrement car elles ne correspondent pas à des mouvements du voisinage étendu. Leur position est susceptible de changer en cours d'algorithme.

```
template <class Dst> struct AllTwxHshTbl
  : TwxHshTbl, BuildWith<Tour<Dst>, Dst>, TrackWith<Twex<Dst>, Tour<Dst>, Dst>
{
  short *I, *J;
  ~AllTwxHshTbl() { delete [] I; delete [] J; }

  AllTwxHshTbl(const Tour<Dst>&, const Dst&);
  void track(Twex<Dst>&, const Tour<Dst>&, const Dst&);
};
```

Figure B.2 – Définition de la Caractéristique `AllTwxHshTbl`

L'implémentation du constructeur est aisée. On circule sur la solution courante en affectant progressivement les champs des tableaux `I` et `J`. Parallèlement, les cellules de `tab` sont progressivement insérées.

```
template <class Dst> AllTwxHshTbl<Dst>::
AllTwxHshTbl(const Tour<Dst>& sol, const Dst& dst)
 : TwxHshTbl(dst.N*dst.N,4*dst.MAX+1), I(new short [dst.N]), J(new short [dst.N])
{ for (short p=0; p<dst.N; p++)
  {
    J[I[p] = p] = sol.succ(p);
    for (short q=0; q<p; q++)
    {
      if (J[p]==q || J[q]==p) continue;
      int k = dst(p,J[p]) + dst(q,J[q]) + 2*dst.MAX;
      insert(tab[q*dst.N+p], k - dst(p,q) - dst(J[p],J[q]));
      insert(tab[p*dst.N+q], k - dst(p,J[q]) - dst(J[p],q));
    }
  }
}
```

Figure B.3 – Constructeur de la Caractéristique `AllTwxHshTbl`

```
template <class Dst> void AllTwxHshTbl<Dst>::
track(const Twex<Dst>& mvt, const Tour<Dst>& sol, const Dst& dst)
{
  short p[2];
  for (short i=0; i<2; i++) for (short q=0; q<dst.N; q++)
     if      (I[q] == mvt.c[i][0] && J[q] == mvt.c[i][1]) { p[i] = q; break; }
     else if (I[q] == mvt.c[i][1] && J[q] == mvt.c[i][0]) { p[i] = q; break; }

  for (short i=0; i<2; i++) for (short q=0; q<dst.N; q++)
  {  if (I[q]==I[p[i]] || I[q]==J[p[i]] || J[q]==I[p[i]] || J[q]==J[p[i]]) continue;
     remove(tab[p[i]*dst.N+q]);
  }  if (q != p[1-i]) remove(tab[q*dst.N+p[i]]);

  I[p[0]] = mvt.c[0][0]; J[p[0]] = mvt.c[1][0];
  I[p[1]] = mvt.c[0][1]; J[p[1]] = mvt.c[1][1];

  for (short i=0; i<2; i++) for (short q=0; q<dst.N; q++)
  {  if (I[q]==I[p[i]] || I[q]==J[p[i]] || J[q]==I[p[i]] || J[q]==J[p[i]]) continue;
     int k = dst(I[p[i]],J[p[i]]) + dst(I[q],J[q]) + 2*dst.MAX;
     int dk1 = dst(I[p[i]],I[q]) + dst(J[p[i]],J[q]);
     int dk2 = dst(I[p[i]],J[q]) + dst(J[p[i]],I[q]);
     insert(tab[p[i]*dst.N+q], k - ((p[i]<q) ?dk1 :dk2));
     if (q != p[1-i]) insert(tab[q*dst.N+p[i]], k - ((q<p[i]) ?dk1 :dk2));
  }
}
```

Figure B.4 – Méthode `track` de la Caractéristique `AllTwxHshTbl`

La méthode `track` comprend quatre parties. Premièrement, on détermine les coordonnées du mouvement `mvt` dans `tab`. Deuxièmement, on déconnecte tous les mouvements qui utilisent une des deux arêtes supprimées par `mvt`. Troisièmement, on réaffecte les extrémités d'arête des deux rangées déconnectées. Enfin quatrièmement, on connecte les nouveaux mouvements qui apparaissent sur ces deux rangées.

B.3 Le Serveur `ByGanAllTwxSvr`

Le serveur `ByGanAllTwxSvr` peut traverser une table de hachage `AllTwxHshTbl` afin de visiter dans l'ordre décroissant des gains la totalité du voisinage 2-*opt*. Dans une

méthode taboue, le serveur s'arrête par exemple dés qu'un mouvement non tabou est identifié. Les coordonnées de chaque mouvement dans le formalisme de la structure `Twex` sont simplement calculés étant donnée la position de la cellule pointée dans le tableau `ht.tab`. Dans les méthodes `first` et `next`, chaque fois qu'un mouvement non valide du voisinage étendu est atteint, le Serveur prolonge la traversée. À chaque appel de `next`, le pointeur `cell` se déplace sur la cellule suivante dans la même ligne d'enregistrement si possible et au départ de la prochaine ligne d'enregistrement non vide sinon.

```
template <class Dst> struct ByGanAllTwxSvr
  : EnumerateWith<Twex<Dst>, Tour<Dst>, AllTwxHshTbl<Dst>>
{
  const TwxHshTbl::Cell *cell;
  short p, q;

  void begin(Twex<Dst>&, const Tour<Dst>&, const AllTwxHshTbl<Dst>&);
  bool next (Twex<Dst>&, const Tour<Dst>&, const AllTwxHshTbl<Dst>&);
};
```

Figure B.5 – Définition du Serveur `ByGanAllTwxSvr`

```
template <class Dst> void ByGanAllTwxSvr<Dst>::
begin(Twex<Dst>& mvt, const Tour<Dst>& sol, const AllTwxHshTbl<Dst>& ht)
{
  cell = ht.beg[ht.uk];

  for (;;)
  {
    p = (int)((cell - ht.tab)/(sol.N));
    q = (int)((cell - ht.tab)%(sol.N));

    if ((p<q)^((sol.succ(ht.I[p])==ht.J[p])^(sol.succ(ht.I[q])==ht.J[q]))) break;
    if (cell->n) { cell = cell->n; }
    else { int k=cell->k-1; while (!ht.beg[k]) k--; cell = ht.beg[k]; }
  }

  mvt.c[0][0] = (ht.J[p]==sol.succ(ht.I[p]))? ht.I[p] : ht.J[p];
  mvt.c[1][0] = (ht.J[q]==sol.succ(ht.I[q]))? ht.I[q] : ht.J[q];
  mvt.c[0][1] = sol.succ(mvt.c[0][0]);
  mvt.c[1][1] = sol.succ(mvt.c[1][0]);
}
```

Figure B.6 – Méthode `begin` du Serveur `ByGanAllTwxSvr`

```
template <class Dst> bool ByGanAllTwxSvr<Dst>::
next(Twex<Dst>& mvt, const Tour<Dst>& sol, const AllTwxHshTbl<Dst>& ht)
{
  for (;;)
  {
    if (cell->n) { cell = cell->n; }
    else {
       int k = cell->k;
       do { if (k-- == 0) return false; } while (!ht.beg[k]);
       cell = ht.beg[k];
    }
    p = (int)((cell - ht.tab)/(sol.N));
    q = (int)((cell - ht.tab)%(sol.N));
    if ((p<q)^((sol.succ(ht.I[p])==ht.J[p])^(sol.succ(ht.I[q])==ht.J[q]))) break;
  }
  mvt.c[0][0] = (ht.J[p]==sol.succ(ht.I[p]))? ht.I[p] : ht.J[p];
  mvt.c[1][0] = (ht.J[q]==sol.succ(ht.I[q]))? ht.I[q] : ht.J[q];
  mvt.c[0][1] = sol.succ(mvt.c[0][0]);
  mvt.c[1][1] = sol.succ(mvt.c[1][0]);
  return true;
}
```

Figure B.7 – Méthode next du Serveur ByGanAllTwxSvr

B.4 La Caractéristique NstNgbTwxHshTbl

La Caractéritique `NstNgbTwxHshTbl`, dont les trois extraits de code suivants donne la définition complète, repose sur la même architecture et sur la même interprétation de `tab` que `AllTwxHshTbl`. Cependant, seuls les Mouvements du voisinage étendu qui introduisent dans la tournée au moins un arc dont une des deux extrémités se situe parmi les K plus proches villes de l'autre extrémité sont ici enregistrés. Si l'on compare à `AllTwxHshTbl`, deux nouveaux tableaux, `P` et `Q`, sont ici introduits de sorte que `P[c]` et `Q[c]` indiquent les deux rangées de `tab` où apparaît la ville `c`. À noter qu'une représentation creuse implantée au moyen de listes serait ici plus optimale en terme de mémoire et de temps de calcul asymptotique. La version qu'on propose est cependant plus efficace en pratique pour des tailles de problèmes autorisant la construction d'un tableau de taille quadratique.

```
template <class Dst> struct NstNgbTwxHshTbl
  : TwxHshTbl, BuildWith<Tour<Dst>, Dst>, TrackWith<Twex<Dst>, Tour<Dst>, Dst>
{
  short *I, *J, *P, *Q;
  ~NstNgbTwxHshTbl() { delete [] I; delete [] J; delete [] P; delete [] Q;}

  NstNgbTwxHshTbl(const Tour<Dst>&, const Dst&);
  void track(Twex<Dst>&, const Tour<Dst>&, const Dst&);
};
```

Figure B.8 – Définition de la Caractéristique `NstNgbTwxHshTbl`

```
template <class Dst> NstNgbTwxHshTbl<Dst>::
NstNgbTwxHshTbl(const Tour<Dst>& sol, const Dst& dst)
: TwxHshTbl(dst.N*dst.N,4*dst.MAX+1),
  I(new short [dst.N]), J(new short [dst.N]),
  P(new short [dst.N]), Q(new short [dst.N])
{
  for (short p=0; p<dst.N; p++)
  {
    J[I[p] = p] = sol.succ(p);
    Q[J[P[p] = p]] = p;

    for (short q=0; q<p; q++)
    {
      if (J[p]==q || J[q]==p) continue;
      int k = dst(p,J[p]) + dst(q,J[q]) + 2*dst.MAX;

      D_IpIq = (dst.max[I[p]] < dst.max[I[q]])? dst.max[I[q]] : dst.max[I[p]];
      D_JpJq = (dst.max[J[p]] < dst.max[J[q]])? dst.max[J[q]] : dst.max[J[p]];
      D_IpJq = (dst.max[I[p]] < dst.max[J[q]])? dst.max[I[p]] : dst.max[J[q]];
      D_JpIq = (dst.max[J[p]] < dst.max[I[q]])? dst.max[J[p]] : dst.max[I[q]];

      if (dst(p,q) <= D_IpIq || dst(J[p],J[q]) <= D_JpJq)
        insert(tab[q*dst.N+p], k - dst(p,q) - dst(J[p],J[q]));
      if (dst(p,J[q]) <= D_IpJq || dst(J[p],q) <= D_JpIq)
        insert(tab[p*dst.N+q], k - dst(p,J[q]) - dst(J[p],q));
    }
  }
}
```

Figure B.9 – Constructeur de la Caractéristique `NstNgbTwxHshTbl`

La programmation du constructeur est très similaire au cas de `AllTwxHshTbl`. On vérifie juste avant enregistrement d'un Mouvement si au moins une des arêtes qu'il introduit dans la tournée vérifie la condition portant sur les plus proches voisins.

À noter que pour que le programme compile, la structure de Problème `dst` doit présenter une fonction `max(c)`, qui à une ville `c`, associe la visibilité maximale depuis `c`, autrement dit, la distance de `c` à son $K^{\text{ième}}$ plus proche voisin.

```
template <class Dst> void NstNgbTwxHshTbl<Dst>::
track(const Twex<Dst>& mvt, const Tour<Dst>& sol, const Dst& dst)
{
  short c00=mvt.c[0][0], c01=mvt.c[0][1], c10=mvt.c[1][0], c11=mvt.c[1][1];
  short q, p[2] = {  ((P[c00]==P[c01] || P[c00]==Q[c01])? P : Q)[c00],
                  }; ((P[c10]==P[c11] || P[c10]==Q[c11])? P : Q)[c10]

  for (short i=0; i<2; i++) for (short j=0; j<2; j++)
     for (short r=0; r<dst.nmates[mvt.c[i][j]]; r++)
     {
        q = P[dst.mates[mvt.c[i][j]][r]];
        if (tab[p[i]*dst.N+q].k >= 0) remove(tab[p[i]*dst.N+q]);
        if (tab[q*dst.N+p[i]].k >= 0) remove(tab[q*dst.N+p[i]]);
        q = Q[dst.mates[mvt.c[i][j]][r]];
        if (tab[p[i]*dst.N+q].k >= 0) remove(tab[p[i]*dst.N+q]);
     }  if (tab[q*dst.N+p[i]].k >= 0) remove(tab[q*dst.N+p[i]]);

  I[p[0]] = mvt.c[0][0]; J[p[0]] = mvt.c[1][0];
  I[p[1]] = mvt.c[0][1]; J[p[1]] = mvt.c[1][1];
  ((P[mvt.c[1][0]] == p[1])? P[mvt.c[1][0]] : Q[mvt.c[1][0]]) = p[0];
  ((P[mvt.c[0][1]] == p[0])? P[mvt.c[0][1]] : Q[mvt.c[0][1]]) = p[1];

  for (short i=0; i<2; i++) for (short j=0; j<2; j++) for (short d=0; d<2; d++)
     for(short r=0; r<dst.nmates[mvt.c[j][i]]; r++)
     {
        q = (d==0)? P[dst.mates[mvt.c[j][i]][r]] : q[dst.mates[mvt.c[j][i]][r]];
        if (tab[p[i]*dst.N+q].k >= 0) continue;
        if (I[q]==I[p[i]]||I[q]==J[p[i]]||J[q]==I[p[i]]||J[q]==J[p[i]]) continue;
        int k = dst(I[p[i]],J[p[i]]) + dst(I[q],J[q]) + 2*dst.MAX;
        int dk1 = dst(I[p[i]],I[q]) + dst(J[p[i]],J[q]);
        int dk2 = dst(I[p[i]],J[q]) + dst(J[p[i]],I[q]);
        insert(tab[p[i]*dst.N+q], k - ((p[i]<q)?dk1 :dk2));
     }  insert(tab[q*dst.N+p[i]], k - ((q<p[i])?dk1 :dk2));
}
```

Figure B.10 – Méthode `track` de la Caractéristique `NstNgbTwxHshTbl`

Dans la méthode `track`, on commence par déterminer les rangées de `tab` affectées par `mvt`. Après avoir supprimé les enregistrements présents sur ces deux rangées, on modifie ensuite les champs de `I`, `J`, `P` et `Q` touchés par `mvt`. Finalement, on enregistre les Mouvements entrant dans le voisinage étendu selon le critère de proximité d'une des deux arêtes entrantes. À noter que pour que `track` compile, la structure de Problème `dst` doit présenter deux tableaux, `nmates` et `mates`. Pour chaque ville `c`, `nmates[c]` indique le nombre de villes `d` pour lesquelles `c` (ou `d`) est une des K plus proches villes de `d` (ou `c`). Dans le tableau à deux entrées `mates`, `mates[c]` contient l'ensemble des villes vérifiant la condition de proximité précédente.

Notons pour finir que le Serveur `NstNgbTwxHshTbl` se code exactement de la même façon que `AllTwxHshTbl` à ceci près que le type `AllTwxHshTbl` composé doit être partout remplacé par `NstNgbTwxHshTbl`. Même si cette approche n'a pas été suivie à la section B.3 par souci de clarté, les deux Serveurs peuvent donc en fait être programmés génériquement en fonction du type de table de hachage composé.

B.5 La Caractéristique `BndDiaTwxHshTbl`

La Caractéristique `BndDiaTwxHshTbl`, dont les trois extraits de code suivants donne la définition complète, maintient dans une table de hachage `TwxHshTbl`, dont elle dérive, les mouvements 2-*opt* correspondant à l'inversion d'une sous-séquence de diamètre compris entre 2 et `2*R+1`. En notant chaque mouvement par un couple (c, d) désignant la ville de départ et le diamètre de la sous-séquence à inverser, `tab[c*(2*R+2)+d]` contient la cellule du mouvement (c, d). Dans le constructeur, les mouvements définis autour de la tournée initiale sont ajoutés un à un. Dans la fonction `track`, le tableau `buf`, qui simule un extrait de la nouvelle tournée si `mvt` était appliqué, permet de circuler sur l'ensemble des mouvements affectés. Chaque cellule ainsi traversée est d'abord rétractée de son ancienne ligne d'enregistrement puis insérée dans sa nouvelle.

```
template <class Dst> struct BndDiaTwxHshTbl
  : TwxHshTbl,
    BuildWith<Tour<Dst>, Dst, short>, TrackWith<Twex<Dst>, Tour<Dst>, Dst>
{ short R, *buf;
  ~BndDiaTwxHshTbl() { delete [] buf; }
  BndDiaTwxHshTbl(const Tour<Dst>&, const Dst&, short);
};void track(Twex<Dst>&, const Tour<Dst>&, const Dst&);
```

Figure B.11 – Définition de la Caractéristique BndDiaTwxHshTbl

```
template <class Dst> BndDiaTwxHshTbl<Dst>::
BndDiaTwxHshTbl(const Tour<Dst>& sol, const Dst& dst, short r)
  : TwxHshTbl(dst.N*(2*r+2), 4*dst.MAX+1), R(r), buf(new short [6*r+5]),
{ for (short d=2; d<2*R+2; d++) for (short c=0; c<dst.N; c++)
  { int k = dst(sol.pred(c),c) + dst(sol.succ(c,d-1),sol.succ(c,d))
          - dst(sol.pred(c),sol.succ(c,d-1)) - dst(c,sol.succ(c,d)) + 2*dst.MAX;
    insert(tab[c*(2*R+2)+d],k);
  }
}
```

Figure B.12 – Constructeur de la Caractéristique BndDiaTwxHshTbl

```
template <class Dst> void BndDiaTwxHshTbl<Dst>::
track(Twex<Dst>& mvt, const Tour<Dst>& sol, const Dst& dst)
{ short len = sol.rank[mvt.c[1][0]] - sol.rank[mvt.c[0][1]];
  if (len++ < 0) len += sol.N;
  for (short i=2*R+1, c=mvt.c[0][0]; i>=0; c=sol.pred(c)) buf[i--] = c;
  for (short i=2*R+2, c=mvt.c[1][0]; i<2*R+2+len; c=sol.pred(c)) buf[i++] = c;
  for (short i=2*R+2+len, c=mvt.c[1][1]; i<4*R+4+len; c=sol.succ(c)) buf[i++] = c;
  for (short d=2; d<2*R+2; d++)
  { for (short i=2*R+2-d; i<=2*R+len+2; i++)
    { remove(tab[buf[i]*(2*R+2)+d]);
      int k = dst(buf[i-1],buf[i]) + dst(buf[i+d-1],buf[i+d])
            - dst(buf[i-1],buf[i+d-1]) - dst(buf[i],buf[i+d]) + 2*dst.MAX;
      insert(tab[buf[i]*(2*R+2)+d], k);
    }
  }
}
```

Figure B.13 – Méthode track de la Caractéristique BndDiaTwxHshTbl

B.6 Le Serveur ByGanBndDiaTwxSvr

Le Serveur `ByGanBndDiaTwxSvr` traverse une table de hachage `BndDiaTwxHshTbl` afin de visiter dans l'ordre décroissant des gains le voisinage 2-*opt* restreint à de petits diamètres. Les coordonnées de chaque mouvement dans le formalisme de la structure `Twex` sont simplement calculés étant donnée la position de la cellule pointée dans le tableau `ht.tab`. À chaque appel de `next`, le pointeur `cell` se déplace sur la cellule suivante dans la même ligne d'enregistrement si possible et au départ de la prochaine ligne d'enregistrement non vide sinon.

```
template <class Dst> struct ByGanBndDiaTwxSvr
  : EnumerateWith<Twex<Dst>, Tour<Dst>, BndDiaTwxHshTbl<Dst>>
{
  const typename BndDiaTwxHshTbl<Dst>::Cell *cell;
  short c, d;

  void begin(Twex<Dst>& mvt, const Tour<Dst>& sol, const BndDiaTwxHshTbl<Dst>& ht)
  { cell = ht.beg[ht.uk];
    c = (int)(cell - ht.tab)/(2*ht.R+2);
    d = (int)(cell - ht.tab)%(2*ht.R+2);
    mvt.c[0][0] = sol.pred(mvt.c[0][1] = c);
  } mvt.c[1][0] = sol.pred(mvt.c[1][1] = sol.succ(c,d));

  bool next(Twex<Dst>& mvt, const Tour<Dst>& sol, const BndDiaTwxHshTbl<Dst>& ht)
  {
    if (cell->n) { cell = cell->n; }
    else {
      int k = cell->k;
      do { if (k- == 0) return false; } while (!ht.beg[k]);
      cell = ht.beg[k];
    }

    c = (int)(cell - ht.tab)/(2*ht.R+2);
    d = (int)(cell - ht.tab)%(2*ht.R+2);
    mvt.c[0][0] = sol.pred(mvt.c[0][1] = c);
    mvt.c[1][0] = sol.pred(mvt.c[1][1] = sol.succ(c,d));
    return true;
  }
};
```

Figure B.14 – Définition du Serveur `ByGanBndDiaTwxSvr`