LS(Graph): Un cadre de recherche locale pour des problèmes d'optimisation sous contraintes sur des graphes

Pham Quang Dung[(1)] **Yves Deville**[(1)] **Pascal Van Hentenryck**[(2)]

[(1)]Département d'Ingénierie Informatique
UCLouvain
B-1348 Louvain-la-Neuve, Belgique
{quang.pham,Yves.Deville}@uclouvain.be

[(2)]Brown University, Box 1910
Providence, RI 02912, USA
pvh@cs.brown.edu

Résumé

COMET est un langage de programmation orienté objet supportant l'architecture de recherche locale basée sur les contraintes. Ce papier propose un cadre LS(Graph) en COMET qui simplifie la modélisation ainsi que les algorithmes de recherche locale pour résoudre des problèmes d'optimisation sous contraintes sur des graphes .

1 Introduction

Des problèmes d'optimisation sous contraintes sur des graphes (GCSOPs) apparaissent dans de nombreuses applications réelles dans des domaines tels que les télécommunications, les réseaux de transport [7] [12] [13] [16] [22] [24], la disposition des services [15], les systèmes d'exploitation [18], la récupération d'informations [5], etc.

COMET [9] est un langage de programmation de haut niveau fournissant des abstractions de modélisation et de contrôle innovatrices pour la recherche locale. COMET offre une architecture basée sur les contraintes pour des programmes de recherche locale organisés en deux composants indépendants : un composant déclaratif qui modélise le problème en terme de contraintes et de fonctions et un composant de recherche qui spécifie des algorithmes de recherche locale heuristiques et metaheuristiques. Grâce à cette architecture, un algorithme de recherche locale est de haut niveau, compositionel et modulaire. Il est aisé d'ajouter de nouvelles contraintes, ainsi que de modifier ou de supprimer des contraintes existantes sans devoir prêter attention à l'impact global de ces changements. D'ailleurs, COMET permet aux programmeurs d'expérimenter plusieurs stratégies de recherche heuristiques et metaheuristiques sans modifier la modélisation du problème.

Ce papier propose un cadre, appellé LS(Graph), qui a pour objectif de simplifier la modélisation de GCSOPs en COMET. Le programmeur ne doit plus se préoccuper de structures de données ou d'algorithmes sophistiqués sur des graphes (par exemple, le calcul du nombre de composants connexes d'un graphe non-dirigié ou le diamètre d'un arbre) parce que ceux-ci sont déjà implémentés dans LS(Graph). Le programmeur peut alors se concentrer sur la modélisation et sur l'exploitation de diverses stratégies de recherche heuristiques et metaheuristiques. LS(Graph) est conçu et implémenté en COMET (le code source COMET de la première version de LS(Graph) contient environ 10,000 lignes) et inclut des contraintes spécifiques sur des graphes telles que la contrainte *Connected(G)* qui spécifie qu'un graphe non-dirigé G est connexe, *VarTree* qui contraint le graphe à être un arbre. Cette dernière est fondamentale pour modéliser des problèmes de découverte d'arbres sous contraintes. Les contraintes *Connected(G)* et *VarTree* sont implémentées en utili-

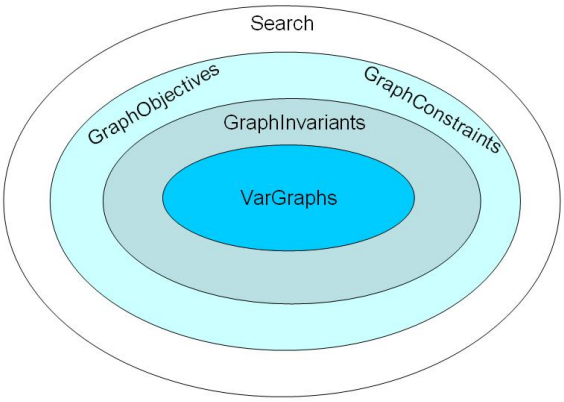


FIG. 1 – Architecture de LS(Graph)

sant des structures de données sophistiquées et des algorithmes (incrémentaux) sur des graphes. `LS(Graph)` est une environnement ouvert et extensible, permettant la conception et l'implémentation de nouveaux composants tels que de nouvelles contraintes sur des graphes.

Le reste de ce papier est organisé comme suit. Dans la section 2, nous introduisons le modèle de calcul de `LS(Graph)`. Une brève description de `LS(Graph)` est presentée en section 3. La section 4 montre une application de `LS(Graph)` : la résolution du problème de Edge-weighted k-cardinality Tree (KCT). Nous montrons que par rapport à un algorithme de recherche tabou implémenté avec KCTLib [10], une exploration sur un voisinage plus large donne des resultats compétitifs sur des graphes euclidiens générés aléatoirement. Enfin, la section 5 résume nos contributions et présente quelques directions de recherche.

2 Modèle de calcul

Tout comme l'architecture de la Constraint-Based Local Search [9], l'architecture de `LS(Graph)` est organisée en quatre couches, présentées en Figure 1.

La *VarGraph* G, appellée variable de graphe, est une abstraction representant un graphe dynamique qui peut être modifié au sein d'un intervalle $[glb, lub]$ ($glb \subseteq G \subseteq lub$). Une mise-à-jour d'une *VarGraph* est réalisée sur le graphe dynamique correspondant. Cette mise-à-jour peut correspondre à l'insertion et la suppression de *Noeud*s ou d'*Arête*s. A chaque étape du calcul, l'ensemble de noeuds (arêtes) appartenant à la borne supérieure (lub) est partitionné en deux classes : les noeuds (arêtes) qui appartiennent au graphe courant et les noeuds (arêtes) qui n'appartiennent pas au graphe courant mais qui peuvent être ajoutés au graphe courant. Les noeuds (arêtes) de la deuxième classe sont appellés *noeuds optionnels* (arêtes *optionnelles*) du graphe courant. On recherche habituellement un sous-graphe d'un graphe donné g qui satisfait certaines contraintes. Le domaine de la *VarGraph* est alors $[\oslash, g]$. La plupart de GCSOPs présenté dans litérature consiste à trouver un sous-graphe d'un graphe non-dirigé donné satisfaisant des contraintes. Dès lors, dans cette première version de `LS(Graph)`, nous considérons $glb \equiv \oslash$ et lub est un graphe non-dirigé fixé ; *VarGraph* représente donc un graphe non-dirigé dynamique.

Le concept de *GraphInvariant* permet de décrire des objets maintenant des propriétés sur des graphes, comme par exemple le chemin simple le plus long, l'ensemble de ponts, le nombre de composants connexes, etc. d'un graphe. L'ensemble des arêtes du graphe et l'ensemble des arêtes qui n'appartiennent pas au graphe mais qui peuvent être ajoutées au graphe sont aussi des *GraphInvariant*s. Ils sont très utiles pour la modélisation des GCSOPs.

Le concept de *GraphConstraint* permet de décrire des objets qui représentent des contraintes sur des *VarGraph*s. Il existe de nombreuses contraintes sur des graphes, comme par exemple *Tree(G)* qui spécifie que le graphe G est un arbre, la contrainte *SimplePath(G,s,t)* qui spécifie que le graphe G est un chemin simple du noeud s au noeud t.

Dans beaucoup de problèmes sur des graphes, on considère non seulement la faisabilité mais aussi l'optimalité. Dans ce cas, on préfère maintenir un objet qui retourne la valeur d'une fonction objectif et qui permet d'évaluer l'impact des mouvements locaux sur cette fonction objectif. Le concept de *GraphObjective* permet de décrire des objets qui représentent des fonctions objectifs sur des *VarGraph*s, comme par exemple la somme des poids de toutes les arêtes d'un graphe, le nombre de noeuds, etc.

Tout comme les objets différentiables de `COMET`, les *GraphConstraint*s et les *GraphObjective*s sont des objets différentiables qui maintiennent un nombre de propriétés et qui peuvent être questionnés pour évaluer l'impact des mouvements locaux sur ces propriétés.

2.1 Le Modèle

Dans la littérature, la plupart des GCSOPs consistent à trouver un ou quelques sous-graphes d'un graphe non-dirigé donné, satisfaisant des contraintes, et optimisant une fonction objectif. Nous considérons donc dans cette première version de `LS(Graph)` des GCSOPs où seules des variables de graphes sont présents dans la modélisation du problème. La modélisation d'un GCSOP se compose de :

- Une liste de *VarGraph*s $G_1, G_2, ..., G_n$. Chaque G_i représente un objet graphe dont la valeur est dans

l'interval $[glb_i, lub_i]$,
- Une liste de *GraphObjective*s et une liste de *GraphConstraint*s qui sont définis sur $G_1, G_2, ..., G_n$,
- Un système de contraintes *GraphConstraintSystem* sur graphes, ou un combinateur *GraphObjective*s qui collecte et intègre toutes les contraintes et les *GraphObjective*s dans un objet global. Cet objet sera utilisé dans le composant de recherche.

2.2 La recherche

Le voisinage et les mouvements locaux Le voisinage d'une solution est l'ensemble de solutions voisines de la solution courante qui sont générées par un changement local sur cette solution. Un mouvement local est un changement local sur la solution courante pour générer une nouvelle solution. Normalement, un changement local sur un graphe dynamique consiste en une insertion ou une suppression d'un ou de quelques noeuds (arêtes), ou un remplacement de quelques noeuds (arêtes) par d'autres noeuds (arêtes). Dans ce papier, nous considérons les movements locaux suivants :

- *addNode(g,u)* : insérer un noeud u dans le graphe g.
- *removeNode(g,u)* : supprimer le noeud u du graphe g ; cette action supprime implicitement toutes les arêtes adjacentes au noeud u.
- *addEdge(g,e)* : insérer une arête e au graphe g, cette action insère implicitement les deux extrémités de e si celles-ci n'existent pas.
- *removeEdge(g,e)* : supprimer l'arête e de g.
- *removeCompleteEdge(g,e)* : supprimer l'arête e de g et ses extrémités si celles-ci sont isolées après la suppression de cette arête.
- *replaceEdge(g,oe,ne)* : remplacer l'arête oe de g par une autre arête ne. Cette action est en fait constituée par une séquence de deux actions consécutives : <*removeEdge(g,oe)*, *addEdge(g,ne)*>.
- *replaceCompleteEdge(g,oe,ne)* : remplacer l'arête oe de g par une autre arête ne. Cette action est en fait constituée par une séquence de deux actions consécutives : <*removeCompleteEdge(g,oe)*, *addEdge(g,ne)*>.

Nous pouvons également combiner les actions ci-dessus pour constituer des mouvements plus complexes. Cependant, de tels mouvements locaux complexes sont plus coûteux.

La procédure de recherche Dès qu'une modélisation est disponible, nous pouvons appliquer différentes stratégies heuristiques et metaheuristiques [9] pour explorer l'espace de solutions potentielles. Grâce à l'architecture du cadre proposé, la procédure de recherche est indépendante de la modélisation. Il est donc facile d'ajouter de nouvelles contraintes sans avoir besoin de modifier le composant de recherche, ainsi qu'explorer diverses techniques heuristiques et metaheuristiques sans avoir besoin de changer la modélisation.

```
class VarGraph{
  1. void          addNode(Node v);
  2. void          removeNode(Node v);
  3. void          addEdge(Edge e);
  4. void          removeEdge(Edge e);
  5. void          replaceEdge(Edge oe, Edge ne);
  6. void          assign(Graph cg);
  7. void          assign(VarGraph g);
  ...
  //** queries
  8. set{Node}     getNodes();
  9. set{Edge}     getEdges();
  10. set{Node}    getOptionalNodes();
  11. set{Edge}    getOptionalEdges();
  ...
}
```

FIG. 2 – Classe `VarGraph` (description partielle)

3 Apperçu de `LS(Graph)`

3.1 Classes du noyau

Les classes de noyau qui représentent des objets reliés aux graphes sont `Node`, `Edge`. La classe `Graph` décrit un graphe non-dirigé fixé. La Figure 2 est une description partielle de la classe `VarGraph` qui représente un graphe non-dirigé dynamique. Les lignes 1-7 déclarent les actions de mise-à-jour sur des graphes dynamiques qui sont mentionnées ci-dessus. Comme les variables incrémentales de `COMET`, les objets de type `VarGraph` sont utilisés pour définir des objets différentiables. Une mise-à-jour sur un objet (`VarGraph`) induit une propagation qui met à jour tous les objets différentiables qui sont définis sur cet objet. Les lignes 8-11 questionnent quelques propriétés de graphes. Par exemple, les lignes 8-9 retournent l'ensemble de noeuds et l'ensemble d'arêtes du graphe courant ; les lignes 10-11 retournent l'ensemble de noeuds optionnels et l'ensemble d'arêtes optionnelles du graphe courant.

3.2 Interface `GraphConstraintInterface`

Il existe plusieurs contraintes sur des graphes. Afin d'assurer la compositionalité et la réutilisabilité, toutes les *GraphConstraint*s implémentent la même interface `GraphConstraintInterface` (voir la Figure 3). De plus, ceci permet de concevoir des combinateurs des contraintes sur des graphes (e.g. *GraphConstraintSystem*).

La méthode `getVarGraphs` (ligne 1) permet d'accéder à la liste de `VarGraph`s intervenant dans la *GraphConstraint*. Le nombre de violations et la valeur de

```
interface GraphConstraintInterface{
 1. VarGraph[]      getVarGraphs();
 2. var{int}        violations();
 3. var{boolean}    isTrue();
 4. int getAddNodeDelta(VarGraph g, Node v);
 5. int getRemoveNodeDelta(VarGraph g, Node v);
 6. int getAddEdgeDelta(VarGraph g, Edge e);
 7. int getRemoveEdgeDelta(VarGraph g, Edge e);
 8. int getReplaceEdgeDelta(VarGraph g, Edge oe, Edge ne);
 ...
}
```

FIG. 3 – Interface `GraphConstraintInterface` (description partielle)

```
class GraphConstraintSystem implements
GraphConstraintInterface{

 //** ...GraphConstraintInterface...
 1. GraphConstraintInterface     getGraphConstraint(int i);
 2. GraphConstraintInterface[]   getGraphConstraints();
 3. void post(GraphConstraintInterface gc, int w);
 ...
}
```

FIG. 4 – Classe `GraphConstraintSystem` (description partielle)

vérité de *GraphConstraint* sont retournées par les méthodes `violations()` (ligne 2) et `isTrue()` (ligne 3). Les méthodes restant (lignes 4-8) fournissent l'API différentiable de *GraphConstraint*. Ceci permet de questionner la variation du nombre de violations pour divers mouvements locaux.

3.3 Classe `GraphConstraintSystem`

Tout comme le `ConstraintSystem` de `COMET`, le *GraphConstraintSystem* est un combinateur de contraintes qui collecte toutes les *GraphConstraint*s du modèle et fournit une séparation propre entre le modèle et le composant de recherche. Il est donc possible d'ajouter de nouvelles *GraphConstraint*s au *GraphConstraintSystem* sans devoir changer le composant de recherche. Nous pouvons expérimenter différentes heuristiques et metaheuristiques dans la procédure de recherche locale, telle qu'ajouter un composant de tabou ou ajouter un composant d'intensification sur le modèle spécifié.

La Figure 4 est une description partielle de la classe `GraphConstraintSystem` qui implémente l'interface `GraphConstraintInterface`. Les méthodes aux lignes 1-2 accèdent aux *GraphConstraint*s postées dans le *GraphConstraintSystem*. La méthode `post()` (ligne 3) enregistre un *GraphConstraint* `gc` avec le poids `w` dans le *GraphConstraintSystem*.

```
void stateModel(){
1.   LocalSolver ls();
2.   VarGraph t(ls,glb,lub);
3.   GraphConstraintSystem gcs = new GraphConstraintSystem(ls);
4.   NBNodes nbNodes = new NBNodes(t);
5.   NBEdges nbEdges = new NBEdges(t);

6.   Connected connected = new Connected(t);
7.   gcs.post(connected,1);

8.   GEq eq = new GEq(nbNodes,n);
9.   gcs.post(eq,1);

10.  eq = new GEq(nbEdges,n-1);
11.  gcs.post(eq,1);

12.  BoundedDegree bd = new BoundedDegree(t,maxD);
13.  gcs.post(bd,1);

14.  WeightGraph cost = new WeightGraph(t);
15.  GraphObjectiveInterface go;
16.  go = new GOAdd(cost, new GOMul(gcs,100));
17.  ls.close();
}
```

FIG. 5 – La modélisation de DCMST

3.4 Exemple : Problème du minimum spanning tree with degree constraint (DCMST)

Nous illustrons `LS(Graph)` sur le problème du minimum spanning tree with degree constraint (Degree Constrained Minimum Spanning Tree problem, DCMST). DCMST est défini comme suit : étant donné un graphe non-dirigé pondéré g où chaque noeud v de g a une borne supérieure d_v sur son degré. Le problème DCMST [14][2] consiste à trouver un spanning tree de coût minimum t de g tel que $deg(v) \leq d_v, \forall v \in V(t)$. Puisque DCMST est un problème d'optimisation sous contraintes, nous combinons dans cet exemple la faisabilité et l'optimalité dans une fonction objectif unifiée. La modélisation du problème est présentée dans la Figure 5.

La ligne 3 initialise un *GraphConstraintSystem* `gcs`. Deux *GraphObjective*s, `nbNodes` et `nbEdges` représentent respectivement le nombre de noeuds et le nombre d'arêtes du graphe courant `t` et initialisés en lignes 4-5. La contrainte *Tree* qui spécifie qu'un graphe non-dirigé t est un spanning tree du graphe donné lub (qui est la borne supérieure de t) est décomposée en trois *GraphConstraint*s :

1. t est connexe
2. Le nombre de noeuds de t est égal au nombre de noeuds de lub :

$$\sharp(V(t)) = \sharp(V(lub))$$

3. Le nombre d'arêtes de t est égal au nombre de noeuds - 1 :

$$\sharp(E(t)) = \sharp(V(t)) - 1$$

Les trois contraintes ci-dessus qui spécifient que le graphe `t` est un spanning tree sont crées et postées aux lignes 6-11. Les lignes 6-7 initialisent et postent une contrainte `Connected` spécifiant que le graphe `t` est connexe. Les lignes 8-9 initialisent et postent une contrainte spécifiant que le nombre de noeuds de `t` est égal à n. Les lignes 10-11 initialisent et postent une contrainte qui spécifie que le nombre d'arêtes de `t` est égal à $n-1$. Les lignes 12-13 initialisent et postent une contrainte `BoundedDegree` sur les degrés des noeuds de `t`. L'objet `cost` est un *GraphObjective* qui est créée à la ligne 14. Les lignes 15-16 initialisent un combinateur `go` des *GraphObjective* qui combinent le nombre de violations des *GraphConstraint*s avec un poids de 100 afin de privilégier la recherche vers des solutions correctes (solutions qui satisfont toutes les contraintes données) et le coût de la fonction objectif avec un poids de 1.

3.5 VarTree

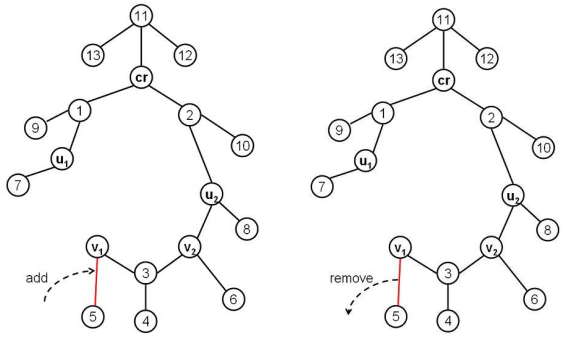
Il existe de nombreux problèmes où l'on souhaite découvrir des arbres optimaux satisfaisant des contraintes tels que :

- Degree Constrained Minimum Spanning Tree (DCMST) [14] [2] [3] [22] [21] [24]
- Bounded Diameter Minimum Spanning Tree (BDMST) [8][20][11]
- Capacitated Minimum Spanning Tree Problem (CMST) [19][1]
- Minimum Diameter Spanning Tree (MDST) [17]
- Minimum k-Cardinality Tree (KCT) [4]
- Steiner Minimal Tree (SMT) [23]

Ces problèmes peuvent être modélisés en utilisant *VarGraph* pour la recherche locale, mais nous devons dans ce cas poster une contrainte *Tree(G)* qui spécifie que le graphe considéré G est un arbre. Parfois, la contrainte *Tree* peut être considérée comme une contrainte forte et peut être maintenue pendant la recherche locale. Ceci exige des mouvements spécifiques plutôt que les mouvements généraux de *VarGraph*. En utilisant *VarGraph* dans la modélisation, l'utilisateur doit manipuler une structure de données supplémetaire assez compliquée pour calculer des voisins qui sont des arbres. *VarTree* est une abstraction qui représente un arbre dynamique. Le *VarTree* fournit un mécanisme permettant d'explorer facilement des voisins qui sont toujours des arbres.

Mouvements généraux pour VarTree Etant donné un arbre dynamique t, on définit l'ensemble d'actions de mise-à-jour (insertion, suppression, remplacement des arêtes) sur t conservant la propriété d'arbre :

1. Une arête $e = (u, v)$ peut être ajoutée à t s'il existe exactement un noeud u ou v de t : $u \in V(t)$ `XOR` $v \in V(t)$ (voir la Figure 6a). Cette arête est appellé arête *insérable*. L'insertion d'une arête ajoute implicitement ses extrémités à t si elles n'existent pas dans t.
2. Une arête $e = (u, v)$ peut être supprimée de t si un noeud u ou v est une feuille de t : $deg(u) = 1 \vee deg(v) = 1$ (voir la Figure 6b). Cette arête est appellé arête *supprimable*. La suppression d'une arête enlève aussi ses extrémités si celles-ci sont des feuilles de t.
3. Une arête e_1 de t peut être remplacée par une autre arête e_2 si e_2 réconnecte deux sous-arbres générés par la suppression de e_1 de t (voir la Figure 7). e_2 est appellée arête *remplaçante* et e_1 est appellée arête *remplaçable* de e_2. L'ensemble de noeuds de t n'est pas changé par ce remplacement.



a. insérer une arête b. supprimer une arête

FIG. 6 – actions de mise-à-jour : insertion et suppression d'une arête

Le code décrit Figure 8 permet d'accéder aux ensembles d'arêtes qui sont utilisés pour des mouvements sur un arbre dynamique `tree`. Par exemple, `e` est une arête *remplaçante*, la ligne 5 retourne l'ensemble d'arêtes *remplaçable*s de `e`. Ces ensembles sont maintenus de façon incrémentale grâce aux structures de données spécifiques.

4 Problème du Edge-weighted k-Cardinality Tree (KCT)

4.1 Problème du KCT

Etant donné un graphe non-dirigé pondéré $G = (V, E)$ et une valeur k ($1 \leq k \leq |V| - 1$), le problème KTC consiste à trouver un sous-arbre de g ayant exactement k arêtes et dont la somme des poids de toutes les arêtes est minimale. Le problème KCT a intéressé plusieurs chercheurs. Récemment, [4] a proposé trois

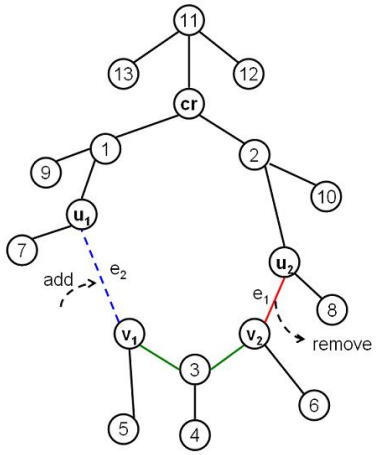


FIG. 7 – action de mise-à-jour : remplacement d'une arête

```
1. VarTree tree(ls,glb,lub) ;
2. set{Edge} removableEdges = tree.getRemovableEdges() ;
3. set{Edge} insertableEdges = tree.getInsertableEdges() ;
4. set{Edge} replacingEdges = tree.getReplacingEdges() ;
5. set{Edge} replacableEdges = tree.getReplacableEdges(e) ;
```

FIG. 8 – Ensembles d'arêtes utilisés pour les mouvements de *VarTree*

approches metaheuristiques pour résoudre KCT. Un algorithme de recherche tabou exploite un voisinage simple obtenu par application simultanée des deux premiers types d'action de mise-à-jour décrites dans la section 3.5. Le troisième type d'action de mise-à-jour n'est cependant pas exploité. Cet algorithme de recherche tabou fonctionne très bien sur un ensemble de graphes 4-regular publiés comme benchmarks dans KCTLib [10]. Une implémentation en C++ de cette recherche tabou (dénotée TS_KCT) est aussi disponible dans KCTLib.

4.2 Modèle LS(Graph) de KCT

Nous avons implémenté un algorithme de recherche tabou en utilisant l'abstraction *VarTree* du cadre `LS(Graph)` (dénoté par MTS_KCT_VT) et exploitant le même schéma que TS_KCT. Notre recherche tabou explore un voisinage plus large en incluant le troisième type d'action de mise-à-jour sur `VarTree` (voir la section 3.5). La solution désirée est un sous-graphe de G qui :

1. *est un arbre*,
2. *a k arêtes*,

```
void stateModel{
     1. LocalSolver ls() ;
     2. VarTree tree(ls,glb,lub) ;
     3. WeightGraph fo(tree) ;
     4. ls.close() ;
}
```

FIG. 9 – Modélisation du problème KCT

```
void exploreNeighborhood(MinNeighborSelector NS){
1.   exploreGreedyNeighborhood1(NS) ;
2.   exploreGreedyNeighborhood2(NS) ;
3.   if(NS.getIntMin() + fCur >= frb){
4.        N.getMove() ;
5.        exploreNeighborhoodWithTabu(NS) ;
6.   }
}
```

FIG. 10 – Exploration de voisinage (partie I)

3. *et dont la somme des poids de toutes les arêtes est minimale.*

La première contrainte est toujours respectée par l'abstraction *VarTree*. La deuxième contrainte est assurée par la construction de la solution initiale (un arbre de k arêtes) et par le voisinage choisi qui est constitué par le *remplacement* d'une arête de l'arbre courant par une autre arête satisfaisant certains critères. La fonction objectif de la troisième condition est représenté par un *GraphObjective* appellé *WeightGraph*. La modélisation du problème est montrée dans le code de la Figure 9 où `glb` est nul et `lub` est le graphe d'entrée G. La recherche tabou contient aussi un critère d'aspiration qui accepte des solutions tabou mais qui améliorent la restart-best solution (la meilleure solution découverte depuis le redémarrage de la recherche). Le code de la Figure 10 décrit l'exploration du voisinage. `MinNeighborSelector NS` est une abstraction de contrôle de `COMET` qui stocke le meilleur mouvement soumis jusqu'a présent ainsi que son évaluation. Le voisinage est d'abord exploré de façon gloutonne (méthodes en lignes 1-2 qui sont détaillées dans les Figures 11, 12) ignorant les listes tabou. Si la solution sélectionnée n'améliore pas la restart-best solution (ligne 3), le mouvement choisi est supprimé de `NS` (ligne 4) et l'exploration utilisant les listes tabou est appliquée (méthode en ligne 5 qui est détaillée dans la Figure 13).

La Figure 11 explore le voisinage utilisé par [10]. La méthode scanne tous les couples <`ei,eo`> (`ei` appartient à l'ensemble des arêtes `insertables` (ligne 8), `eo` appartient à l'ensemble des arêtes `supprimables` (ligne 4)). Le remplacement de `eo` par `ei` représente un mouvement local. La ligne 9 retourne la variation de la fonction objectif lors d'un mouvement local. Le

```
void exploreGreedyNeighborhood1(NeighborSelector NS){
1.   int eval = System.getMAXINT();
2.   Edge edgeOut = null;
3.   Edge edgeIn = null;
4.   forall(eo in tree.getRemovableEdges()){
5.       Node u = eo.getBeginPoint();
6.       if(tree.getAdjNodes(u).getSize() > 1)
7.           u = eo.getEndPoint();
8.       forall(ei in tree.getInsertableEdges() :
                 !ei.contain(u)){
9.           int d = go.getReplaceEdgeDelta(tree,eo,ei);
10.          if(eval > d){
11.              eval = d;
12.              edgeOut = eo;
13.              edgeIn = ei;
14.          }
15.      }
16.  }
17.  if(edgeOut != null && edgeIn != null){
18.      neighbor(eval,NS){
19.          tree.removeEdge(edgeOut);
20.          tree.addEdge(edgeIn);
21.          tbIn.makeTabu(edgeOut,it);
22.          tbOut.makeTabu(edgeIn,it);
23.      }
24.  }
}
```

FIG. 11 – Exploration de voisinage (partie II)

```
void exploreGreedyNeighborhood2(NeighborSelector NS){
1.   selectMin(ei in tree.getReplacingEdges(),
               eo in tree.getReplacableEdges(ei),
               d = go.getReplaceEdgeDelta(tree,eo,ei))(d){
2.       neighbor(d,NS){
3.           tree.replaceEdge(eo,ei);
4.           tbIn.makeTabu(eo,it);
5.           tbOut.makeTabu(ei,it);
6.       }
7.   }
}
```

FIG. 12 – Exploration de voisinage (partie III)

cas où `ei` et `eo` ont une extrémité commune qui est une feuille de l'arbre courant est évité (lignes 5-8) car un tel mouvement local déstruirait la structure d'arbre. Le meilleur mouvement et son évaluation `eval` sont stockés dans `NS` (lignes 18-22). Le mouvement sélectionné rend tabou les deux arêtes choisies `edgeIn` et `edgeOut` (lignes 21-22, où `it` est l'itération courante de la recherche locale).

L'exploration du voisinage supplémentaire est illustrée à la Figure 12. Tous les couples d'une arête *replaçant* `ei` et d'une arête *replaçable* `eo` de `ei` sont scannés par l'abstraction de contrôle `selectMin` de `COMET`.

Le code de la Figure 13 explore le voisinage utilisé par [10] et utilise les listes tabou. L'instruction qui vérifie l'appartenance à la liste tabou y est insérée (ligne 9). Lorsqu'une solution voisine qui améliore la solution courante est découverte, l'exploration s'arrête.

```
void exploreNeighborhoodWithTabu(NeighborSelector NS){
1.   int eval = System.getMAXINT();
2.   Edge edgeOut = null;
3.   Edge edgeIn = null;
4.   forall(eo in tree.getRemovableEdges()){
5.       Node u = eo.getBeginPoint();
6.       if(tree.getAdjNodes(u).getSize() > 1)
7.           u = eo.getEndPoint();
8.       forall(ei in tree.getInsertableEdges() :
                 !ei.contain(u)){
9.           if(!tbOut.isTabu(eo,it) || !tbIn.isTabu(ei,it)){
10.              int d = go.getReplaceEdgeDelta(tree,eo,ei);
11.              if(eval > d){
12.                  eval = d;
13.                  edgeOut = eo;
14.                  edgeIn = ei;
15.              }
16.              if(eval < 0){
17.                      break;
18.              }
19.          }
20       }
21.      if(eval < 0){
22.          break;
23.      }
24.  }
25.  if(edgeOut != null && edgeIn != null){
26.      neighbor(eval,NS){
27.          tree.removeEdge(edgeOut);
28.          tree.addEdge(edgeIn);
29.          tbIn.makeTabu(edgeOut,it);
30.          tbOut.makeTabu(edgeIn,it);
31.      }
32.  }
}
```

FIG. 13 – Exploration de voisinage (partie IV)

4.3 Résultats expérimentaux

Nous avons expérimenté le MTS_KCT_VT sur un nouvel ensemble de données formé de graphes euclidiens complets générés comme suit : on génère aléatoirement n noeuds avec des coordonnés (x, y) où x et y sont générés par une distribution uniforme dans l'intervalle [1..500]. Le poids d'une arête est la distance euclidienne entre ses extrémités. Cinq graphes de taille 200 (avec $k = 150$) et cinq graphes de taille 500 (avec $k = 166$) ont été générés.

Nous comparons notre implémentation MTS_KCT_VT avec TS_KCT sur cet ensemble de données. Chaque programme a été exécuté 5 fois pour chaque graphe. Le temps est limité à 10 minutes. Les résultats expérimentaux sont présentés dans la Table 1. Pour chaque instance, les colonnes 2-4 reportent la valeur minimale, maximale et moyenne de la fonction objectif de la meilleure solution trouvée (de 5 exécutions) par MTS_KCT_VT et les colonnes 5-6 présentent la valeur moyenne (de 5 exécutions) de la première itération et le moment (en sec.) où cette solution est découverte. Les mêmes informations pour TS_KCT sont présentées en colonnes 7-11.

Les résultats expérimentaux montrent que MTS_KCT_VT trouve une meilleure solution que

Graph	MTS_KCT_VT					TS_KCT[10]				
	min	Max	avg	avgIt	avgT	min	Max	avg	avgIt	avgT
gEcl200.in1	**1259***	1288	1274.8	39	117.48	1261	1273	1268	7850	489.11
gEcl200.in2	**1185***	1197	1190	50	148.3	1191	1260	1221.4	6231.2	412.64
gEcl200.in3	**1243***	1267	1262.2	55.2	164.34	1250	1284	1267.4	4659.2	325.38
gEcl200.in4	**1216***	1218	1216.6	46.8	140.67	1231	1259	1245	2636.2	175.95
gEcl200.in5	**1314***	1341	1323.8	70.4	231.39	1332	1400	1360.2	4257.6	272.25
gEcl500.in1	**2109***	2220	2167.2	48	559.64	2198	2427	2307.8	269.2	349.07
gEcl500.in2	**2176***	2272	2234.6	36.6	462.87	2328	2633	2463.4	250.6	314.69
gEcl500.in3	**2078***	2215	2162.8	34.2	508.11	2095	2498	2338	229.6	289.26
gEcl500.in4	**2129***	2282	2197.8	32	415.04	2153	2745	2362.6	253.8	332.15
gEcl500.in5	**2010***	2377	2158.8	33.4	416.81	2057	2447	2196.2	207.4	243.18

TAB. 1 – Comparaison entre MTS_KCT_VT et l'implementation de la recherche tabou en C++ de [10]

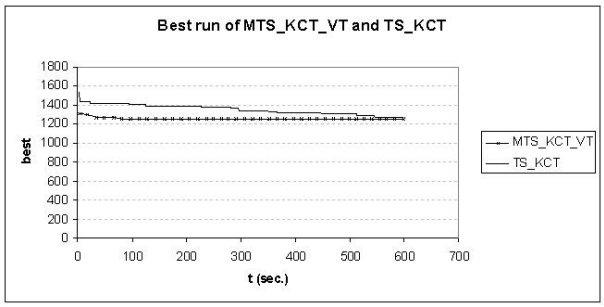


FIG. 14 – Comportements de MTS_KCT_VT et TS_KCT sur l'instance graphEucliden200e19900.in1

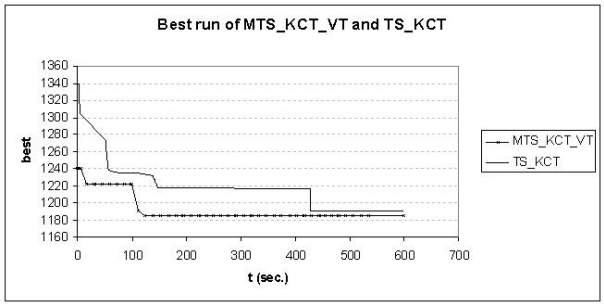


FIG. 15 – Comportements MTS_KCT_VT et TS_KCT sur l'instance graphEucliden200e19900.in2

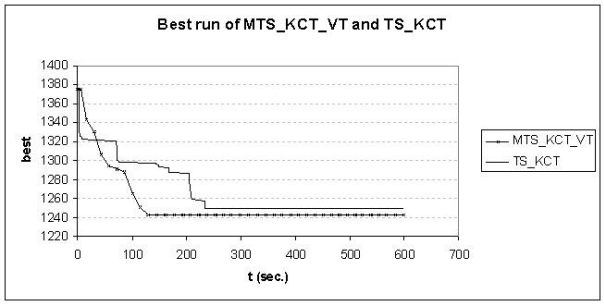


FIG. 16 – Comportements de MTS_KCT_VT et TS_KCT sur l'instance graphEucliden200e19900.in3

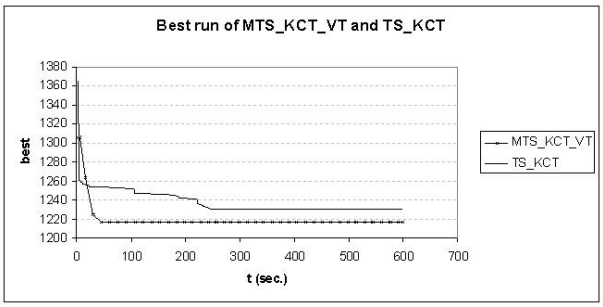


FIG. 17 – Comportements de MTS_KCT_VT et TS_KCT sur l'instance graphEucliden200e19900.in4

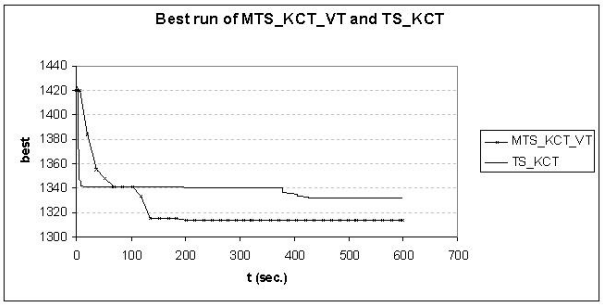


FIG. 18 – Comportements de MTS_KCT_VT et TS_KCT sur l'instance graphEucliden200e19900.in5

TS_KCT grâce à l'exploration d'un voisinage plus large. De plus, la valeur moyenne de la fonction objectif des meilleures solutions et la valeur moyenne du nombre d'itérations pour atteindre ces solutions avec MTS_KCT_VT sont plus petites que celle avec TS_KCT. Avec les instances de taille 200 (cinq premières rangées dans la Table 1), le temps moyen pour obtenir les meilleures solutions avec MTS_KCT_VT est plus petit que celui avec TS_KCT. Cependant, la taille large du voisinage a pour conséquence une exploration de voisinage plus lente.

Les Figures 14, 15, 16, 17, 18 comparent le comportement des meilleures exécutions de MTS_KCT_VT et TS_KCT sur quelques instances. Nous voyons que le MTS_KCT_VT trouve rapidement une meilleure solution que TS_KCT (la valeur de la fonction objectif de la solution trouvée par MTS_KCT_VT est plus petite que celle de la solution trouvée par TS_KCT).

5 Conclusion

Cette recherche propose un nouveau modèle de calcul pour simplifier la modélisation de recherche locale sur une classe de problèmes combinatoires : les problèmes d'optimisation sous contraintes sur des graphes. Le modèle de calcul a de nombreux avantages. Au niveau de la programmation, les algorithmes de recherche locale sont courts et concis. Au niveau de la performance, le modèle s'intègre bien avec le Constraint-Based Local Search de `COMET`. Les composants built-in sont implémentés efficacement avec des structures de données et des algorithmes (incrémentaux) spécifiques sur des graphes. Les utilisateurs peuvent implémenter efficacement des algorithmes de recherche locale sans avoir besoin de manipuler des structures de données complexes et des algorithmes sophistiqués sur des graphes. Au niveau du langage de programmation, le modèle de calcul se caractérise par sa compositionalité et sa réutilisabilité. Il est facile d'ajouter de nouvelles contraintes au modèle sans devoir modifier le composant de recherche et on peut aussi explorer diverses stratégies de recherche heuristiques et metaheuristiques. Le temps de développement est fortement réduit, permettant ainsi d'expérimenter rapidement plusieurs algorithmes de recherche locale. Cette caractéristique est importante lorsqu'il n'y a pas d'algorithmes de recherche locale qui fonctionne bien sur tous les types de données. Le noyau du modèle de calcul est le concept de *VarGraph* qui représente un graphe dynamique. D'autre composants peuvent être définis sur les *VarGraph*s, tels que *GraphInvariant*s, *GraphConstraint*s et *GraphObjective*s. Tout comme les variables incrémentales de `COMET`, une mise-à-jour sur *VarGraph* induit une propagation qui met à jour les *GraphInvariant*s, les *GraphConstraint*s et les *GraphObjective*s définis sur cette *VarGraph*. *VarTree* est une abstraction représentant un arbre dynamique. Cette abstraction permet de modéliser des problèmes de découverte d'arbres sous contraintes. Le cadre `LS(Graph)` a été appliqué pour modéliser et résoudre le problème de KCT en utilisant l'abstraction *VarTree*. En utilisant une exploration sur un voisinage plus large, nous avons montré des résultats compétitifs par rapport à une implémentation en C++ d'une recherche tabou décrite par [10], sur un ensemble de graphes euclidiens complets générés aléatoirement. Dans le futur, notre travail se focalisera sur la conception et l'implémentation d'un *GraphObjective* appellé *Diameter* représentant le diamètre d'un arbre dynamique, ainsi que son application dans la résolution le problème de Bounded Diameter Minimum Spanning Tree Problem (BDMST). Un objectif à plus long terme de notre recherche consiste à développer un cadre unifié permettant d'exploiter des techniques d'hybridation : la programmation par constraintes et la recherche locale où la programmation par constraintes réduit la borne $[glb, lub]$ par des techniques de filtrage [6].

Remerciements Nous remercions Christian Blum et Maria José Blesa Aguilera pour nous avoir fourrni un C++ software de la résolution du problème de KCT. Une partie de cette recherche est supportée par la Région Wallonne (projet Transmaze, WIST516207), ainsi que par le PAI Moves (Politique scientifique belge).

Références

[1] Ravindra K. Ahujaa, James B. Orlinb, and Dushyant Sharma. A composite very large-scale neighborhood structure for the capacitated minimum spanning tree problem. *Operations Research Letters 31*, pages 185–194, 2003.

[2] Rafael Andrade, Abilio Lucena, and Nelson Maculan. Using lagrangian dual information to generate degree constrained spanning trees. *Discrete Applied Mathematics*, pages 703–717, 2006.

[3] Y.T. Bau1, C.K. Ho, and H.T. Ewe. An ant colony optimization approach to the degree-constrained minimum spanning tree problem. *Computational Intelligence and Security, Volume 3801/2005*, pages 657–662, 2006.

[4] C. Blum and M. Blesa. New metaheuristic approaches for the edge-weighted k-cardinality tree problem. *Computers and Operations Research*, pages 32(6) :1355–1377, 2005.

[5] Abraham Bookstein and Shmuel T. Klein. Compression of correlated bit-vectors. *Information Systems*, 16(4) :387–400, 1991.

[6] Grégoire Dooms. Phd thesis. *UCL*, 2006.

[7] M.N. Gomes, R.C. Andrade, C.P. Santiago, and N. Maculan. Spanning tree algorithms to some hard combinatorial problems. *in : Proceedings of Optimization Days, Montreal/Canada*, pages 83–84, 1997.

[8] M. Gruber, J.I. van Hemert, and G.R. Raidl. Neighborhood searches for the bounded diameter minimum spanning tree problem embedded in a vns, ea, and aco. *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1187–1194, 2006.

[9] Pascal Van Hentenrych and Laurent Michel. *Constraint-based local search.* The MIT Press, London, England, 2005.

[10] http ://iridia.ulb.ac.be/ cblum/kctlib/.

[11] B. A. Julstrom. Greedy heuristics for the bounded-diameter minimum spanning tree problem. *Technical report, St. Cloud State University, Submitted for publication in the ACM Journal of Experimental Algorithmics*, 2004.

[12] J. Knowles and D. Corne. A new evolutionary approach to the degree constrained minimum spanning tree problem. *IEEE Trans. Evolutionary Comput. 4(2)*, pages 125–134, 2000.

[13] Mohan Krishnamoorthy and Yazid M. Sharaiha Andreas T. Ernst. Comparison of algorithms for the degree constrained minimum spanning tree. *Journal of Heuristics, v.7 n.6*, pages 587–611, November 2001.

[14] Mohan Krishnamoorthy, Andreas T. Ernst, and Yazid M. Sharaiha. Comparison of algorithms for the degree constrained minimum spanning tree. *Journal of Heuristics*, pages 587–611, 2001.

[15] Foulds LR, Hamacher HW, and Wilson J. Integer programming approaches to facilities layout models with forbidden areas. *Annals of Operations Research*, 81 :405–417, 1998.

[16] A.P. Lucane and J.E. Beasley. Advances in linear and integer programming. *Oxford Lecture Series in Mathematics and its Applications, Oxford University Press*, 1996.

[17] Enrico Nardelli and Guido Proietti. Finding all the best swaps of a minimum diameter spanning tree under transient edge failures. *Journal of Graph Algorithms and Applications vol. 5, no. 5*, pages 39–57, 2001.

[18] Kerry Raymond. A tree-based algorithm for distributed mutual exclusion. *ACM Trans. Comput. Syst.*, 7(1) :61–77, 1989.

[19] Marc Reimann and Marco Laumanns. A hybrid aco algorithm for the capacitated minimum spanning tree problem. *Proceedings of First International Workshop on Hybrid Metaheuristics*, pages 1–10, 2004.

[20] Alok Singh and Ashok K. Gupta. Improved heuristics for the bounded-diameter minimum spanning tree problem. *Soft Computing - A Fusion of Foundations, Methodologies and Applications*, pages 911–921, 2006.

[21] M. C. Souza and C. C. Ribeiro. Variable neighborhood search for the degree-constrained minimum spanning tree problem. *Discrete Appl. Math. 118*, pages 43–54, 2002.

[22] A. Volgenant. A lagrangian approach to the dcmst problem. *European J. Oper. Res. 39*, pages 325–331, 1989.

[23] M. Zachariasen. Local Search for the Steiner Tree Problem in the Euclidean Plane. *European Journal of Operational Research*, 119 :282–300, 1999.

[24] G. Zhou and M. Gen. A note on genetic algorithms for degree constrained spanning tree problems. *Network 30*, pages 91–95, 1997.