INCOP : Une bibliothèque C++ de méthodes incomplètes pour l'optimisation combinatoire version 1.1

Bertrand Neveu
Gilles Trombettoni
Projet COPRIN, CERMICS-I3S-INRIA
Route des lucioles, BP 93, 06902 Sophia Antipolis France
Bertrand.Neveu@sophia.inria.fr, Gilles.Trombettoni@sophia.inria.fr

16 septembre 2004

Résumé

Nous présentons une nouvelle bibliothèque, `INCOP`, d'algorithmes incomplets pour des problèmes d'optimisation combinatoire. Cette bibliothèque offre des méthodes de recherche locale classiques comme le recuit simulé, la recherche avec liste taboue, ainsi qu'une méthode à population, "Go with the winners". Différents problèmes ont été codés, incluant les problèmes de satisfaction de contraintes, le coloriage de graphes, l'affectation de fréquences, l'ordonnancement de voitures dans une chaîne de montage, la clique maximum, le carré latin ...

`INCOP` est une bibliothèque C++ extensible. L'utilisateur peut facilement ajouter de nouveaux algorithmes et coder de nouveaux problèmes. La gestion du voisinage a été étudiée avec soin. D'abord, une sélection de mouvement originale et paramétrée permet d'implanter facilement la plupart des métaheuristiques existantes. Ensuite, différents niveaux d'incrémentalité peuvent être spécifiés pour le calcul du coût d'un mouvement, ce qui augmente grandement l'efficacité en terme de temps de calcul.

`INCOP` s'est montré très performant sur quelques bancs d'essais connus. Il surpasse la plupart des outils concurrents. L'instance difficile de coloriage de graphe `flat300_28` a été coloriée en 30 couleurs pour la première fois par un algorithme de Metropolis standard.

1 Introduction

Les problèmes d'optimisation discrets peuvent être résolus par deux principales classes de méthodes : les méthodes complètes et les méthodes incomplètes. Les algorithmes complets sont basés sur une recherche arborescente avec un schéma de "séparation - évaluation" (Branch and Bound). Plusieurs outils logiciels commerciaux proposent de telles méthodes. Quand un problème d'optimisation peut être modélisé par des contraintes et un critère linéaires, des logiciels de programmation linéaire en nombres entiers (PLNE) peuvent être utilisés. Autrement, des outils de programmation par contraintes, comme `IlogSolver` or `Chip`, peuvent traiter des contraintes quelconques.

Quand l'espace de recherche devient trop grand, ces techniques de recherche systématique sont souvent dépassées par les méthodes incomplètes stochastiques. Ces algorithmes incomplets ne peuvent prouver l'optimalité d'une solution, mais peuvent fournir rapidement une bonne solution. Les méthodes incomplètes les plus courantes sont à base de recherche locale : elles essaient de modifier localement *une* configuration pour améliorer son coût. D'autres méthodes incomplètes explorent l'espace de recherche en manipulant une population de configurations, c'est-à-dire *plusieurs* configuration à la fois.

Implanter efficacement des algorithmes complets requiert un grand effort. Il faut en effet propager les contraintes le plus possible pour réduire la combinatoire et chaque type de contraintes peut nécessiter le développement d'algorithmes spécifiques. Des logiciels commerciaux ont été réalisés et sont utilisés avec succès. Au contraire, il est facile d'implanter une méthode incomplète et aucun effort important n'a été réalisé pour construire un outil commercial. `IlogSolver` a récemment ajouté un module de recherche locale, mais ce dernier est inclus dans la bibliothèque entière et ne peut être utilisé séparément.

Ce manque d'outil a conduit récemment de nombreux chercheurs à construire leurs propres bibliothèques de méthodes incomplètes de recherche. Beaucoup de projets ont été présentés ces dernières années [26]. Citons `iOpt` [27] par British Telecom, `SCOOP` [20] par SINTEF, `Localizer++` [14] à Brown University, `Hotframe` [25] à l'université de Braunschweig, `Discropt` [21] à State University of New York. Philippe Galinier et Jin-Kao Hao [7] ont aussi proposé un cadre pour la recherche locale.

Cependant, pour le moment, toutes ces bibliothèques ne sont pas libres ou pas disponibles. Nous n'avons trouvé qu'une seule bibliothèque libre sur Internet : `EasyLocal++`, à l'Université d'Udine [3], qui implante des méthodes de recherche locale.

Au départ, nous voulions tester une méthode à population et la comparer avec des méthodes de recherche locale dans la même implantation. En effet, une comparaison équitable ne peut être réalisé que sur une même plate-forme logicielle. Nous avons remarqué que l'implantation elle-même, les structures de données utilisées et le degré d'incrémentalité étaient très importants pour les performances. C'est pourquoi nous avons décidé de réaliser une bibliothèque libre, implantant les principales métaheuristiques de recherche locale et des méthodes à population efficaces.

1.1 Principales caractéristiques de la bibliothèque `INCOP`

Sans perte de généralité, notre bibliothèque ne traite que des problèmes de minimisation.

Pour le moment, seul un type de configuration est implanté : une configuration est représentée par un ensemble fixé de variables entières avec des domaines de valeurs connus a priori. De nombreux problèmes d'optimisation combinatoire peuvent être codés dans ce cadre. Il est immédiat de représenter les problèmes de voyageur de commerce (TSP) et de satisfaction de contraintes (CSP). Le coloriage de graphe et certains problèmes d'affectation de fréquences peuvent aussi être vus comme des CSP spécifiques. La bibliothèque étant extensible, d'autres types de configurations peuvent être ajoutés.

Contributions.

Puisque la recherche locale effectue des déplacements dans l'espace des configurations, nous devons proposer un composant pour les mouvements. Pour des raisons d'efficacité, `INCOP` permet aux algorithmes de calculer, lorsque la fonction d'évaluation est décomposable, de manière incrémentale le coût d'une configuration quand un mouvement est réalisé. De plus, INCOP possède les caractéristiques suivantes :

- Une façon originale et paramétrée pour sélectionner un voisin est proposé : c'est une sorte de stratégie de liste de candidats [9], qui a conduit à la définition d'une nouvelle métaheuristique `idw` (Intensification-Diversification Walk) [19].
- Une méthode à population (GWW) est implantée, avec une version hybridée avec de la recherche locale, qui la rend plus efficace (GWW-idw).
- Il est possible d'ajouter une nouvelle méthode et d'implanter un nouveau problème d'optimisation combinatoire.
- Les métaheuristiques de recherche locale les plus courantes sont implantées comme la descente, GSAT [22], le recuit simulé [10], Metropolis [1], la recherche taboue [8].

1.2 Plan

La partie 2 présente l'architecture de la bibliothèque programmée par objets. Cette partie montre aussi comment ajouter une nouvelle métaheuristique, un nouveau problème ou comment définir un nouveau voisinage. La partie 3 détaille les contributions de INCOP : les calculs incrémentaux des coûts des configurations, la sélection paramétrée des mouvements et les algorithmes à population implantés. Les résultats expérimentaux obtenus sur des instances de coloriage de graphe et d'affectation de fréquences sont présentés dans la partie 4 et mettent en valeur les bonnes performances de INCOP. Les travaux en rapport sont présentés dans la partie 5.

2 Architecture

Nous avons choisi une conception par objets et implanté la bibliothèque en C++, en utilisant les méthodes virtuelles et les structures de données fournies par la STL. Bien qu'aucun langage spécifique pour les algorithmes incomplets n'ait été développé (comme OPL [14] ou SALSA [13]), nous allons montrer qu'il n'est pas difficile de définir de nouvelles métaheuristiques ou de nouveaux problèmes.

INCOP comprend six classes principales :

- La classe `Configuration` spécifie le codage d'une solution.
- Spécialiser la classe `Move` permet de définir un nouveau voisinage pour les configurations.
- La classe `NeighborhoodSearch` permet à une configuration de chercher un voisin de façon paramétrée (cf partie 3.3).
- La spécialisation de la classe `IncompleteAlgorithm` définit un nouvel algorithme incomplet.
- La spécialisation de la classe `Metaheuristic` définit une nouvelle métaheuristique.
- La spécialisation de la classe `OpProblem` ajoute un nouveau problème.

On notera que les algorithmes et les problèmes sont séparés en deux hiérarchies de classe. Cela rend possible l'ajout d'un nouveau problème et sa résolution avec toutes les métaheuristiques déjà implantées, ou inversement, l'ajout d'un nouvel algorithme et son test avec tous les problèmes déjà implantés.

Signalons aussi qu'une gestion originale du voisinage est proposée dans INCOP. L'attribut `tabconflicts` des sous-classes de `Configuration`, et les méthodes de la classe `OpProblem` offrent un haut degré d'incrémentalité pour l'évaluation du coût d'un mouvement par les métaheuristiques. Les attributs de la classe `NeighborhoodSearch` permettent une sélection paramétrée d'un mouvement (cf partie 3.3).

2.1 Définir une configuration

Dans la version courante, la classe `Configuration` est seulement spécialisée pour représenter la solution d'un problème `MAX-CSP`, où l'on minimise le nombre (ou une somme pondérée) des violations de contraintes. La hiérarchie de classes courantes est la suivante :

```
Configuration
    CSPconfiguration
         IncrCSPconfiguration
         FullincrCSPconfiguration
```

Un objet de la classe `CSPconfiguration` a trois attributs principaux. La solution est codée dans l'attribut `config` comme un vecteur de valeurs, une pour chaque variable de l'instance de CSP. Un autre attribut `valuation` contient le coût de la configuration.

Un troisième attribut, appelé `tabconflicts`, est défini dans les sous-classes de `CSPconfiguration` pour stocker le nombre de conflits de la valeur courante ou de chaque valeur selon le degré d'incrémentalité choisi. La partie 3.1 détaille la manière dont l'attribut `conflicts` est mis à jour quand un mouvement est effectué.

2.2 Définir un nouveau voisinage

Le voisinage est défini en deux classes principales : `NeighborhoodSearch` et `Move`. Les objets de la classe `Move` contiennent l'information associée à la modification de la configuration (le mouvement) : un calcul du coût de ce mouvement permet le calcul incrémental du coût de la configuration modifiée par ce mouvement.

Pour ajouter un nouveau voisinage, il faut définir une sous-classe de cette classe. Deux sous-classes sont actuellement implantées. La sous-classe `SwapMove` définit un mouvement d'échange dans une configuration de type permutation : les valeurs de deux variables sont échangées. Un tel mouvement est défini par les 2 variables échangées `variable1` et `variable2`.

```
class SwapMove : public Move {
  public :
     int variable1;
     int variable2;
     SwapMove();
     ...
};
```

La sous-classe `CSPMove` réalise la manière la plus courante de définir un voisinage pour les CSP : deux voisins diffèrent par la valeur d'une variable.

Un objet de la classe `NeighborhoodSearch` peut aussi être utilisé pour affiner le voisinage d'un CSP. En se basant sur les travaux de Minton et al [15], `INCOP` propose des variantes pour la définition du voisinage. Deux attributs booléens sont définis à cet effet.

- `var_conflict` : indicateur de la restriction aux variables participant à un conflit.
- `min_conflict` : indicateur de la restriction aux valeurs de la variable qui minimisent les conflits pour cette variable (heuristique Min-conflicts [15]).

On pourrait aussi être plus sélectif dans la détermination des voisins possibles en choisissant des variables ayant une participation forte aux violations de contraintes ou au contraire moins sélectif et implanter des voisinages larges (comme dans la méthode LNS [23]).

2.3 Algorithmes incomplets

`IncompleteAlgorithm` est la classe racine. Elle est subdivisée en `LSAlgorithm` et `GWWAlgorithm`.

```
IncompleteAlgorithm
   LSAlgorithm
   GWWAlgorithm
        ThresholdGWWAlgorithm
        NoThresholdGWWAlgorithm
```

La classe `LSAlgorithm` implante les algorithmes de recherche locale qui effectuent une marche pour une seule configuration. Cette classe a les trois attributs principaux suivants :

- la nombre de mouvements à effectuer,
- un pointeur `nbsearch` vers un objet de la classe `NeighborhoodSearch`, qui permet à l'algorithme de passer d'une configuration à une autre de façon paramétrée,
- un pointeur `mheur` vers un objet `Metaheuristic`.

Les métaheuristiques courantes sont implantées comme des sous-classes de la classe abstraite `Metaheuristic`, citons `TabuSearch`, `GreedySearch`, `SimulatedAnnealing`. D'autres peuvent être facilement ajoutées. Les sous-classes contiennent une méthode spécifique `acceptance` qui décide si un mouvement essayé est accepté et des structures de données spécifiques à la métaheuristique. Par exemple, la classe `TabuSearch` a un attribut pour stocker sa liste taboue.

La classe `GWWAlgorithm` est une spécificité importante de `INCOP` détaillée dans la partie 3.2. Elle implante des algorithmes qui traitent plusieurs configurations à la fois.

Ajouter une nouvelle métaheuristique

Pour implanter une nouvelle métaheuristique, il faut définir une sous-classe de `Metaheuristic`, avec ses données, et trois méthodes :

- `reinit` : initialisation des données de la métaheuristique,
- `acceptance` : condition d'acceptation d'un mouvement candidat,
- `executebeforemove` : mise à jour des données de la métaheuristique (comme la température du recuit simulé ou la liste taboue) avant d'exécuter un mouvement.

Pour illustrer cette partie, nous présentons le code de la métaheuristique d'acceptation à seuil (Threshold Accepting) [6] : un mouvement est accepté si la détérioration de l'évaluation de la configuration courante est inférieure à un seuil ; ce seuil baisse linéairement de `thresholdinit` à 0 pendant la recherche (une marche de longueur `walklength`). A chaque mouvement, le seuil est décrémenté de `delta`.

```
class ThresholdAccepting: public Metaheuristic {
```

```
  public :
    double thresholdinit;
    double delta;
    double thresholdaccept;
    ThresholdAccepting(double maxthreshold, int walklength);
    int acceptance (Move * move, Configuration* config);
    void executebeforemove(Move*  move, Configuration * configuration,
                           OpProblem * problem);
    void reinit();
};

ThresholdAccepting::ThresholdAccepting
                 (double maxthreshhold,int walklength) {
  thresholdinit=maxthreshhold; delta=thresholdinit/walklength;}

int ThresholdAccepting::acceptance (Move * move, Configuration*  config) {
  return ((move->valuation - config->valuation) <  thresholdaccept);}

void ThresholdAccepting::reinit() {
   thresholdaccept = thresholdinit;}

void ThresholdAccepting::executebeforemove (Move * move,
 Configuration * configuration, OpProblem * problem) {
   thresholdaccept -= delta; }
```

La métaheuristique peut alors être directement utilisée en remplissant l'attribut `mheur` de l'objet de classe LSAlgorithm.

```
algo->mheur = new ThresholdAccepting (100., algo->walklength);
```

Quelques ajouts sont aussi nécessaires dans l'interface pour intégrer cette métaheuristique dans la liste des méthodes connues au niveau de la ligne de commande.

Il faut pour cela modifier légèrement le fichier incoputil.cc : il faut ajouter la méthode dans la liste des méthodes connues par :

- ajouter `liste_methodes.push_back("threshold");` dans definir_liste_methodes
- ajouter le nom de la métaheuristique dans la liste partielle de méthodes utilisées dans algo_creation pour séparer les algorithmes de recherche locale simple des algorithmes à population GWW.
- ajouter dans algo_marche une entrée pour lire les arguments de la métaheuristique et une autre pour installer la métaheuristique dans l'attribut `mheur` de l'algorithme.

On peut voir comment c'est fait dans incoputil.cc pour cette métaheuristique pour faire la même chose pour implanter une nouvelle métaheuristique.

2.4 La hiérarchie des problèmes

Les problèmes qui peuvent être implantés dans la bibliothèque sont des problèmes d'optimisation discrète comme, par exemple, le problème du voyageur de commerce. Les problèmes de satisfaction de contraintes (CSP) sont transformés en problèmes MAX-CSP d'optimisation pour lesquels on cherche à minimiser le nombre de contraintes non satisfaites(ou plus généralement un critère calculé sur ces violations de contraintes). Une solution est obtenue quand on a trouvé une configuration avec un coût égal à zéro. Nous avons implanté plusieurs CSP, incluant le coloriage de graphe et les affectations de fréquences.

```
OpProblem : problème d'optimisation
   CSProblem : problème de satisfaction de contraintes
      BinaryCSProblem : CSP binaire
         ColorCSProblem : Coloriage de graphe
         CelarCSProblem : Affectation de fréquences du CELAR (critère additif)
         ExtensionBinaryCSProblem : CSP binaire en extension (traités en MAX-CSP)
         CliqueProblem : Trouver une clique de taille donnée dans un graphe
         NQueenProblem   : problème des N reines
      NaryCSProblem : CSP n-aire en extension (traités en MAX-CSP)
      Latinsquare : carré latin (de base et équilibré)
      Carseq : ordonnancement de voitures (de la CSPLIB)
```

Ajouter un nouveau problème

Si le problème a des contraintes binaires, il peut hériter de la structure `constraints` de la classe `BinaryCSProblem`. Si nécessaire, un voisinage spécifique peut être créé en spécialisant la classe `Move`. Dans ce cas, la méthode `Next_move` doit être redéfinie et remplir un tel mouvement.

Le critère à optimiser est spécifique à un problème donné. Trois méthodes de la classe `OpProblem` calculent le coût d'une configuration. La méthode `config_evaluation` évalue le coût d'une configuration initiale ; `move_evaluation` réalise l'évaluation incrémentale d'un mouvement ; les méthodes `incr_update_conflicts` ou `fullincr_update_conflicts` mettent à jour la structure de données des conflits d'une configuration quand un mouvement est exécuté, selon le degré d'incrémentalité choisi.

Par exemple, les instances de coloriage de graphe sont implantées par la classe `ColorCSProblem`. Nous utilisons une incrémentalité totale pour les évaluations des mouvements.[1].

```
class ColorCSProblem : public BinaryCSProblem {
  public :
    ColorCSProblem (int nvar, int nconst, int nbcol);
    int config_evaluation (CSPconfiguration * config);
    void fullincr_update_conflicts (FullincrCSPConfiguration *config,
                                    Move * move);
    void init_domains (int nbvar, int nbcol);
    Configuration* create_configuration();
};
```

[1]En plus, il faut définir une fonction qui lit les données du graphe à partir d'un fichier au format Dimacs, appelle le constructeur `ColorCSProblem`, et remplit la structure de données des contraintes.

Les instances de coloriage de graphe peuvent être codées comme des instances de MAX-CSP : les variables sont les sommets du graphe à colorier ; le nombre `nbcol` de couleurs avec lesquelles le graphe doit être colorié donne les domaines allant de 1 à `nbcol` ; les sommets liés par une arête doivent être coloriés avec des couleurs différentes : les variables correspondantes doivent prendre des valeurs différentes. Colorier un graphe en `nbcol` couleurs revient à minimiser le nombre de contraintes non satisfaites et à trouver une solution de coût 0. Cette modélisation nous permet de réutiliser une grande part du code défini dans la classe `BinaryCSProblem`.

Les mouvements sont aussi les mêmes que pour un CSP standard : changer la couleur d'un sommet revient à modifier la valeur d'une variable du CSP correspondant, La structure de données des conflits et la méthode `move_evaluation` sont aussi hérités de `BinaryCSProblem`.

Seules les méthodes `config_evaluation` et `fullincr_update_conflicts` ne sont pas immédiates à écrire. Elles sont détaillées ci-après.

La méthode `create_configuration` spécifie quel degré d'incrémentalité est utilisé (ici le degré d'incrémentalité totale).

```
Configuration* ColorCSProblem::create_configuration()
{return (new FullincrCSPConfiguration (nbvar,domainsize ));}
```

La méthode `config_evaluation` calcule le coût d'une configuration et remplit la structure de données des conflits.

```
int ColorCSProblem::config_evaluation (Configuration* configuration)
{int value=0;
 configuration->init_conflicts();
 for(int i=0; i<nbvar; i++)
   for (int j=i+1; j< nbvar; j++)
     if (constraints[i][j])
       {if (configuration->config[i]==configuration->config[j])
          value++;
       configuration->incr_conflicts(i,configuration->config[i],
                                     configuration->config[j],1);
       configuration->incr_conflicts(j,configuration->config[i],
                                     configuration->config[i],1);
       }
 return value;
}
```

La méthode qui met à jour la structure de données des conflits `tabconflicts` est implantée comme suit.

```
void ColorCSProblem::fullincr_update_conflicts
                     (FullincrCSPConfiguration* configuration,Move* move)
{int changed_variable = ((CSPMove*)move)-> variable;
  int changed_value = ((CSPMove*)move)-> value;
```

```
 for (vector<int>::iterator vi= connections[changed_variable].begin();
                        vi!= connections[changed_variable].end(); vi++)
   {configuration->tabconflicts[*vi][configuration->config[changed_variable]]--;
    configuration->tabconflicts[*vi][changed_value]++;
   }
}
```

Ce code illustre les principales structures de données utilisées. Le tableau `config` contient la valeur courante de chaque variable d'une configuration. Les conflits sont stockés dans le tableau bidimensionnel `tabconflicts` indexé par une variable et un indice de valeur. Les contraintes binaires,c.à.d les liens entre les sommets, sont stockées de manière redondante pour des raisons d'efficacité dans deux structures. Un tableau bidimensionnel `constraints` indique si deux variables i et j sont liées par une contrainte.

3 Contributions

Cette partie détaille les caractéristiques originales de `INCOP`. D'abord, les calculs de coûts incrémentaux offerts par notre bibliothèque assurent son efficacité. Ensuite, des algorithmes à population efficaces peuvent être utilisés pour traiter les instances les plus difficiles quand la recherche locale ne suffit plus. Enfin, une sélection originale et paramétrée des mouvements, gérant une sorte de “candidate list strategy” [9] a mené à définir facilement une nouvelle métaheuristique `idw` avec ses variantes.

3.1 Incrémentalité

Il est important de mettre à jour de manière incrémentale la structure `tabconflicts` des objets `Configuration` (voir partie 2.1). On suppose qu'on a à traiter un critère additif comme c'est le cas dans les MAX-CSP. En effet, la contribution de chaque valeur d'une variable à l'évaluation du coût d'une configuration est le nombre de contraintes non satisfaites par cette valeur, les autres variables ayant leur valeur courante. Puisque cette opération est réalisée très souvent (à chaque mouvement testé), il est utile d'évaluer rapidement l'impact d'un mouvement sur le coût de la configuration entière :

1. Dans `CSPconfiguration` : les conflits ne sont pas stockés : il faut calculer le nombre de contraintes non satisfaites par les anciennes et nouvelles valeurs.
2. Dans `IncrCSPconfiguration` : la contribution de la valeur courante est stockée dans `tabconflicts` ; il ne faut calculer que la contribution de la nouvelle valeur.
3. Dans `FullincrCSPconfiguration` : toutes les contributions de toutes les valeurs possibles sont stockées dans `tabconflicts` ; l'évaluation d'un mouvement est immédiate.

Cela est implanté par deux méthodes de la classe du problème à résoudre (sous-classe de `OpProblem`).

- La méthode `move_evaluation` est appelée à la fin de la méthode `next_move`, c.à.d quand un mouvement est essayé.
- Les méthodes `incr_update_conflicts` ou `fullincr_update_conflicts` sont appelées chaque fois qu'un mouvement est effectué.

L'incrémentalité totale implantée dans les instances `FullincrCSPConfiguration` implique que l'effort de calcul est réalisé principalement dans la méthode `fullincr_update_conflicts`. C'est intéressant quand beaucoup de mouvements doivent être testés avant d'en accepter un. Quand le problème est peu dense, ce qui est le cas dans la plupart des instances de coloriage de graphe, la mise à jour est peu coûteuse : elle ne concerne que les valeurs des quelques variables liées par une contrainte à la variable courante. En pratique, ce mode tout incrémental peut faire gagner un ordre de grandeur dans le temps de calcul. La mémoire requise est aussi raisonnable pour les problèmes de coloriage : la taille de la structure de données des conflits est $N \times D$, où N est le nombre de sommets et D est le nombre de couleurs. Les bons résultats expérimentaux décrits dans la partie 4 montrent l'intérêt de cette incrémentalité.

3.2 Algorithmes Go With the Winners

Les algorithmes à population implantés dans `INCOP` sont des variantes de l'algorithme "Go With the Winners" [4]. Plusieurs configurations sont traitées simultanément. Chaque configuration, appelée particule, réalise une marche aléatoire et, périodiquement, les plus mauvaises particules sont redistribuées sur les meilleures. Pour assurer une amélioration de la population, un seuil est baissé durant la recherche et aucun mouvement ne peut passer au dessus du seuil.

L'hybridation avec de la recherche locale est immédiate : au lieu de réaliser une marche aléatoire, chaque particule effectue de la recherche locale. Dans la bibliothèque, l'objet `GWWAlgorithm` possède un attribut, nommé `walkalgorithm`, qui contient un objet `LSAlgorithm`. Ainsi, aucun code supplémentaire n'est nécessaire et toute méthode de recherche locale peut être utilisée. Une de ces hybridations, `GWW-idw` [18], qui hybride `GWW` avec l'algorithme de recherche locale `idw` a donné de très bons résultats (cf la partie 4 pour les résultats expérimentaux et [17] pour une présentation complète de l'algorithme).

Il serait possible d'implanter dans la bibliothèque des algorithmes génétiques classiques ou des algorithmes mémétiques (hybrides d'algorithmes génétiques et de recherche locale). On notera cependant que `GWW` et `GWW-idw` présentent plusieurs avantages en comparaison avec des algorithmes génétiques. D'abord, ils sont plus simples. Ensuite, l'opérateur de croisement des algorithmes génétiques n'est généralement pas très pertinent dans des problèmes très structurés comme les CSP. Enfin, des évaluations incrémentales des mouvements ne peuvent pas être réalisées par l'opérateur de croisement, ce qui rend les algorithmes génétiques difficilement compétitifs pour des problèmes à critère additif.

3.3 Parcours du voisinage

Le point crucial dans les algorithmes de recherche locale est la manière dont les voisins de la configuration courante sont examinés et comment la prochaine configuration est choisie. Un parcours original du voisinage a été implanté dans `INCOP`.

Plusieurs paramètres spécifient ce parcours. Ils sont définis dans les objets de la classe `NeighborhoodSearch`. La classe `LSAlgorithm` possède un attribut qui contient un objet de la classe `NeighborhoodSearch`.

D'abord, il faut spécifier la méthode `is_feasible` qui donne la condition de faisabilité d'un mouvement. Par exemple, dans les algorithmes `ThresholdGWWAlgorithm`, le coût de la configuration doit rester sous le seuil courant. Le voisin est alors dit *atteignable*.

Pour régler finement l'effort d'intensification de la recherche, trois paramètres sont à la disposition de l'utilisateur pour le parcours du voisinage de la configuration courante : `Min_neighbors`, `Max_neighbors` et `No_acceptation`.

1. On teste d'abord `Min_neighbors` voisins pour sélectionner le *meilleur*.

2. Si aucun voisin n'a été accepté par la métaheuristique, on teste d'autres voisins jusqu'à ce qu'un soit accepté ou que `Max_neighbors` voisins aient été examinés.
3. Finalement, si aucun voisin parmi ces `Max_neighbors` n'a été accepté, le paramètre `No_acceptation` avec sa valeur `no_move`, `one_neighbor` ou `bestneighbor` indique quelle prochaine configuration choisir :
   - `0` : la configuration courante n'est pas modifiée,
   - `1` : on choisit un voisin quelconque atteignable parmi les `Max_neighbors` voisins (sans tenir compte du critère d'acceptation de la métaheuristique),
   - `k` : on choisit le meilleur voisin atteignable parmi les k premiers visités et non acceptés .

Dans les algorithmes existants, `Min_neighbors` vaut généralement 0 (par exemple, dans Metropolis), ou `Max_neighbors` (par exemple, dans une recherche avec liste taboue). Ce paramétrage permet de réaliser un compromis entre intensification de la recherche et une exploration qui tente de s'échapper des minimums locaux. Il permet d'implanter facilement des parcours de voisinage classiques. Par exemple :

- Rechercher le meilleur voisin dans le voisinage entier :
  `Min_neighbors` = `Max_neighbors` = taille du voisinage
- Rechercher le premier voisin acceptable dans un échantillon de K voisins :
  `Min_neighbors` = 0, `Max_neighbors` = K.

Remarque

`Min_neighbors` et `No_Acceptation` sont deux paramètres qui sont rarement explicités dans les algorithmes existants. Des études plus approfondies seront nécessaires pour évaluer leur importance. Nous avons déjà observé des comportements très intéressants avec des variantes d'algorithmes existants obtenus avec ce parcours du voisinage. Par exemple, nos expérimentations semblent indiquer que fixer `No_acceptation` à 1 est un bon moyen pour s'échapper des minimums locaux pour les problèmes à voisinage chahuté comme ceux d'affectation de fréquences du CELAR. Nous avons ainsi défini un algorithme simple de recherche locale nommé `idw` pour "intensification diversification walk" avec `Min_neighbors` = 1. Cet algorithme n'a qu'un seul paramètre à régler, le nombre `Max_neigbors` de voisins examinés au plus à chaque pas. Dès qu'un voisin ne détériorant pas l'évaluation est trouvé, il est choisi. Si tous les `Max_neighbors` voisins détériorent l'évaluation du courant, un voisin atteignable parmi eux est choisi et l'on peut ainsi sortir des minimums locaux. Le choix de ce voisin atteignable conduit à deux variantes de l'algorithme (un voisin quelconque `No_Acceptation = 1` ou le meilleur voisin atteignable `No_Acceptation = Max_neighbors`).

Cet algorithme `idw` est détaillé dans [19].

L'algorithme `Configuration_move` (cf Algorithme 1) est implanté comme une méthode de la classe `LSAlgorithm`.

4 Expérimentations

Nous avons réalisé des expérimentations sur des instances difficiles issues principalement de deux catégories de problèmes codés comme des problèmes MAX-CSP pondérés : des instances de coloriage de graphe proposées dans le challenge Dimacs il y a 10 ans et

```
algorithm Configuration_move(x : a configuration) Retourne : a configuration
  i ← 0
  best ? ← (Min_neighbors > 1) or (No_acceptation=best-neighbor)
  best-cost ← +∞ ; one-x ← ⊥
  accepted ? ← false
  while (i < Min_neighbors) or (i < Max_neighbors and not(accepted ?)) do
    x' ← next_move(x)
    if is_feasible(x') then
      if acceptance ?(x, x') then accepted ? ← true
      if best ? and (cost(x') < best-cost) and (accepted ? or i <
      No_acceptation) then
        x-best ← x'
        best-cost ← cost(x')
      end
      if (No_acceptation = 1) then one-x ← x'
    end
    i ← i + 1
  end
  if accepted ? then
    if best ? then
      return x-best
    else
      return x'
    end
  end
  if No_Acceptation = 0 or one-x=⊥ then
    return x
  end
  if No_Acceptation = 1 then return one-x
  if No_Acceptation > 1 then return x-best
end.
```

Algorithm 1: La méthode `Configuration_move`

des problèmes d'allocation de fréquence du CELAR [2]. Tous les tests ont été réalisés sur un `PentiumIII 935 Mhz` avec un système d'exploitation Linux.

4.1 Coloriage de graphe

Nous avons choisi des instances difficiles dans deux catégories du catalogue : les graphes de Leighton avec 450 sommets `le450_15c` (16680 arêtes), `le450_15d` (16750 arêtes), `le450_25c` (17425 arêtes) et `le450_25d` (17343 arêtes), et l'instance plus dense `flat300_28` avec 300 sommets et 21695 arêtes. Pour ces instances, une solution est cachée coloriable en respectivement 15, 25 et 28 couleurs.

L'instance `flat300_28` n'a encore jamais été coloriée en 28 couleurs (ni 29 ou 30 couleurs). A notre connaissance, trois algorithmes incomplets ont réussi à la colorier en 31 couleurs : la version distribuée de l'algorithme Impasse [16], une recherche avec liste taboue [5], et un algorithme hybride à base d'algorithme génétique et de recherche

[2] "Centre d'ELectronique de l'ARmement".

locale [7]. Nous avons réussi à le colorier en 31 couleurs en quelques minutes et en 30 colors en 1.6 heures en moyenne en utilisant un algorithme de Metropolis (c.a.d., un recuit simulé à température constante comme proposé par [1]) et le voisinage `min-conflict` implantant l'heuristique Min-conflicts.

Nous présentons ces résultats dans le tableau 1 : pour chaque instance, nous avons réalisé 10 essais pour un nombre donné de couleurs en changeant la graine du générateur de nombres aléatoires. Nous donnons le temps de calcul moyen, le nombre de conflits moyen restant à la fin de chaque essai et le taux de succès.

	nb-coul	**temps**	**nb-coul**	**conflits**	**temps**	**succès**	**algo**	**voisin.**
`le450_15c`	15	min	15	0	1.1 min	10/10	`GWW-idw`	`var-conflict`
`le450_15d`	15	min	15	1.4	1 min	5/10	`GWW-idw`	`var-conflict`
`le450_25c`	25	min	25	1.5	55 min	1/10	Metropolis	`var-conflict`
`le450_25d`	25	min	25	1.3	58 min	1/10	Metropolis	`var-conflict`
`flat300_28`	31	h	31	0.3	4 min	9/10	Metropolis	`min-conflict`
`flat300_28`	31	h	**30**	1.6	1.6 h	5/10	Metropolis	`min-conflict`

Tab. 1 – Résultats de coloriage de graphes. Les meilleurs résultats connus sont indiqués sur la patrie gauche du tableau(nombre de couleurs, temps) ; les résultats avec `INCOP` sur la partie droite (nombre de couleurs, nombre de conflits (moyenne), temps de calcul (moyenne), taux de succès, algorithme et voisinage utilisés.

4.2 Comparaison avec d'autres bibliothèques

Nous avons aussi comparé `INCOP` avec d'autres bibliothèques de recherche locale sur les mêmes instances de coloriage de graphe. Nous présentons les résultats publiés dans trois articles : `iOpt` [27] implanté en Java, `Easylocal++` [3], Galinier et Hao dans [7]. Ces comparaisons ont été faites sur les graphes aléatoires DSJC du challenge DIMACS avec un nombre de couleurs supérieur au nombre chromatique pour rester conforme avec les résultats déjà publiés. Récemment, quelques résultats sur ces graphes ont été publiés avec `Discropt` [21]. Comme le temps de calcul était limité à 100 s, ces graphes ont été coloriés avec un nombre de couleurs plus grand que celui des expérimentations décrites ci-après et cette approche n'est pas apparue compétitive. Les résultats présentés pour `INCOP` dans le tableau 2 sont les temps de calcul moyens sur 10 essais pour la résolution de chaque instance (la méthode ayant réussi tous ses essais). Nous avons utilisé un algorithme de Metropolis avec température constante et un voisinage `var-conflict`. Nous pensons que les bons résultats que nous avons obtenus sont en grande partie due à l'incrémentalité totale du calcul des coûts réalisée par notre bibliothèque lors de l'exploration du voisinage. Le seul autre système qui mentionne le même type d'incrémentalité est le système de Galinier et Hao [7] qui obtient des performances comparables.

4.3 Problème des N-reines

Sur la page Web de `Easylocal++`, il y a deux exemples de problèmes qui sont fournis : les n-reines et un problème d'emploi du temps. Nous avons téléchargé la bibliothèque `Easylocal++` avec son implantation des n-reines et l'avons fait tourner sur notre machine. Nous avons alors implanté dans `INCOP` ce problème avec le même voisinage (échange dans un codage par permutation). Nous avons obtenu des temps de calcul comparables en utilisant le même algorithme de descente : par exemple, 10 essais de taille 3000,

Problème	Nb coul	INCOP	Easylocal++	Galinier	IOPT
Machine		P III 935	P III 750	Sparc II 333	P III 866
Method		Metropolis	Tabou	Tabou	Tabou ou RS
125.1	6	0.005	0.031		166 (RS)
125.1	5	0.36			44 (T)
125.5	18	0.18	2.65		557 (T)
125.9	44	0.97	13.5		2904 (T)
250.1	9	0.04	0.61		
250.5	30	1.42	52.2	15	
250.5	29	7.81		85	
250.9	75	3.07	118.		
500.1	14	0.18	7.05		
500.5	54	2.43	457		
500.5	52	14.7		14	
500.5	51	96		47	
500.9	140	8.75	1856		

TAB. 2 – Résultats sur les graphes aléatoires DSJC (temps de calcul en secondes)

initialisés aléatoirement, sont résolus en 20.5 s en moyenne avec Easylocal++ et en 21.6 s avec INCOP.

4.4 Affectation de fréquences

Nous avons aussi traité les trois instances les plus difficiles du problème d'affectation de fréquences du CELAR. : les instances celar6, celar7 et celar8. Ces problèmes sont réalistes ayant été construits à partir de sous-parties d'un problème réel. celar6 a 200 variables et 1322 contraintes ; celar7 a 400 variables et 2865 contraintes ; celar8 a 916 variables et 5744 contraintes.

Les variables sont les fréquences auxquelles ont doit affecter une valeur appartenant à un ensemble de fréquences autorisées. (les domaines ont environ 40 valeurs). Les contraintes sont de la forme $|x_i - x_j| = \delta$ ou $|x_i - x_j| > \delta$ (pour éviter les interférences). Notre codage est standard et ne crée que les variables paires dans le CSP avec seulement les contraintes d'inégalités [3].

La fonction objectif à minimiser est une somme pondérée des violations de contraintes. Les contraintes appartiennent à quatre catégories avec différents coûts de violation :

- les contraintes de celar8 ont pour coût 1, 2, 3 ou 4.
- les contraintes de celar6 ont pour coût 1, 10, 100 ou 1000.
- les contraintes de celar7 ont pour coût 1, 100, 10000 ou 10^6.

[2] et [12] décrivent des algorithmes complets basés sur une décomposition du graphe de contraintes et de la programmation dynamique pour résoudre l'instance celar6. Les deux autres instances sont réellement difficiles et aucun algorithme complet n'a fourni de preuve d'optimalité [4]. L'algorithme décrit dans [11] est basé sur un algorithme génétique spécialisé avec un opérateur de mutation codant de la PNLE. L'algorithme décrit dans [28] est de la recherche locale prenant en compte les conflits pour sélectionner les voisins. Les bons résultats obtenus par l'algorithme GWW-idw implanté dans INCOP sont présentés

[3] Une bijection existe entre les variables paires et impaires. Une simple propagation des égalités permet de déduire les valeurs des variables impaires.

[4] Cela peut être expliqué par la taille de celar8 et par le critère de celar7.

sur le tableau 3. Nous avons par ailleurs obtenu la meilleure borne connue 343592 avec une variante de `GWW-idw` utilisant un autre paramétrage dans la gestion du voisinage.

	borne	**ref**	**meilleur (moyen)**	**temps**	**succès**	**algo**
`celar6`	3389	mn (GVS97,KHK99)	3389 (3405.7)	9 mn	4/10	`GWW-idw`
`celar7`	343592	mn (KHK99,VT98)	343596 (343657)	4.5 h	1/10	`GWW-idw`
`celar8`	262	mn (KHK99,VT98)	262 (267.4)	33 mn	2/10	`GWW-idw`

TAB. 3 – Résultats sur les bancs d'essai du CELAR. Les résultats des meilleurs algorithmes sont présentés sur la gauche du tableau ; les résultats avec INCOP sur le côté droit.

5 Autres bibliothèques

Comme il a été mentionné dans l'introduction, de nombreuses bibliothèques ont été développées ces dernières années. Il n'est pas possible de les présenter en détails en quelques lignes. Certaines sont limitées à des méthodes de recherche locale, d'autres fournissent des algorithmes de recherche locale pure et des algorithmes à population comme des algorithmes génétiques hybridés avec de la recherche locale. `Templar` permet d'écrire des algorithmes distribués et d'utiliser du parallélisme. Tous ces outils sont présentés dans [26]. Nous pouvons résumer leurs principales caractéristiques dans le tableau suivant.

Bibliothèque	**langage**	**métaheuristique**	**divers**
`Templar`	C++	tabou, RS, algos gén.	parallélisme
`Neighbor Searcher`	C++ Templates	rec. locale + VNS	
`Hotframe`	C++ extension	rec. locale	dispo. annoncéee
`Esylocal++`	C++	rec. locake	ouvert ; disponible
`Iopt`	Java	rec. locale + algo mémétique	
`Discropt`	C++	rec. locale	dispo. annoncée
`Localizer++`	C++	rec.locale	

TAB. 4 – Bibliothèques de méthodes incomplètes pour l'optimisation combinatoire

Le caractère principal de la plupart de ces bibliothèques est souvent plus la simplicité et l'extensibilité que l'efficacité. Il est difficile de comparer leurs performances car les résultats ont été publiés sur des problèmes différents. Nous avons analysé dans la partie 4.1 les résultats publiés en coloriage de graphe qui montrent que `INCOP` est souvent compétitif.

6 Conclusion

Cet article a présenté la nouvelle bibliothèque `INCOP` écrite en C++ pour l'optimisation combinatoire avec des algorithmes incomplets. Nous avons implanté plusieurs algorithmes de recherche locale et à population originaux et efficaces. Un grand effort a porté sur l'incrémentalité des évaluations des mouvements que nous avons obtenue en maintenant une structure de données spécifique pour les conflits. Une autre caractéristique est notre processus paramétré de gestion du voisinage qui améliore des métaheuristiques existantes et fournit un nouvel algorithme simple de recherche locale `idw`.

Nous pensons qu'il n'existe pas d'algorithme incomplet pouvant résoudre efficacement tous les problèmes. Ainsi il est important de pouvoir tester rapidement différents algorithmes, différents voisinages. Une telle bibliothèque le permet et nous avons obtenu de bons résultats pour les problèmes d'affectation de fréquences du `CELAR` avec `GWW-idw` et pour les problèmes de coloriage de graphe avec `idw` ou `Metropolis`, avec un voisinage de type `min-conflict` ou `var-conflict` selon l'instance résolue. Nous avons, pour la première fois, colorié l'instance `flat300_28` avec 30 couleurs.

La version 1.1 de `INCOP` est téléchargeable.

Références

[1] D. T. Connolly. An improved annealing scheme for the QAP. *European Journal of Operational Research*, 46 :93–100, 1990.

[2] S. de Givry, G. Verfaillie, and T. Schiex. Bounding the optimum of constraint optimization problems. In *Proc. of Constraint Programming CP97*, number 1330 in LNCS, 1997.

[3] L. DiGaspero and A. Schaerf. Easylocal++ : An object oriented framework for flexible design of local search algorithms. Technical Report UDMI/13, Universita degli Studie di Udine, 2000.

[4] Tassos Dimitriou and Russell Impagliazzo. Towards an analysis of local optimization algorithms. In *Proc. STOC*, 1996.

[5] Raphaël Dorne and Jin-Kao Hao. Tabu search for graph coloring, t-colorings and set t-colorings. In I.H. Osman S. Voss, S.Martello and C. Roucairol, editors, *Metaheuristics : Advances and Trends in Local Search Paradigms for Optimization*, pages 77–92. Kluwer Academic Publishers, 1998.

[6] G. Dueck and T. Scheuer. Threshold accepting : a general purpose algorithm appearing superior to simulated annealing. *Journal of Computational Physics*, 90 :161–175, 1990.

[7] Philippe Galinier and Jin-Kao Hao. Hybrid evolutionary algorithms for graph coloring. *Journal of Combinatorial Optimization*, 3(4) :379–397, 1999.

[8] F. Glover. Future paths for integer programming and links to artificial intelligence. In *Computers and Operation Research*, volume 13, pages 533–549, 1986.

[9] F. Glover and M.Laguna. *Tabu Search*. Kluwer Academic Publishers, 1997.

[10] S. Kirkpatrick, C. Gellat, and M. Vecchi. Optimization by simulated annealing. *Science*, 220 :671–680, 1983.

[11] A. Kolen. A genetic algorithm for frequency assignment. Technical report, Universiteit Maastricht, 1999.

[12] A. Koster, C. Van Hoesel, and A. Kolen. Solving frequency assignment problems via tree-decomposition. Technical Report 99-011, Universiteit Maastricht, 1999.

[13] François Laburthe and Yves Caseau. SALSA : A language for search algorithms. In *Proc. CP'98*, number 1520 in LNCS, pages 310–324, 1998.

[14] L. Michel and P. Van Hentenryck. Localizer++ : An open library for local search. Technical Report CS-01-02, Brown University, 2001.

[15] S. Minton, M. Johnston, A. Philips, and P. Laird. Minimizing conflict : a heuristic repair method for constraint satisfaction and scheduling problems. *Artificial Intelligence*, 58 :161–205, 1992.

[16] C. Morgenstern. Distributed coloration neighborhood search. In D. Johnson and M. Trick, editors, *Cliques, Coloring, and Satisfiability*, volume 26 of *dimacs*, pages 335–357. American Mathematical Society, 1996.

[17] Bertrand Neveu and Gilles Trombettoni. Hybridation de gww avec de la recherche locale. In *9èmes Journées Nationales sur la Résolution Pratique de Problèmes NP-Complets, JNPC'03*, pages 277–292, Amiens, France, 2003.

[18] Bertrand Neveu and Gilles Trombettoni. When Local Search Goes with the Winners. In *Fifth International Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimisation Problems, CPAIOR'2003*, pages 180–194, Montréal, Canada, 2003.

[19] Bertrand Neveu, Gilles Trombettoni, and Fred Glover. Idwalk : A candidate list strategy with a simple diversification device. In *Principles and Practice of Constraint Programming, CP'04*, LNCS. Springer, 2004.

[20] P. K. Nielsen. *SCOOP 2.0 Reference Manual.* SINTEF Report 42A98001, 1998.

[21] V. Phan and S. Skiena. Coloring graphs with a general heuristic search engine. In *Computational Symposium of Graph Coloring and Generalizations*, 2002.

[22] B. Selman, H. Levesque, and D. Mitchell. A new method for solving hard satisfiability problems. In *Proc. AI'92*, pages 440–446, 1992.

[23] Paul Shaw. Using constraint programming and local search methods to solve vehicule routing problems. In *Principles and Practice of Constraint Programming, CP'98*, volume 1520 of *LNCS*, pages 417–431, Pise, Italie, 1998. Springer.

[24] S. Voß and D. Woodruff. *Optimization Software Class Libraries.* Kluwer, 2002.

[25] S. Voß and D.L. Woodruff. Hotframe : A heuristic optimization framework. [24], pages 81–154.

[26] Stefan Voß and David L. Woodruff. Optimization software class libraries. [24], pages 1–24.

[27] C. Voudouris and R.Dorne. Integrating heuristic search and one-way constraints in the iOpt toolkit. [24], pages 177–192.

[28] C. Voudouris and E. Tsang. Solving the radio link frequency assignment problem using guided local search proceedings. In *NATO Symposium on Radio Length Frequency Assignment, Sharing and Conservation Systems (Aerospace)*, 1998.