La plate-forme *iOpt* dédiée au développement des systèmes d'optimisation basés sur les méthodes heuristiques

Raphaël Dorne, Christos Voudouris, David Lesaint, Anne Liret, Cédric Ladde
Intelligent Complex Systems Research Group
BTexact Technologies
Adastral Park, B62 Orion Building,
MLB1/PP12, Martlesham Heath
Ipswich, IP5 3RE
Suffolk, United Kingdom
email: {`raphael.dorne`, `chris.voudouris`, `david.lesaint`, `anne.liret`, `cedric.ladde`}`@bt.com`

8 avril 2002

Résumé

Les méthodes méta-heuristiques sont connues pour leur capacité à fournir des solutions de bonne qualité en un temps raisonable pour les problèmes NP-difficiles. Cependant, jusqu'à présent, les tentatives de réalisation d'une plate-forme consacrée à ces méthodes, pouvant pleinement intégrer tous les aspects nécessaires au développement d'un système complet, n'ont pas complètement abouti. Les deux principales raisons pour cela sont, premièrement, le manque d'outils de modélisation pour les problèmes d'optimisation combinatoire, spécialement adaptés aux opérations des méthodes heuristiques. Deuxièmement, le très grand nombre d'algorithmes disponibles dans le domaine des méthodes heuristiques requiert une grande capacité de représentation d'une telle plate-forme, capable d'intégrer un large éventail de techniques heuristiques. Et ce, tout en laissant la possibilité à l'utilisateur de développer ses propres algorithmes en spécialisant des techniques heuristiques génériques préexistantes dans le système.

La plate-forme *iOpt* essaye de répondre à ces questions en fournissant des cadres pour modéliser les problèmes d'optimisation combinatoire et pour synthétiser une grande variété d'algorithmes. En plus de ces cadres, des d'outils graphiques sont également disponibles pour valider le modèle mathématique du problème ainsi que pour faciliter la conception et le contrôle des algorithmes.

1 Introduction

Le projet de recherche *iOpt* (pour "Intelligent Optimisation Toolkit") développé au sein de *BTexact Technologies*, centre de recherche de *British Telecom*, a été motivé par le manque d'outils appropriés pour supporter le développement de méthodes heuristiques en vue de résoudre des applications réelles. L'objectif était, à l'origine, de développer un ensemble d'outils et de librairies logicielles consacrés aux méthodes méta-heuristiques. Appliquant les techniques actuelles de programmation, *iOpt* permet la réutilisation de code en incitant à implémenter de multiples extensions de façon à réduire, autant que possible, le code source frais, à écrire pour toute nouvelle application. De plus, le développement d'applications avec *iOpt* est additif, dans le sens où la plate-forme s'enrichit, après chaque nouvelle utilisation, des nouvelles extensions implémentées diminuant l'effort de développement pour des applications semblables. Enfin, *iOpt* est entièrement écrit en *Java* permettant un déploiement plus facile sur des systèmes d'exploitation et des environnements différents ainsi qu'une meilleure intégration avec les navigateurs et applications pour Internet.

La plate-forme comprend une librairie de contraintes, un cadre (ou "framework" [1]) pour modéliser les problèmes d'optimisation, un cadre pour concevoir les méthodes heuristiques ainsi qu'un ensemble d'outils graphiques pour visualiser les modélisations des problèmes et concevoir les algorithmes.

Dans cet article, nous fournissons seulement une description générale des différentes parties de la plate-forme. Notre objectif principal étant de démontrer la simplicité d'utilisation et la capacité de représentation de la plate-forme que ce soit pour les problèmes ou les algorithmes.

2 La modélisation des problèmes d'optimisation

Un cadre dédié à la modélisation du problème appelé "Problem Modelling Framework" (*PMF*) fait partie de la plate-forme. Le rôle de ce cadre est de fournir une ontologie pour les problèmes d'optimisation combinatoire. *PMF* inclut des classes de base pour énoncer le problème sous la forme d'un *COP* en définissant des variables de décision avec ou sans domaine de valeurs, des contraintes ainsi que la fonction objectif. Les types de données acceptés par *iOpt* sont les booléens, entiers, réels, chaînes de caractères et un type générique *objet*.

Pour supporter la modélisation du problème, des possibilités de programmation déclarative en *Java* sont présentes. Le paradigme suivi est semblable à celui que l'on retrouve dans les librairies *C++* pour la programmation par contraintes telles que *Ilog Solver* [Pug95]. Ces librairies se composent d'un certain nombre de relations prédéfinies qui sont à la disposition de l'utilisateur pour énoncer son problème. Un algorithme efficace de satisfaction de contraintes est alors utilisé pour maintenir d'une manière transparente la validité de ces relations. Contrairement aux outils de programmation par contraintes, les relations disponibles dans *iOpt* sont basées exclusivement sur des contraintes à sens

1. On appelle "framework" en programmation par objets, une architecture, un cadre ou une structure générique permettant le codage d'applications dans un domaine spécifique, par spécialisation ou extension du code préexistant dans le "framework". Par la suite, nous utiliserons le mot cadre pour renvoyer au terme framework.

unique communément appelées des contraintes “one-way” [ZHM+99, ZMS94] ou invariants [MH99, MH98].

La raison du choix des invariants pour soutenir la recherche heuristique, plutôt que des contraintes relationnelles, tient au fait que les invariants sont particulièrement appropriés à la technique de recherche des méthodes heuristiques. Ces dernières exigent une manière efficace d'évaluer l'impact de tout changement de l'état des variables sur les contraintes du problème (c.-à-d. vérification des contraintes), ainsi que sur la valeur de la fonction objectif. Les invariants sont particulièrement adaptés à cette tâche puisqu'ils peuvent incorporer des mécanismes de mise à jour incrémentaux spécialisés pour les différents opérateurs présents dans la librairie. Ceci en plus des algorithmes généraux déjà disponibles pour limiter les parties du graphe de contraintes qui doivent être mises à jour après un changement des variables (principalement les variables de décision dans notre cas).

De plus, la conception d'un mécanisme efficace d'évaluation de mouvements nécessite normalement une personne avec une expertise significative dans le domaine des méthodes heuristiques. Ceci freine considérablement un usage plus répandu de la recherche heuristique. La librairie des invariants a été conçue tout particulièrement pour résoudre ce problème en permettant de réaliser des évaluations de mouvements de manière générique, automatisée et efficace sans requérir une expertise particulière ou des qualifications spéciales en programmation de la part de l'utilisateur. Pour plus d'information sur l'utilisation de la librairie d'invariants au sein de *iOpt*, le lecteur pourra consulter le papier suivant [VDLL01].

Dans la table 1, nous présentons un exemple de code démontrant la simplicité d'utilisation de *PMF* pour modéliser un problème d'optimisation classique: la k-coloration de graphes.

```
Problem createGraphColoring(int k, int nbSommets, int nbAretes, int
aretes[][])
{
    Problem p = new Problem(true,true);
    IntegerDomainVar X[] = new IntegerDomainVar[nbSommets];
    BooleanExp C[] = new BooleanExp[nbAretes];
    p.beginModelChanges();
    for(int i=0;i<nbSommets;i++)
      p.addDecisionVar(X[i] = new IntegerVar(``0..''+k-1));
    for(int i=0;i<nbAretes;i++)
      p.addConstraint(C[i] = Inv.neq(X[aretes[0][i]],X[aretes[1][i]]));
    p.addObjective(Inv.toReal(Inv.minus(nbAretes,Inv.sum(C)));
    p.endModelChanges();
}
```

TAB. 1 – *Exemple de code source pour modéliser un problème avec PMF*

Le code exposé dans la table 1 est complet pour modéliser un problème de k-coloration et démontre comment *PMF* permet de modéliser en très peu de lignes de code un problème d'optimisation combinatoire (COP).

Tout d'abord, un objet *Problem* vide est créé en utilisant le constructeur suivant:

Problem(*boolean* minimisation, *boolean* unfeasibleSolutionAccepted): les paramètres de cette fonction spécifient si le problème d'optimisation est une minimisation ou une maximisation, et si les solutions de l'espace de recherche ne vérifiant pas les contraintes peuvent être également considérées comme valide (contraintes molles). Dans l'exemple de la table 1, le problème de k-coloration est considéré comme une minimisation du nombre de conflits de couleurs (ou contraintes non satisfaites) entre les sommets. Par conséquent l'espace des solutions potentielles ne vérifiant pas les contraintes doit également être considéré (ligne 1).

Ensuite les variables de décision, contraintes et fonction objectif sont créées, puis ajoutées au modèle en utilisant les trois méthodes publiques décrites ci-dessous:
final void addDecisionVar(*DomainVariable* v) ajoute une nouvelle variable de décision au modèle. Avec *PMF*, les variables de décision sont modélisées par des invariants à valeur booléenne, réelle, chaîne de caractères ou objet, variant dans un domaine spécifié par l'utilisateur.
final void addConstraint(*BooleanExp* r) ajoute une contrainte au modèle. Dans *PMF*, une contrainte est modélisée à l'aide d'un invariant booléen. Cet invariant prend en entrée un ensemble de variables de décision et retourne en sortie la valeur de la validité de la contrainte: *Vrai* si la relation est satisfaite, *Faux* sinon.
final void addObjective(*RealExp* v) ajoute un terme réel à la fonction objectif du modèle. La fonction objectif globale est la somme de tous les termes ajoutés à l'aide de cette fonction.

Dans notre exemple, chaque sommet du graphe est représenté par une variable de décision modélisée par une variable invariant entière avec un domaine égal à l'intervalle [0..k-1] (ensemble des couleurs possibles pour chaque sommet)(ligne 6). Le tableau X contient donc l'ensemble des variables de décision.

Chaque arête est modélisée par une contrainte de différence à respecter par les deux variables de décision associés aux deux sommets reliés. L'opérateur *BooleanExp Inv.neq(IntegerExp p, IntegerExp q)* qui retourne un invariant booléen contenant la valeur du résultat de la condition ($p \neq q$), est utilisé pour créer chaque contrainte (ligne 8). Le tableau C contient l'ensemble des contraintes.

Notre objectif étant de minimiser le nombre de contraintes en conflit, la ligne 9 définit la fonction objectif comme la soustraction du nombre de contraintes satisfaites au nombre total de contraintes (nombre d'arêtes). Dans ce cas, la valeur optimale est égale à zéro correspondant à une coloration sans conflits de couleur. L'opérateur *IntegerExp Sum(BooleanExp b[])* retourne dans un invariant entier le nombre de ses invariants booléens en entrée dont la valeur est *Vrai*, *IntegerExp Inv.minus(int i, IntegerExp p)* retourne la valeur de la soustraction i - p et *RealExp Inv.toReal(IntegerExp i)* convertit un invariant entier en un invariant réel.

Maintenant que le modèle est défini, il suffit d'exécuter le source code présenté dans la table 2 pour modifier l'affectation des variables (de décision) du modèle et effectuer une propagation au travers du réseau d'invariants pour obtenir l'évaluation de l'impact de ces changements sur les contraintes et sur la fonction objectif. Seules les parties du réseau d'invariants affectées par ces changements de valeur seront réévaluées.

Les exemples des tables 1 et 2, très simples, servent uniquement d'illustration. En pratique, le cadre *PMF* permet de supporter le développement d'autres cadres plus complexes dédiés à la modélisation d'une famille particulière de problèmes comme, par exemple, le cadre "Scheduling Framework", décrit dans [VDLL01], destiné aux problèmes de planification et d'ordonnancement. D'autres problèmes tels que l'ordonnancement des tâches de

```
void changeCouleur(Problem p, int sommet, int
couleur)
{
   p.beginValueChanges();
   p.getDecisionVar(sommet).setValue(couleur);
   p.endValueChanges();
   p.propagateValueChanges();
   p.saveValueChanges();
}
```

TAB. 2 – *Exemple de code source pour modifier les valeurs des variables avec PMF*

production de voitures (Car Sequencing), l'optimisation de chaînes logistiques (Supply Chain), l'allocation de fréquences ainsi qu'une application réelle concernant la planification des techniciens de *British Telecom*, ont également déjà été modélisés en s'appuyant sur *PMF*.

3 La modélisation des algorithmes

Le cadre *HSF* (pour "Heuristic Search Framework") a été conçu pour être une architecture générique pour la famille des techniques d'optimisation connues sous le nom de méthodes (méta)-heuristiques. Il couvre les méthodes manipulant une seule solution (recherche locale, etc.), les méthodes basées sur une population de solutions (les *Algorithmes Génétiques*, les méthodes évolutives, etc.) et les méthodes hybrides combinant plusieurs algorithmes de types différents (combinaison de la recherche locale avec les *Algorithmes Génétiques*, etc.). Les méthodes heuristiques sont connues pour être efficaces sur un large éventail de problèmes d'optimisation mais elles demeurent difficiles à concevoir, à configurer, et à comparer. De plus, elles tendent à être spécifiques au problème traité exigeant bien souvent une complète re-implémentation pour adresser un nouveau problème.

HSF propose une nouvelle façon de concevoir une méthode heuristique. L'idée principale consiste à pouvoir composer toute méthode heuristique en assemblant un sous-ensemble de "composants de recherche". La liste des composants disponibles s'appuie sur une analyse des techniques existantes rencontrées dans les méthodes heuristiques comme, par exemple, la notion de "voisinage" pour une méthode de recherche locale. Ainsi, une méthode heuristique M_1 est décrite comme contenant les composants a, b et c tandis qu'une autre méthode M_2 contient les composants a, b et d. L'objectif est alors de pouvoir réutiliser les parties communes (ici, a et b) en ne devant implémenter que les parties vraiment spécifiques des méthodes (ici, le composant d pour M_2).

Dans *HSF*, chaque technique heuristique identifiée représente une catégorie de composants qui peut être déclinée en de nombreuses variantes et très facilement étendue par l'utilisateur pour définir sa propre version si nécessaire. Ces catégories de composants sont représentées par des interfaces *Java* tandis que les composants proprement dits sont définis par des classes *Java*. Le concepteur a ainsi la possibilité de créer un algorithme complet et complexe en assemblant des composants prédéfinis ou ceux précédemment étendus par d'autres utilisateurs de *HSF*. Plus précisément, *HSF* repose principalement sur trois concepts: le *composant de recherche*, la *solution heuristique* et le *problème heuristique*.

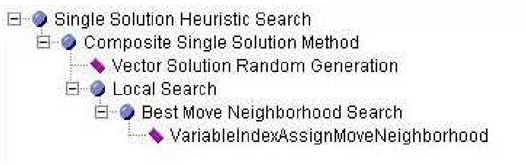

FIG. 1 – *Algorithme générique de Hill-Climber*

Le *composant de recherche* comme expliqué ci-dessus est l'entité de base ou atomique qui est utilisée pour construire une méthode heuristique. Un algorithme complet est un arbre valide de composants de recherche. La figure 1 montre un exemple d'arbre valide de composants pour un simple algorithme de *Hill-Climber*.

La *solution heuristique* est la représentation de la solution d'un problème d'optimisation au sein de *HSF*. Les représentations sous la forme d'un vecteur de variables ou sous la forme d'un ensemble de listes sont actuellement disponibles dans *HSF*. Ces représentations permettent déjà de modéliser les solutions d'un grand nombre de problèmes d'optimisation combinatoire. Par exemple, un vecteur de variables peut modéliser la famille des problèmes *CSP/COP* tandis qu'un ensemble de listes permet de représenter diverses applications dans le domaine de la planification de tâches avec ressources. C'est la solution heuristique qui se charge de modifier les affectations des variables. Dans le cas d'une modélisation faite à l'aide de *PMF*, les modifications seront réalisées en suivant la transaction de modification des variables décrite dans la table 2. Et par conséquent, le développeur de l'algorithme bénéficiera dans ce cas de l'ensemble des fonctionalités automatisées de *PMF* qui lui éviteront de consacrer un temps important à l'implémentation difficile des mises à jour incrémentales nécessaires pour vérifier la validité de ses contraintes ou/et calculer la nouvelle valeur de sa fonction objectif.

Le *problème heuristique* est une interface entre la modélisation du problème d'optimisation énoncé avec *PMF* (ou par d'autres moyens) et *HSF*. Cette interface laisse la possibilité d'appliquer le même algorithme à une famille de problèmes d'optimisation éventuellement énoncés avec différents cadres. Par exemple, si nous considérons deux problèmes d'ordonnancement respectivement modélisés avec *PMF* dans le cadre d'un prototype et avec un *système d'information d'entreprise* dans le cas d'une application réelle, alors l'existence d'un *problème heuristique* pour les problèmes d'ordonnancement, définit au sein de *HSF*, permet de réutiliser le même algorithme pour ces deux modélisations. Seule l'interface entre le *problème heuristique* et le système d'information de l'entreprise devra être implémentée ici.

Dans la version actuelle de *HSF*, les catégories de composants de recherche couvrent de nombreux concepts de base existants dans les méthodes heuristiques manipulant une seule solution tels que la génération de solution initiale, la recherche locale, le mouvement, le voisinage, la stratégie du voisinage, les mécanismes d'aspiration, les restrictions "Tabou", ainsi que la modification dynamique de la fonction objectif et d'autres. En ce qui concerne les méthodes heuristiques manipulant une population de solutions, nous trouvons des composants comme la génération de population initiale, l'opérateur de croisement, l'opérateur de mutation, l'opérateur de sélection, la mutation d'une population, le croisement d'une population, la sélection d'une population, etc.

Pour chaque composant, nous définissons une interface ou protocole de communication précisant son rôle et les conditions à satisfaire pour obtenir de sa part un fonctionnement cohérent. Par exemple, l'interface de communication du composant "génération de solution initiale" se décrit comme un composant recevant en entrée un *problème heuristique* pour retourner en sortie une *solution heuristique* (initiale). De même, le composant "recherche locale" reçoit en entrée une *solution heuristique* sur laquelle il applique son processus d'optimisation itératif pour retourner en sortie une nouvelle *solution heuristique* (optimisée). Pour un bon fonctionnement, ce composant doit contenir un composant "stratégie de voisinage" dans sa liste de composants "enfants". Ce dernier sera utilisé durant le processus itératif pour sélectionner le prochain voisin parmi ceux fournis par le composant de voisinage (*cf.* figure 1 en considérant les composants "Local Search", "Best Move Neighborhood Search", "Variable Index Assign Move Neighborhood").

Il est important de remarquer que dans *HSF*, chaque *composant de recherche* est indépendant du sens global de l'algorithme auquel il appartient. Il considère uniquement les composants qui lui sont parent ou enfants dans l'arbre des composants. Le code de la table 3 correspond à l'implémentation du simple *Hill-Climber* de la figure 1 et démontre la simplicité de la modélisation d'un algorithme avec *HSF*.

```
HeuristicSearch creerHillClimbing()
{
1.    HeuristicSearch myHS = new HeuristicSearch();
2.    SingleSolutionGeneration myG = new VectorRandomGeneration();
3.    LocalSearch myLS = new LocalSearch();
4.    NeighborhoodSearch myNS = new BestMoveNeighborhoodSearch();
5.    Neighborhood myN = new VariableIndexAssignMoveNeighborhood();
6.    CompositeSingleSolutionMethod myCSSM = new CompositeSingleSolutionMethod();
     /* construction proprement dite de l'algorithme */
7.    myNS.add(myN);
8.    myLS.add(myNS); 9.    myCSSM.add(myG);
10.   myCSSM.add(myLS);
11.   myHS.add(myCSSM);
12.   return myHS;
}
```

TAB. 3 – *Exemple de code source pour concevoir un algorithme générique avec HSF*

Notons, qu'en aucun cas, le type de représentation utilisé pour la solution n'est explicitement décrit dans l'algorithme. En fait, le composant qui va générer la solution initiale (VectorRandomGeneration) décide de la représentation à utiliser (vecteur de variables dans cet exemple) par tout ou partie de l'algorithme et la transmettra aux autres composants lors du renvoi de la solution initiale.

Ce point est important car il permet à *HSF* de changer très facilement la représentation de la solution manipulée par un algorithme pour autant que les autres composants de l'algorithme soient compatibles avec ces différentes représentations. Pour les représentations présentes à l'heure actuelle dans *HSF*, un vecteur de variables ou un ensemble de listes, un ensemble de voisinages génériques a été intégrés à la plate-forme, permettant de modifier les valeurs des variables ou les positions des variables dans la représentation. Ainsi, en remplaçant seulement les lignes 2 et 5 de la table 3, par le code de la table 4 on obtient un algorithme de *Hill-Climbing* générique manipulant une solution représentée par un ensemble de listes et effectuant des changements de position des éléments (ou variables de décision) affectés aux listes.

```
2. SingleSolutionGeneration myG = new SequencesRandomGeneration();
5. Neighborhood myN = new PositionMoveNeighborhood();
```

TAB. 4 – *Code à remplacer pour créer un simple Hill-Climber manipulant une représentation sous forme de listes*

Typiquement, cet algorithme peut être utilisé pour résoudre un problème d'allocation des ressources. Dans ce cas, les éléments manipulés sont les tâches et chaque liste représente les tâches allouées à une resource particulière. Il ne manque alors plus que l'implémentation d'un problème d'allocation de resources en utilisant *PMF* qui définira comment les tâches sont autorisées à être affectées (contraintes) et la qualité de toute allocation de resources (fonction objectif), pour obtenir une solution complète pour cette famille de problèmes d'optimisation.

Pour revenir à *HSF*, la flexibilité et la généricité du cadre est évidente dans cet exemple, où la substitution de seulement deux composants a permis de créer un nouvel algorithme complétement différent dont les composants indépendants de la représentation utilisée ont été conservés et ceux dépendants remplacés.

Notons que le changement de la ligne 5 n'est pas obligatoire pour que l'algorithme fonctionne, puisque le voisinage *VariableAssignNeighborhood* est compatible avec la représentation sous forme de listes mais dans ce cas les positions des éléments dans les listes ne seront pas modifiées seulement les valeurs des variables de décisions associées à ces éléments. L'intérêt d'un tel algorithme restant bien entendu à définir !

Une autre conséquence de l'indépendance d'un algorithme vis-à-vis de la représentation de la solution est que *HSF* peut supporter un algorithme contenant des représentations de types différents. Ainsi, nous pouvons imaginer un algorithme décomposé en deux étapes (ou deux sous-arbres de composants) où lors de la première phase, l'algorithme effectue une affectation d'une partie des variables du problème via une représentation sous la forme d'un vecteur de variables, et lors de la deuxième phase, un ordonnancement d'une autre partie des variables utilisant une représentation sous forme de listes. De même, *HSF* permet également de représenter un algorithme manipulant une seule solution durant certaines phases du processus d'optimisation et une population de solutions lors d'autres phases.

L'ensemble des types de composants actuellement disponibles dans *HSF* permettent de synthétiser la plupart des méthodes méta-heuristiques très connues comme le *Recuit Simulé*, la *Recherche Tabou*, la méthode *Guided Local Search*, les *Algorithmes Génétiques* ainsi que les méthodes hybrides. Et ce, à un niveau générique tout en laissant à tout moment la possibilité à l'utilisateur d'intégrer des composants spécialisés pour son problème.

Pour implémenter un nouveau composant, il suffit de suivre le protocole générique ou interface définie pour la catégorie à laquelle appartiendra ce nouveau composant. Par exemple, pour créer un nouveau composant de génération de solution pour le problème de coloration de graphes, le développeur écrit un composant qui reçoit en entrée un problème heuristique et retourne en sortie une solution heuristique. Le contenu proprement dit d'un tel composant est une boîte noire pour *HSF*. Ceci est l'unique contrainte à respecter pour pouvoir facilement intégrer ce composant de génération à tout ancien ou nouvel algorithme. Cet aspect démontre la capacité de spécialisation et de factorisation de *HSF*.

Ainsi, avec seulement deux composants spécifiques à la coloration de graphes *Graph-*

```
2. SingleSolutionGeneration myG = new GraphColoringGreedyGeneration();
5. Neighborhood myN = new GraphColoringNeighborhood();
```

TAB. 5 – *Code à remplacer pour obtenir un algorithme Hill-Climbing spécialisé à la coloration de graphes*

ColoringGreedyGeneration et *GraphColoringNeighborhood* étant respectivement une génération de solution initiale basée sur une méthode gloutonne et un voisinage plus optimisé considérant uniquement les sommets du graphe en conflit[2], il suffit de remplacer les lignes 2 et 5 de la table 3 par le code de la table 5 pour obtenir un algorithme de *Hill-Climbing* spécialisé pour la coloration.

L'algorithme étant créé, le code de la table 6 montre comment exécuter ce simple et générique algorithme sur un problème de coloration de graphes.

```
Problem p = createGraphColoring(5,10, aretes);
myHS.run(new HeuristicProblem(p));
```

TAB. 6 – *Exemple de code source pour exécuter un algorithme avec HSF*

Ce code est étonnament court. En fait, il suffit de passer un problème heuristique au niveau de la racine de l'arbre des composants pour que l'algorithme soit exécuté. Le processus d'exécution commence par la racine puis respecte le protocole de communication de chaque composant rencontré par la suite. La plupart du temps, un tel processus traverse les composants de l'arbre de la gauche vers la droite en suivant un algorithme de parcours en profondeur. La figure 2 montre le flux des données qui a lieu lors de l'exécution de l'algorithme générique de Hill-Climber de la table 3. Evidemment, chaque algorithme possède son propre flux de données en accord avec les composants qui le constituent.

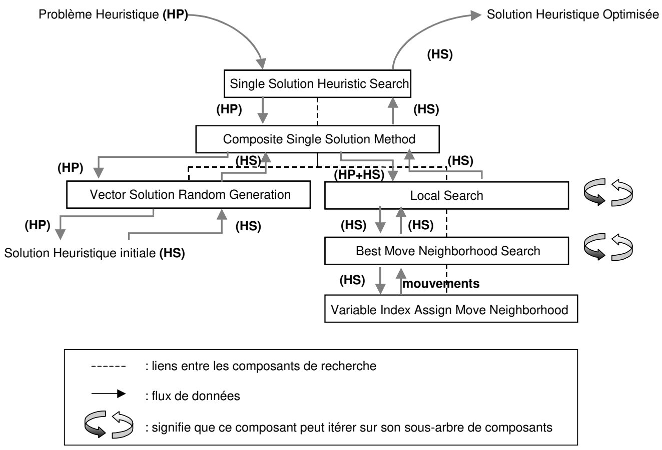

FIG. 2 – *Visualisation des flux de données lors de l'exécution d'un algorithme*

2. Un sommet est dit en conflit si il partage la même couleur avec au moins un des sommets qui lui sont voisins dans le graphe

Pour plus de détails sur la conception et l'exécution des algorithmes avec *HSF*, le lecteur pourra se reporter au papier suivant [DV01].

HSF permet également de modéliser les méthodes hybrides les plus complexes comme celle exposée dans la figure 3 où une méthode hybride de type *Algorithmes Génétiques* utilise une méthode de *Recherche Tabou* comme opérateur de mutation et un croisement spécialisé *UIS* pour la coloration de graphes en plus des composants génériques préexistants.

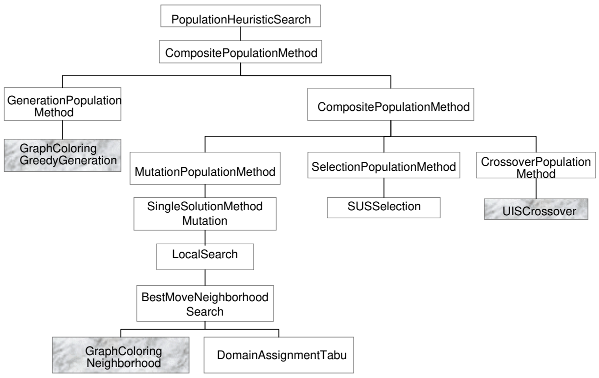

FIG. 3 – *Algorithme hybride pour un problème de coloration de graphes utilisant une méthode de recherche tabou comme opérateur de mutation*

Il apparaît évident que concevoir une méthode heuristique devient une tâche largement simplifiée avec *HSF*: elle ne nécessite que l'assemblage d'un ensemble de composants, même pour une méthode aussi sophistiquée que l'algorithme hybride de la figure 3. De plus, tout nouveau composant peut être incorporé dans la bibliothèque des composants de recherche pour être probablement réutilisé dans des applications semblables par d'autres utilisateurs.

Un autre avantage est que l'on peut facilement mener des comparaisons entre différents algorithmes ou entre différents composants du même type (par exemple comparer les différentes stratégies de voisinage, 15 versions génériques sont actuellement disponibles dans *HSF*). Comme tout nouveau composant du même type peut être directement connecté à l'arbre des composants, nous pouvons rapidement déterminer le réel apport du composant au processus de recherche. Ceci est particulièrement utile pour déterminer quelle est la meilleure variante d'une méthode basée sur une philosophie particulière.

Enfin, une dernière fonctionnalité incluse dans *HSF* est la possibilité de sauvegarder n'importe quel algorithme heuristique composé par l'utilisateur dans un fichier au format *XML*. Un algorithme efficace n'est ainsi jamais perdu et peut être facilement échangé entre différents utilisateurs possédant *HSF*. Pour conclure, nous pouvons dire que *HSF* permet à un utilisateur de se concentrer sur les étapes les plus importantes de conception et de configuration d'un algorithme au lieu de passer une grande partie du temps de développement dans l'implémentation de mécanismes efficaces pour l'évaluation des mouvements.

4 Les outils de visualisation

Pour que les systèmes d'optimisation basés sur la recherche heuristique confirment tout leur potentiel, il est nécessaire de disposer des cadres pour représenter efficacement les problèmes et les algorithmes mais également des outils de visualisation pour permettre à l'utilisateur de participer au processus de résolution des problèmes. Pour répondre à cette question, la plate-forme incorpore des composants de visualisation pour le problème et les algorithmes. Ces outils facilitent le développement des systèmes interactifs d'optimisation pour lesquels des interventions manuelles intelligentes permettent de tester différents scénarios de résolution du problème ou encore de diriger la recherche heuristique en modifiant certains paramètres.

Visualisation du problème

Un outil a été développé pour visualiser les réseaux d'invariants. L'utilisateur peut lire ou modifier les valeurs des invariants dans une table ou encore utiliser une vue graphique pour visualiser la totalité ou partie de la structure du réseau des invariants. Ceci est particulièrement utile pendant les premières phases du développement pour corriger les parties incorrectes du modèle et comprendre également le mécanisme de propagation dans un réseau d'invariants. La figure 4 fournit un exemple de visualisation du réseau d'invariants généré par la plate-forme à partir du code décrit dans la table 1 pour une petite instance de coloration de graphes contenant 5 sommets et 10 arêtes.

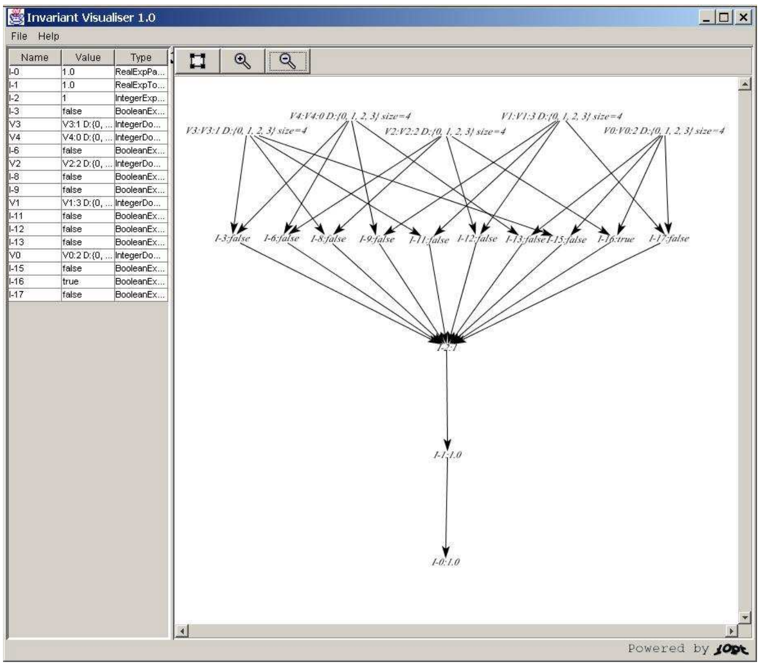

FIG. 4 – *Exemple de visualisation du réseau d'invariants du problème*

Notons la forme arborescente de ce réseau d'invariants avec les 5 variables de décision (partie haute de l'arbre) qui sont les entrées des 10 contraintes (partie du milieu), elles-mêmes entrées de la fonction de coût (partie basse).

Visualisation de l'algorithme

Pour visualiser les algorithmes heuristiques, la plate-forme contient un outil dont l'objectif est de permettre à l'utilisateur de comprendre la structure et le comportement des méthodes. Les fonctionnalités fournies incluent la visualisation de l'arbre des composants de recherche avec la possibilité d'ajuster les paramètres de l'algorithme. L'utilisateur peut parcourir l'arbre des composants de recherche pour toute méthode heuristique et accéder aux paramètres et variables de chaque *composant de recherche*. La figure 5 présente l'interface graphique de cet outil qui est divisée en deux panneaux. Du côté gauche, l'arbre des composants de recherche pour la méthode courante est affiché et du côté droit apparaît un panneau avec tous les paramètres du composant actuellement sélectionné dans le panneau de gauche. Des changements sur ces paramètres peuvent être faits avant de débuter le processus de recherche et même, dans certains cas, en temps réel pendant le processus de recherche.

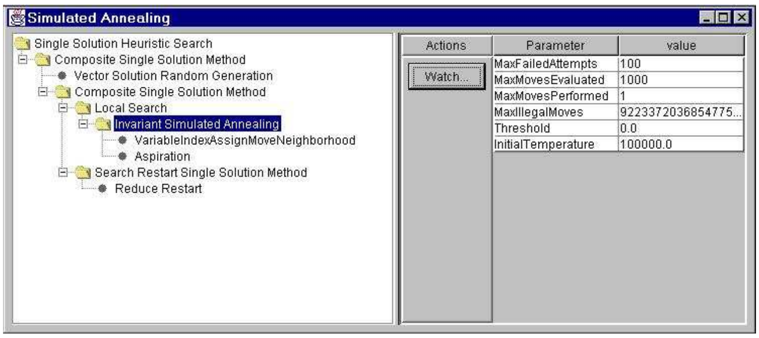

FIG. 5 – *Visualisation des composants d'un algorithme de Recuit Simulé*

Cette dernière fonctionnalité permet à l'utilisateur d'avoir un contrôle interactif sur le comportement de l'algorithme. Par exemple, si une méthode de recherche *Tabou* se retrouve bloquée dans un optimum local, l'utilisateur peut vouloir augmenter (même temporairement) la taille de la liste tabou pour permettre à la recherche d'échapper à cet optimum.

Enfin, un bouton intitulé "Watch" est également présent dans le panneau droit. Ce bouton active un service de surveillance pour le *composant de recherche* concerné. L'observation d'un *composant de recherche* signifie visualiser l'évolution dans le temps de ses différentes variables. Par exemple, l'observation d'un composant de *Recuit Simulé* montrera la valeur objectif de la solution (la courante et la meilleure rencontrée jusqu'a maintenant), sa température courante, la probabilité d'acceptation, etc. (*cf.* figure 6). Cette dernière fonctionnalité permet de visualiser le fonctionnement de l'algorithme au travers de ces composants de recherche et d'exploiter cette information pour faire des changements judicieux au niveau des paramètres.

Cet outil peut être vu comme une sorte de debugger/configurateur pour une méthode heuristique. Après avoir conçu une méthode heuristique de recherche en utilisant *HSF*,

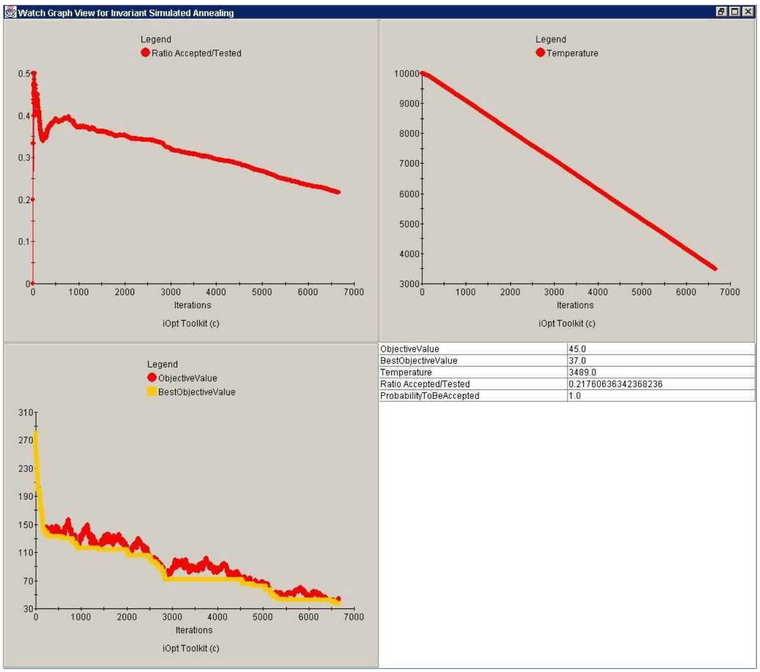

FIG. 6 – *Surveillance et contrôle de l'exécution d'un algorithme*

l'utilisateur peut obtenir une meilleure compréhension de sa méthode en observant, en changeant ou en modifiant dynamiquement les paramètres critiques. Le résultat de cette phase de tests permet la réalisation d'un système final plus efficace.

5 discussions

La plupart des cadres proposés dans la littérature sont basés sur la recherche locale ou sur les *Algorithmes Génétiques*. Pour la recherche locale, on trouve des cadres comme *HotFrame* [FVW99], *Templar* [JMRS98], *LOCAL++* [ASC99], *SCOOP* (http://www.-oslo.sintef.no/SCOOP/), ainsi que le travail de *Andreatta et al.* [ACR98]. Ceux-ci tendent à seulement fournir des templates laissant à l'utilisateur la tâche de définir le modèle pour énoncer son problème plutôt que d'utiliser une représentation basée sur les contraintes comme *PMF*. Une remarque similaire s'applique aux opérateurs de mouvement pour lesquels l'utilisateur doit implémenter lui-même la mise à jour (incrémentale) de la validité des contraintes et de la fonction objectif si il veut obtenir des performances acceptables.

En termes de performances, les résultats sont bien evidemment principalement dépendants de la propagation des changements des valeurs de variables dans le réseau des invariants. Par conséquent, les performances sont différentes pour chaque problème et même pour chaque modèle implémenté avec *PMF*. De manière synthétique, la table 7 fournit le nombre de mouvements testés par seconde lors de l'exécution d'un Hill-climber par la plate-forme pour les problèmes suivants: coloration de graphes, Supply Chain, Vehicle Routing Problem, Workforce Scheduling problem, Job Shop Scheduling problem et

Car Sequencing. Nous considérons un mouvement comme la réalisation complète d'une transaction de changement des valeurs des variables incluant la propagation de ces changements dans le réseau des invariants.

problème	nom de l'instance	informations spécifiques	nombre d'invariants	mouvements évalués par seconde
coloration de graphes	dsjc125.5.col	125 sommets, 3891 arêtes	70912	1933
supply chain	comp.scp	39 unités de production	2373	35981
vehicle routing	vrp.pro	25 véhicules, 100 visites	18330	3475
workforce scheduling	wsp.pro	118 techniciens, 250 tâches	129856	3195
job shop scheduling	abz7.jssp	15 machines, 20 tâches, 15 opérations par tâche (300 opérations au total)	28278	1066
car sequencing	carseq6.txt	9 catégories de voitures, 5 unités de production, 5 lignes à options	966	5431

TAB. 7 – *Ensemble de résultats pour la plate-forme iOpt*

Cette table permet de donner une idée des performances obtenues par la plate-forme sur un ensemble de problèmes. Le lecteur pourra se référer aux publications suivantes pour obtenir des informations plus précises sur les performances globales de la plate-forme pour la coloration de graphes et le Vehicle Routing Problem [DV01, VD02].

6 Conclusion

La plate-forme *iOpt*, présentée dans cet article, correspond au désir de vouloir intégrer un certain nombre de technologies pour offrir un ensemble complet d'outils dédiés au développement rapide de systèmes de résolution basés sur la recherche heuristique. Cette plate-forme comprend une librairie de contraintes, un cadre pour modéliser les problèmes d'optimisation combinatoire fournissant une ontologie simple mais efficace, une architecture générique pour la conception des algorithmes qui s'appuie sur une synthèse des techniques heuristiques existantes, ainsi que divers outils de visualisation intégrés aux modules décrits ci-dessus permettant d'ajuster les modélisations des problèmes et les algorithmes.

Nous avons décrit comment tous ces composants peuvent fonctionner ensemble pour permettre à l'utilisateur de se concentrer sur les tâches de conception de haut niveau du système d'optimisation, automatisant d'une manière efficace les grandes parties de son travail. D'un point de vue commercial, *iOpt* est actuellement en dernière phase de validation sur des applications internes à *British Telecom*. La première version de la plate-forme devrait être disponible au cours du premier semestre 2002. Les premiers résultats sont très prometteurs en termes de réduction des temps de conception du modèle et des algorithmes, de performances des algorithmes et de facilité d'utilisation du système par des développeurs non experts de la plate-forme.

Remerciements

Les auteurs tiennent à remercier les correcteurs pour la pertinence de leurs remarques et commentaires ayant permis d'améliorer la qualité de ce document.

Références

[ACR98] A. Andreatta, S. Carvalho, and C. Ribeiro. An object-oriented framework for local search heuristics. In *Proceedings of TOOLS USA'98*, pages 33–45, 1998.

[ASC99] M. Lenzerini A. Schaerf and M. Cadoli. Local++: a c++ framework for local search algorithms. In *TOOLS Europe '99: Technology of Object Oriented Languages and Systems*, pages 152–161, 1999.

[DV01] R. Dorne and C. Voudouris. HSF: A generic framework to easily design meta-heuristic methods. In *4th Metaheuristics International Conference (MIC' 2001)*, pages 423–428, Porto, Portugal, 2001.

[FVW99] A. Fink, S. Vo, and D. Woodruff. Building reusable software components for heuristic search. In H.-J. Lthi P. Kall, editor, *Operations Research Proceedings 1998*, pages 210–219, 1999.

[JMRS98] M. Jones, G. McKeown, and V. Rayward-Smith. Templar: An object oriented framework for distributed combinatorial optimization. In *UNICOM Seminar on Modern Heuristics for Decision Support*, 1998.

[MH98] L. Michel and P. Van Hentenryck. Localizer. *Tech Report CS-98-02*, March 1998.

[MH99] L. Michel and P. Van Hentenryck. Localizer: A modeling language for local search. *INFORMS Journal of Computing*, 11:1–14, 1999.

[Pug95] J-F. Puget. Applications of constraint programming. In U. Montanari and F. Rossi, editors, *Principles and Practice of Constraint Programming*, pages 647–650, 1995.

[VD02] C. Voudouris and R. Dorne. Integrating heuristic search and one-way constraints in the iopt toolkit. In S. Voss and D. Woodruff, editors, *Optimization Software Class Libraries*, pages 177–192, Wye College (Imperial College), Asford, Kent UK, 2002. Kluwer Academic Publishers.

[VDLL01] C. Voudouris, R. Dorne, D. Lesaint, and A. Liret. iOpt: A software toolkit for heuristic search methods. In Springer-Verlag, editor, *7th International Conference on Principles and Practice of Constraint Programming (CP2001)*, Paphos, Cyprus, 2001.

[ZHM+99] B.V. Zanden, R. Halterman, B.A. Myers, R. McDaniel, R. Miller, P. Szekely, D. Giuse, and D.S. Kosbie. Lessons learned about one-way, dataflow constraints in the garnet and amulet graphical toolkits. *manuscript in preparation*, 1999.

[ZMS94] B.V. Zanden, B.A. Myers, and D. Giuse P. Szekely. Integrating pointer variables into one-way constraint models. *ACM Transactions on Computer-Human Interaction*, 1:161–213, 1994.