Algorithme Li et al. : ensemble dominant connexe glouton

Alexandre Fernandez & Sylvain Ung

22 octobre 2017

Rapport de Recherche

Table des matières

Algorithme Li et al. : ensemble dominant connexe glouton

Résumé : Il va s'agir dans ce rapport de recherche d'étudier une implémentation efficace d'un algorithme répondant au problème de l'ensemble dominant connexe qui est très largement utilisé pour modéliser un réseau sans-fil. L'algorithme étudié est celui de Li et al [3] qui propose une approche gloutonne de la solution. L'objectif étant de minimiser cet ensemble afin de réduire, en pratique, les coûts de maintenance par exemple.

Mots-clés : Théorie des graphes, ensemble dominant, ensemble stable maximum, connexe, NP-complet, algorithme glouton, Li et al.

1 Introduction

Étant donné un graphe $G = (V, E)$ avec V (respectivement E) l'ensemble des sommets (respectivement l'ensemble des arêtes), l'ensemble dominant connexe [1] de G est le sous-ensemble $D \subseteq V$ tel que $\forall v \in V, v \in D \vee v$ est un voisin d'un élément de D et tel que $G[D]$, le sous-graphe induit par D est connexe.

Cette définition nous indique que le CDS n'est pas unique pour un graphe donné (sauf cas dégénérés) et qu'il en existe plusieurs de tailles différentes. Cela nous amène au problème que nous allons traiter : la recherche du plus petit CDS. C'est un problème NP-difficile ce qui signifie que nous ne pourrons pas calculer le plus petit, autrement dit la meilleure solution, en un temps raisonnable. C'est pour cela que nous allons présenter dans ce rapport un algorithme donnant une solution qui s'en rapproche.

Ainsi, l'algorithme Li et al. propose une approche gloutonne à la résolution de ce problème dont le principe de résolution se fait étape par étape. En effet, à chaque étape un optimum local est choisi afin d'arriver à un optimum global à la fin. Cet algorithme peut être décomposé en 2 grandes étapes :

1. Construire l'ensemble stable maximal [2]
2. Connecter les points de l'ensemble précédemment construit entre eux par des points qui n'y appartiennent pas, ceux-ci sont appelés *nœuds de Steiner*

Dans notre démarche, nous présenterons tout d'abord les idées théoriques qui se trouvent derrière l'algorithme puis nous détaillerons l'implémentation que nous en avons fait, éventuellement des choix et des modifications pris par rapport l'algorithme original. Ensuite, nous présenterons les conditions et résultats des tests obtenus suivi d'une discussion sur la pertinence d'un tel algorithme.

Enfin, nous conclurons sur les enjeux et problématiques de l'ensemble dominant connexe dans les graphes géométriques, également appelés graphes de disques.

1. CDS (Connected Dominating Set)
2. MIS (Maximum Independent Set), nous en donnerons la définition dans la section 2.1

2 Algorithme

2.1 Construction de l'ensemble stable maximal

La première étape de l'algorithme consiste en la construction du MIS dont la définition est la suivante et qui possède des propriétés intéressantes pour la suite :

Définition 2.1.1 *Soit $G = (V, E)$ un graphe, l'ensemble stable maximal est le plus grand sous-ensemble $S \subseteq V$ tel que le sous-graphe $G[S]$ induit par S ne contient pas d'arêtes.*

Lemme 2.1.1 *Dans tout graphe géométrique, la taille de ses ensembles stables maximaux est majorée par $3.8opt + 1.2$ où opt est la taille de l'ensemble connexe dominant minimum.*

Lemme 2.1.2 *Toute paire de sous-ensembles complémentaires du MIS a exactement une distance de deux sauts.*

Lemme 2.1.1 garantit que le ratio de la solution de cet algorithme par rapport à la solution optimale est bien $4.8 + ln5$. La validité de ce lemme dépend beaucoup de nos choix d'implémentation pour les algorithmes de construction MIS. Ces algorithmes étant prévus pour fonctionner de façon distribuée, nos implémentations, non distribuées, ne se comporte donc pas exactement comme ceux-ci.

Lemme 2.1.2 est d'une importance majeure pour la validité de la solution. En effet, si une paire de sous-ensembles complémentaires du MIS a une distance inférieure à deux sauts, le MIS sera plus grand que prévu. On risque donc dans ce cas d'obtenir un résultat final plus grand. À l'inverse, si la distance entre deux sous ensembles complémentaires du MIS est de plus de deux sauts. L'algorithme, essayant de relier ces sous-ensembles du MIS en colorant des nœuds contenant au minimum un voisin dans chacun de ces sous ensembles, ne pourra pas relier ces deux sous-ensembles. L'ensemble obtenu sera constitué d'au moins 2 composantes connexes correspondant aux deux ensembles complémentaires du MIS ayant une distance de plus de deux sauts.

Afin de construire un tel MIS qui répond correctement à ces propriété nous nous sommes appuyés une des références [1] cités par les auteurs de l'article. Comme précédemment dit, nous avons dû adapter l'algorithme pour un fonctionnement dans une architecture non distribuée qui faisait passer des messages entre les différents nœuds afin de connaître l'état du réseau. Dans notre cas, nous nous sommes servis d'un système de marquage pour connaître l'avancement de la construction du MIS mais le principe de l'algorithme reste inchangé :

- `noir` : nœud dominant, appartient au MIS
- `gris` : nœud dominé, voisin d'un nœud dominant, n'appartient pas au MIS
- `blanc` : nœud encore non traité par l'algorithme
- `blanc actif` : état particulier d'un nœud potentiellement prêt à devenir dominant

Pour illustrer le fonctionnement de notre algorithme basé sur [1] nous allons prendre l'exemple ci-dessous :

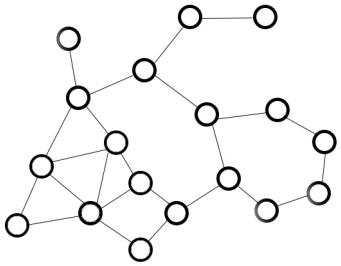

FIGURE 1 – MIS : état initial

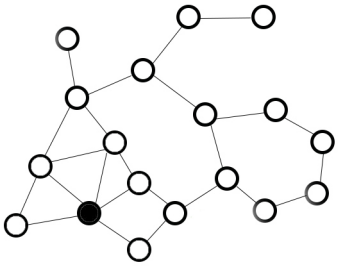

FIGURE 2 – MIS : leader

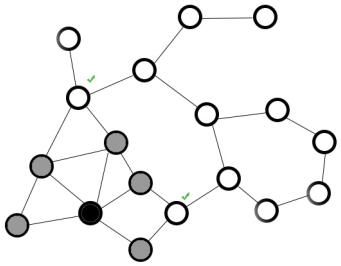

FIGURE 3 – MIS : domination

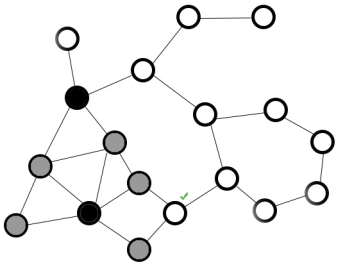

FIGURE 4 – MIS : élection

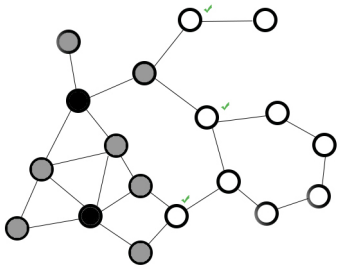

FIGURE 5 – MIS : domination

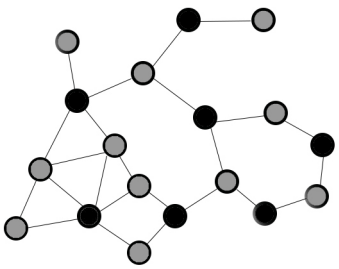

FIGURE 6 – MIS : état final

L'algorithme commence par choisir un tout premier nœud comme leader, en l'occurrence nous avons fait le choix de prendre le sommet de plus haut degré ce qui semble être pour nous un choix justifié puisque une plus large zone du graphe peut être couvert par ce seul point (figure 2). Ce nœud hôte est ainsi coloré en noir pour marqué son appartenance au MIS et tous ses voisins en gris pour indiquer leur domination par ce dernier. On liste ensuite les voisins des tous les nœud gris et les marque comme *actifs* ceci afin de choisir un nouveau nœud dominant (figure 3). Le prochain nœud à marqué en noir choisi parmi cette liste de *blancs actifs*, celui qui possède le plus grand nombre de voisins blancs *non actifs* devient dominateur (figure 4). Le reste de la liste des *blancs actifs* est maintenu et mis à jour dans la suite de l'algorithme. Puisqu'un nouveau nœud noir a été choisi, on peut répéter l'algorithme jusqu'à avoir marqué tous les nœud ou que la liste des *blancs actifs* soit vide (figure 5). Enfin, il reste plus qu'à renvoyer la liste de tous les nœuds en noir pour renvoyer l'ensemble des points qui compose le MIS (figure 6).

Il va ensuite s'agir d'étudier la complexité de cet algorithme. On suppose que le graphe est correctement construit et que nous avons facilement accès aux voisins d'un sommet donné, ce qui est le cas. L'algorithme parcourt simplement les points d'une liste qu'on maintient au fur et à mesure que nous marquons les points d'une certaine couleur ; cette même liste à mise à jour en parcourant les voisins du point qui est actuellement traité. L'algorithme s'arrête soit parce que la liste que l'on a construit est vide soit parce qu'il n'y a plus de points du graphe à traiter (ils ont tous été colorés). Cela nous donne une majoration sur la complexité de l'algorithme qui est en $O(\Delta \times n)$, avec Δ le degré maximal du graphe et n le nombre de sommets, ce qui correspond bien à un parcours de l'ensemble des points dans lequel on parcourt les voisins.

2.2 Construction de l'ensemble stable maximal : 2ème méthode

Un autre algorithme pour le MIS a été proposé par les auteurs. C'est un algorithme distribué qui se résume en les étapes suivantes :

- Construire un arbre couvrant quelconque du graphe (par le biais de messages)
- Choisir une racine et stocker la distance à la racine (le rang) dans chaque point (par le biais de messages)
- Colorer la racine en noir et envoyer un message `noir` a tout ses voisins dans le graphe (broadcast)
- Lorsqu'un point reçoit un message `noir`, il change sa couleur en gris et envoie un message `gris` contenant son rang à tout ses voisins dans le graphe (broadcast)
- Lorsqu'un point reçoit un message `gris`, il vérifie si le message gris vient d'un point au-dessus de lui (de rang inférieur) dans l'arbre, dans ce cas il décrémente un compteur `k` (initialisé a son nombre de voisins dans le graphe de rang inférieur dans l'arbre). Lorsque ce compteur arrive à zéro il devient noir et diffuse un message `noir`.
- Des envois de messages supplémentaires permettent à l'algorithme distribué de notifier la racine de la fin de la procédure.

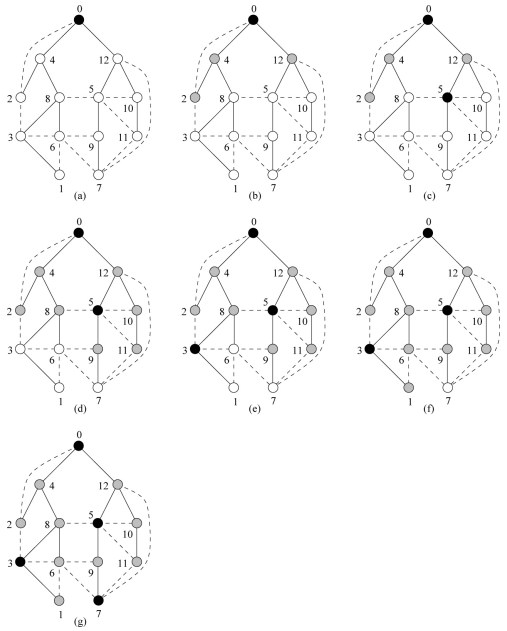

FIGURE 7 – Construction du MIS

Notre implémentation consiste à traduire cet algorithme distribué en un algorithme classique :

- Pour l'arbre couvrant : une implémentation inspirée de Kruskal sans prendre en compte la longueur des segments et en utilisant des *disjoint set*
- Une racine arbitraire est choisie et un parcours en profondeur de l'arbre est réalisé pour initialiser les rangs
- Ensuite un parcours en largeur de l'arbre reproduira le mieux le comportement de l'algorithme distribué :
  - On ajoute la racine dans une file puis tant que la file est non vide :
  - On dépile la liste et on parcours les voisins du nœud dépilé
  - On cherche un voisin noir : dans ce cas on change la couleur du nœud actuel en gris et on ajoute ses fils dans l'arbre dans la file
  - Sinon on compte les voisins gris qui ont un rang inférieur au nœud actuel, on change sa couleur en noir et on ajoute ses fils dans l'arbre dans la file
- On retourne la liste des nœuds noirs

Il s'agit maintenant de donner une borne a la complexité de cet implémentation. La construction de l'arbre couvrant consiste à parcourir tous les points du graphe. Pour chaque voisins n'étant pas dans le même d*Disjoint-Set* (opération `FIND`), faire l'union de leur *disjoint set* avec celui du point considéré et construire la structure d'arbre (stocker les voisins du point dans l'arbre). La complexité amortie des *Disjoint-Set* optimisés peut être considérée comme constante. On réalise pour chaque point du graphe de degré d, au maximum d `find` et d `union`.

Soit Δ le degré maximal de notre graphe et n le nombre de points. On est

donc en $O(n \times \Delta)$.

L'initialisation des rang est un parcours en profondeur de l'arbre et dépend de l'étape précédente. Le parcours en largeur effectué lors du reste de l'algorithme dépend aussi de la taille de l'arbre, chaque nœud a au maximum $\Delta - 1$ fils et il y a n nœuds. La complexité de cet algorithme est donc en $O(n \times \Delta)$.

2.3 Construction de l'ensemble dominant connexe

L'algorithme construit le CDS tel qu'il est décrit dans le papier :

```
compute a MIS satisfying lemmas 1 and 2.
color MIS nodes to black.
color other nodes to grey.
for i = 5; 4; 3; 2 do
  while there exists a grey node adjacent to at least i black nodes in
      different black-blue components do
    change its color from grey to blue;
return all blue nodes.
```

La principale difficulté au niveau de l'implémentation de l'algorithme se situe au niveau des *black-blue components*. Il faut rapidement identifier à quel *black-blue component* appartient un point noir. De plus, un nouveau point bleu choisi provoque la fusion de tout ses *black-blue components* voisins. La fusion de *black-blue components (BBC)* peut être coûteuse en fonction de l'implémentation. Une implémentation naïve consisterait à utiliser un ensemble de listes pour représenter les *BBC*, chaque liste contenant les points appartenant à un *BBC*. Considérons le cas au pire cas de toutes les opérations fusions de *BBC*.

Considérons le cas dégénéré d'un chemin de points de degré 2. Dans ce cas, chaque point gris, a exactement deux voisins noirs. Pour chaque point gris, on va réaliser une unique fusion (i.e. fusionner uniquement 2 *BBC*). Si l'on considère, pour simplifier l'analyse, que l'ordre de choix des points gris est tel qu'on va toujours fusionner un même BBC avec un BBC ne contenant qu'un seul point, dans le pire cas, on vide toujours la liste la plus grande dans la liste ne faisant que 1 en taille. On obtient le nombre d'opération ci dessous :

$$1 + 2 + 3 + 4 + \cdots + n = \frac{n(n+1)}{2} \text{ avec } n = taille(\texttt{MIS})$$

On obtient un coût en $O(n^2)$ pour les fusions.

Pour les recherches, on va faire un parcours de tous les BBC pour y trouver les voisins d'un point gris, ce qui, au pire cas où il faut parcourir tout les BBC pour y trouver les voisins, sachant que l'ensemble des points dans les BBC corresponds à la taille du MIS, on obtiens le nombre de comparaisons suivant :

$$n \times m \text{ avec } n = taille(\texttt{MIS}) \text{ et } m = taille(\texttt{ptsGris})$$

$$taille(\texttt{ptsGris}) = taille(\texttt{points}) - taille(\texttt{MIS})$$

Ces deux opérations sont, grossièrement, au pire cas quadratiques et il convient de les optimiser. Pour cela, nous allons utiliser une structure de *Disjoint-Set* pour réaliser des opérations `UNION` et `FIND` pour représenter les BBC.

Cette structure organise différents ensembles disjoints sous la forme d'une forêt d'arbres dont les nœuds pointent sur leurs parents. Au début, tous les BBC sont initialisés à un point noir et sont disjoints. Puis, lorsque l'on veut

faire l'union de deux BBC, on cherche leur racine et si elles sont différentes, un parent commun est créé pour les deux racines : opération `UNION`.

Lorsque l'on veut vérifier si deux points noirs appartiennent au même BBC, il suffit de remonter leurs parents jusqu'à la racine. Si deux points ont la même racine alors ils sont dans le même ensemble. Au pire cas, avec une implémentation naïve des Disjoint Set, les arbres peuvent prendre la forme d'un peigne et le coût au pire cas des opérations est en $O(n)$. Ce qui nous ramènerait a des complexités quadratiques pour la recherche et la fusion au cours de tout l'algorithme.

Cependant, deux optimisations simples sont très efficaces avec les `UNION-FIND` :

- la *path compression* : lors d'une opération recherche, on remonte les éléments traversés comme fils de la racine.

- l'*union par rang* : lors d'une fusion, si les deux arbres ont la même taille, on créé une nouvelle racine commune pour ces deux arbres. Cependant, quand un arbre est moins haut que l'autre, on peut seulement ajouter comme parent de la racine de l'arbre le moins haut la racine de l'arbre le plus haut.

Avec ces deux optimisations le coût amorti des deux opérations est en $O(a(n))$ avec a l'inverse de la fonction d'Ackermann, une fonction ayant une croissance si lente qu'on peut la considérer comme une constante inférieure à 5 pour une situation quelconque.

Il est justifié de parler de coût amorti dans notre cas car on réalisera toujours des opérations `find` (réalisant la *path compression*) pour trouver les *black blue components* différents qui sont voisins d'un point gris, avant de réaliser des opérations `union` pour fusionner tous les BBC voisins d'un point nouvellement bleu.

`MakeSet` créé des sets singletons pour l'initialisation :

```
function MakeSet(x)
  if x is not already present:
    add x to the disjoint-set tree
    x.parent := x
    x.rank   := 0
```

`Find` implémente la recherche de la racine avec la *path compression* :

```
function Find(x)
  if x.parent != x
    x.parent := Find(x.parent)
  return x.parent
```

`Union` fait l'union par rang :

```
function Union(x, y)
  xRoot := Find(x)
  yRoot := Find(y)

  // x and y are already in the same set
  if xRoot == yRoot
      return
```

```
  // x and y are not in same set, so we merge them
  if xRoot.rank < yRoot.rank
    xRoot.parent := yRoot
  else if xRoot.rank > yRoot.rank
    yRoot.parent := xRoot
  else
    //Arbitrarily make one root the new parent
    yRoot.parent := xRoot
    xRoot.rank  := xRoot.rank + 1
```

L'utilisation des *Disjoint Set* a d'autres avantages : l'algorithme est beaucoup plus simple à écrire une fois cette structure implémentée. De plus, cette structure permet facilement de vérifier la connexité d'un graphe. Elle nous sera donc utile pour implémenter la fonction de vérification `isValid` et pour générer des graphes géométriques connexes.

Nos structures :

```
class NodeVertexDS {
  Point p;
  Color c;

  ArrayList<NodeVertexDS> neighbors;
  DisjointSetElement<NodeVertexDS> disjointsetelement;

  //d'autres attributs sont utilises pour definir la structure d'arbre
      necessaire pour l'algorithme du MIS 2
  //d'autres attributs sont utilises pour l'algorithme du MIS 1
}

class DisjointSetElement<T> {
  int index; //unique index
  T data;
  DisjointSetElement<T> parent;
  int rank;

  //toutes les methodes pour implementer les disjoint set : en
      particulier find et union
}
```

Pseudo-code de notre implémentation :

```
//construire la structure du graphe avec la liste de points en entree et
    edgeTreshold
graph : ArrayList<NodeVertexDS> = makeGraph(points);
MIS : ArrayList<NodeVertexDS> = calculMIS(graph);
grayNodes : ArrayList<NodeVertexDS = { graph } \ { MIS }

for(n : MIS) { n.color = BLACK; n.disjointsetelement = new
    DisjointSetElement<NodeVertexDS>(n); }
```

```
for( i from 5 to 2 ) {
  cont = true;
  while(cont) {
    newblue = null;
    for( gray in grayNodes ) {
      if( gray.degree() < i ) continue;

      //assuming BlkNeighbors returns the black neighbors of a node
      blackneighbors : ArrayList<NodeVertexDS> = BlkNeighbors(gray);
      if( blackneighbors.size() < i ) continue;

      //assuming map is the functional transformation on list
      rootsblackneighbors : ArrayList<DisjointElement> = List.map(
          blackneighbors, (x) => x.disjointsetelement.find() )

      int nbDifferentBBC = new HashSet(rootsblackneighbors).size();
      if( nbDifferentBBC < i ) continue;

      gray.color = BLUE;
      newblue = GRAY;

      for( i from 1 to rootblackneighbors.size() -1)
          rootblackneighbors[i].union(rootblackneighbors[0]);
      break;
    }
    if ( newblue != null ) grayNodes.remove(newblue);
    else cont = false;
  }
}
return BLACK and BLUE nodes;
```

3 Résultats

3.1 Tests

Pour tester notre algorithme nous avons eu besoin de deux choses :
- une fonction permettant de valider notre solution
- un fonction permettant de générer aléatoirement des tests

La méthode `isValid()` qui vérifie la validité de la solution calculée par notre algorithme est donnée par le pseudo-code suivant :

```
isValid(ArrayList<Point> graph, ArrayList<Points> cds, int edgeTreshold)
    {
   valid = true;
   //build the graph structure of the cds
   ArrayList<NodeVertexDS> graphcds = graph(cds);

   //compute connex components
   for( v in graphcds ) {
      for( vn in v.neighbors ) {
         v.disjointsetelement.union(vn.disjointsetelement)
      }
   }

   compconnex = new HashSet<DisjointSetElement>();
   for( v in graphcds ) { compconnex.add(v.disjointsetelement.find()); }

   if(compconnex.size() > 1) { print("Error on connexity : " +
       compconnex.size()); valid = false) }

   rest = points.clone();
   rest.removeall(cds);

   //remove all neighbors of elements of cds from rest
   removeneighbors(rest, cds, edgeTreshold);

   if(rest.size() > 0) { print("Error dominating : " + rest.size());
       valid = false; }

   return valid;
}
```

Nous avons aussi eu besoin de générer des tests. Pour cela il convient de générer des graphes géométriques de différentes tailles et avec des seuils différents (valeur maximale pour que 2 points soient considérés voisins l'un de l'autre). Voici le pseudo code du générateur :

```
generateGraph(width, heigth, nb, edgeTreshold) :
   result : ArrayList<Point>;
   while result.size() != nb :
      add random points in result until result.size() == nb
      compute connected components of result
```

```
if there is multiple connected components :
   keep only the biggest connected components in points (remove
       the points that are in smaller components)
```

Nous avons ensuite établi plusieurs bases de test :

Nombre de points	Largeur × Hauteur	Seuil
100	1000 × 1000	50
500	1000 × 1000	50
1000	1000 × 1000	50
5000	1000 × 1000	50
10000	1000 × 1000	50

TABLE 1 – Base de tests 1

Nombre de points	Largeur × Hauteur	Seuil
100	1000 × 1000	5
500	500 × 500	25
1000	1000 × 1000	50
5000	5000 × 5000	250
10000	10000 × 10000	500

TABLE 2 – Base de tests 2

Nombre de points	Largeur × Hauteur	Seuil
100	100 × 100	25
500	500 × 500	36
1000	1000 × 1000	50
5000	5000 × 5000	161
10000	10000 × 10000	300

TABLE 3 – Base de tests 3

Nous avons mené nos tests sur les 2 implémentations possibles du MIS dot les résultats ont été consignés sur les graphes suivants :

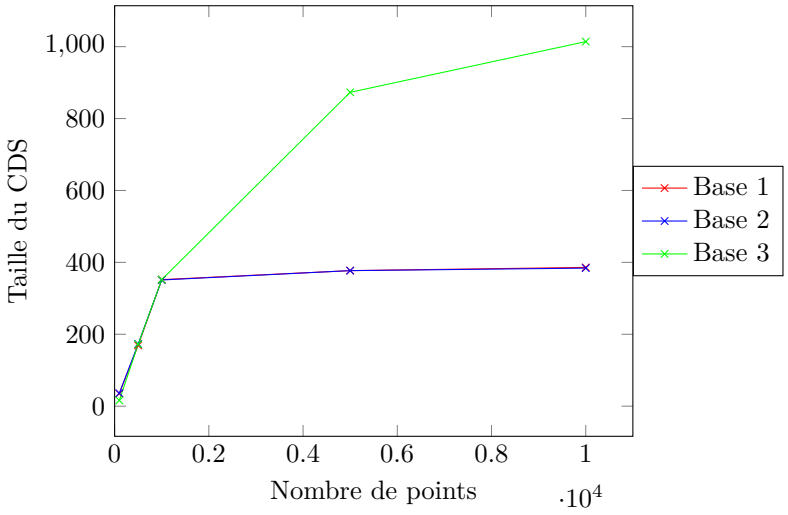

FIGURE 8 – Taille du CDS en fonction du nombre de points avec MIS1

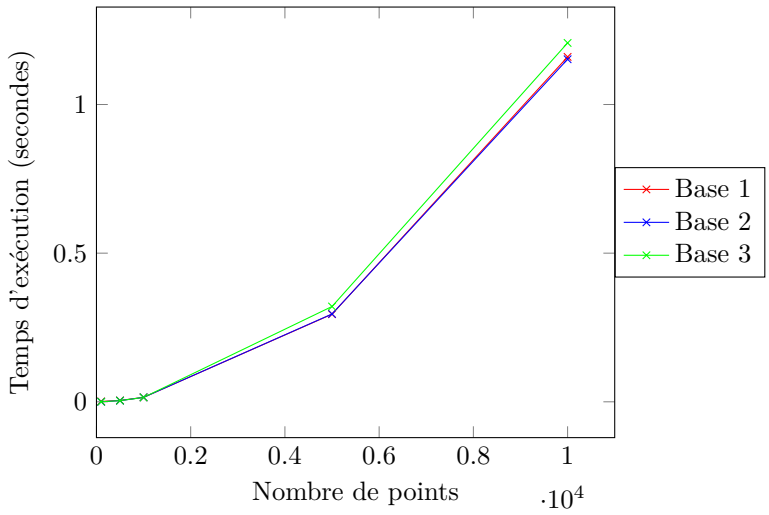

FIGURE 9 – Temps d'exécution en fonction du nombre de points avec MIS1

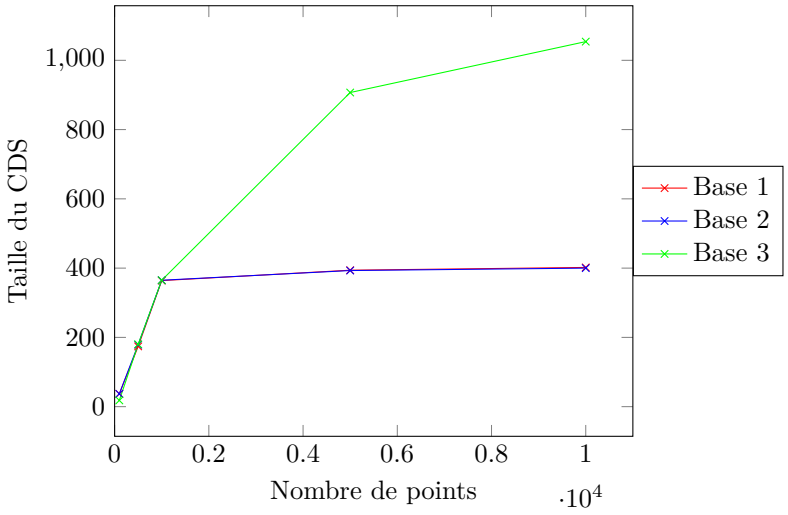

FIGURE 10 – Taille du CDS en fonction du nombre de points avec MIS2

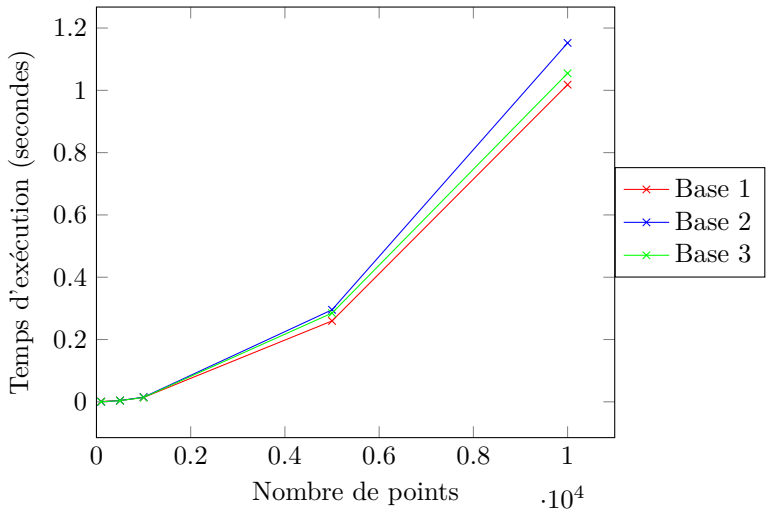

FIGURE 11 – Temps d'exécution en fonction du nombre de points avec MIS2

3.2 Discussion

4 Conclusion

Finalement, il s'avère que l'algorithme Li et al. est très intéressant d'un point de vue temps d'exécution. En effet, l'algorithme glouton qui est proposé est de complexité linéaire ce qui assure de trouver une solution en un temps séduisant. Cependant, bien que l'on puisse penser à une modeste qualité venant de la part d'un algorithme glouton, on nous assure un ratio de $(4.8 + ln5)opt + 1.2$ avec *opt* la solution optimale, ce qui est remarquable. Nous avons donc ici, un compromis plus qu'intéressant entre temps de calcul et approximation de la meilleure solution.

On peut se poser une question sur cet algorithme, que les auteurs n'ont pas abordé : la pertinence d'un *local searching*. En effet, il est coutume d'améliorer une solution donnée par une approche gloutonne au moyen de nombreuses micro optimisations que l'on appelle *local searching*. On pourrait par exemple, prendre n points de notre solution et tenter de les substituer par $n - 1$ points tout en conservant les propriétés d'un ensemble dominant connexe.

Références

[1] Mihaela Cardei, Maggie Xiaoyan Cheng, Xiuzhen Cheng, and Ding-Zhu Du. Connected domination in multihop ad hoc wireless networks. In *JCIS*, pages 251–255, 2002.

[2] Bo Gao, Yuhang Yang, and Huiye Ma. A new distributed approximation algorithm for constructing minimum connected dominating set in wireless ad hoc networks. *International Journal of Communication Systems*, 18(8) :743–762, 2005.

[3] Yingshu Li, My T Thai, Feng Wang, Chih-Wei Yi, Peng-Jun Wan, and Ding-Zhu Du. On greedy construction of connected dominating sets in wireless networks. *Wireless Communications and Mobile Computing*, 5(8) :927–932, 2005.

[4] Wikipedia. Graphe de disques — wikipedia, the free encyclopedia. `https://fr.wikipedia.org/wiki/Graphe_de_disques`. [consulté le 15-Oct-2017].

[5] Wikipedia. Maximal independent set — wikipedia, the free encyclopedia. `https://en.wikipedia.org/wiki/Maximal_independent_set`. [consulté le 15-Oct-2017].