SORBONNE UNIVERSITÉ

Projet : 2i006

Le problème de Via Minimization

Réalisé par :

BOUSBA Abdellah

HADDADI Hacene

Enseignants :

Pierre Fouilhoux

Etienne Simon

Mathilde Carpentier

2ème année Licence Informatique
Mono-disciplinaire
Groupe n°3
2018-2019

Table des matières

Partie A :

1) Description des structures implémentées

Dans cette partie, on cherche à trouver les intersections entre les segments des différents réseaux d'une Netlist, on implémentera donc trois méthodes différentes afin d'obtenir celle avec la meilleure complexité.

Une première méthode dite naïve, une seconde avec des liste linéaires chaînées et une troisième avec des ABR AVL.

Pour cela, nous avons besoin de manipuler les structures suivantes :

- Une structure nommée netlist qui contiendra les données de nos netlist.
- Une structure nommée Echéancier sous forme d'ABR qui contiendra les extrémités de tous les segments de la netlist triés par abscisse (une seule extrémité est ajoutée dans l'échéancier pour les segments verticaux).
- Une structure nommée AVL et une autre nommée Cell_segement représentant respectivement un arbre de recherche binaire et une liste linéaire chaînée qui permettront De réaliser la fonction de balayage.

Toutes les méthodes pour traiter l'insatance A ont été implémenté dans le fichier netlist.c, les structures de données dans netlist.h.

2) Complexité des algorithmes

i) Complexité Intersect_Naïf

On parcourt tous les segments, et pour chacun on reparcours la liste des segments et on vérifie si il y'a intersection entre eux, ce qui donne une complexité de $O(n^2)$ avec n le nombre de segments.

ii) Complexité Intersect_balayge

Soit n le nombre de segment, cette fonction fait un simple appel à :

- creer_echancier qui est en O(n.log(n))
- Intersect_balayge_rec qui fait appel à :
  - insérer_segment qui est en O(n)
  - supprimer_segment qui est en O(n)
  - prem_segment_apres qui est en O(n)
  - au_dessus qui est en O(n)

Donc la complexité de intersect_balayage égale au max entre celle de créer_echancier et de Intersect_balayge_rec.

Calculons la complexité de Intersect_balayge_rec :

Soit C(n) le nombre d'appels récursif à intersect_balayge_rec, donc on a :

$C(n) = 2.C(n/2) + n + n^2 = 4\,C(n/4) + (n + n/2) + (n^2 + n^2/2^2)$

Et par substitution on arrive à :

$C(n) = 2^k C(n/2^k) + (n + n/2 + \dots + n^{k-1}/2) + (n^2 + n^2/2^2 + \dots + n^2/2^{k-1})$

Avec : $2^k = n$ donc $k = \log_2(n)$ et $C(n/2^k) = C(1) = 1$

Donc : $C(n) = n + n(\frac{1-(\frac{1}{2})^k}{\frac{1}{2}}) + n^2 (\frac{1-(\frac{1}{2})^k}{\frac{1}{2}})$

Donc intersect_balayge_rec en $O(n^2)$

Ce qui signifie que intersect_balayge est en $O(n^2)$.

Soit α une borne de segment horizontaux traversés à un moment donné :

La complexité de intersect_balayge_rec devient :

$C(n) = 2.C(n/2) + \alpha + \alpha^2 = 4.C(n/4) + (\alpha + \alpha/2) + (\alpha^2 + \alpha^2/2^2)$

Et par substitution on arrive à :

$C(n) = 2^k C(n/2^k) + (\alpha + \alpha/2 + \ldots + \alpha^{k-1}/2) + (\alpha^2 + \alpha^2/2^2 + \ldots + \alpha^2/2^{k-1})$

Avec : $2^k = n$ donc $k = \log_2(n)$ et $C(n/2^k) = C(1) = 1$

Donc $C(n) = n + \alpha (\frac{1-(\frac{1}{2})^k}{\frac{1}{2}}) + \alpha^2 (\frac{1-(\frac{1}{2})^k}{\frac{1}{2}})$

Donc intersect_balayge_rec en $O(n+ \alpha^2)$ qui est une complexité linéaire

Ce qui signifie que intersect_balayge est en $O(n.\log(n))$.

iii) Complexité Intersect_balayge_AVL

Cette fonction fait un simple appel à :

- Intersect_balayge_rec qui fait appel à :
  - insérer_segment qui est en $O(n)$
  - supprimer_segment qui est en $O(n)$
  - prem_segment_apres qui est en $O(n)$
  - au_dessus qui est en $O(n)$
- créer_echancier qui est en $O(n.\log(n))$

Donc la complexité de intersect_balayage_AVL égale au max entre celle de créer_echancier et de Intersect_balayge_AVL_rec.

Calculons la complexité de Intersect_balayge_rec :

Soit C(n) le nombre d'appels récursif à intersect_balayge_AVL_rec avec n le nombre de segments, donc on a :

$C(n) = 2.C(n/2) + \log_2(n) + n. \log_2(n) = 4.C(n/4) + (\log_2(n) + 2\log_2(n/2)) + n.(\log_2(n) + \log_2(n/2))$

Et par substitution on arrive à :

$C(n) = 2^k C(n/2^k) + (\log_2(n) + 2.\log_2(n/2) + \ldots + 2^{k-1}.\log_2(n/2^{k-1})) + n. (\log_2(n) + \log_2(n/2) + \ldots + \log_2(n/2^{k-1}))$

Avec : $2^k = n$ donc $k = \log_2(n)$ et $C(n/2^k) = C(1) = 1$

Donc $C(n) = n + \log_2(n) + n.\log_2(n)$

Donc intersect_balayge_AVL_rec en $O(n.\log_2(n))$

Ce qui signifie que intersect_balayge_AVL est en $O(n.\log_2(n))$

Soit α une borne de segment horizontaux traversés à un moment donné :

La complexité de intersect_balayge_rec devient :

$C(n) = 2.C(n/2) + \log_2(\alpha) + \alpha.\log_2(\alpha) = 4.C(n/4) + (\log_2(\alpha) + 2\log_2(\alpha /2)) + \alpha.(\log_2(\alpha) + \log_2(\alpha /2))$

Et par substitution on arrive à :

$C(n) = 2^kC(n/2^k) + (\log_2(\alpha) + 2.\log_2(\alpha /2) + \ldots + 2^{k-1}.\log_2(\alpha /2^{k-1})) + \alpha. (\log_2(\alpha) + \log_2(\alpha /2)+\ldots+\log_2(\alpha/2^{k-1}))$

Avec : $2^k = n$ donc $k = \log_2(n)$ et $C(n/2^k) = C(1) = 1$

Donc $C(n) = n + \log_2(\alpha) + \alpha.\log_2(\alpha)$

Donc intersect_balayge_AVL_rec en $O(n + \alpha .\log_2(\alpha))$

Ce qui signifie que intersect_balayge_AVL est en $O(n.\log_2(n))$

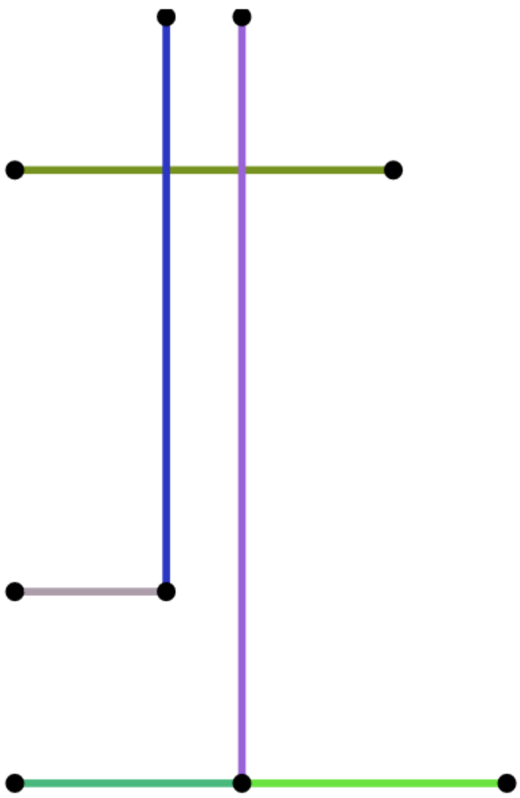
3) Description des jeux d'essais

i) Test de validation du code

Pour tester la validité des trois algorithmes qui recherchent les intersections, on fait le test sur le fichier test.net qui contient la netlist suivante :

```
3
0 2 1
  0 20 40
  1 120 40
  0 1
1 3 2
  0 60 0
  1 60 150
  2 20 150
  1 2
  0 1
2 4 3
  0 80 0
  1 80 200
  2 20 200
  3 150 200
  1 2
  1 3
  0 1
```

Ficher test.net



Visualisation de test.net

Résultat attendu :

On doit donc trouver 2 intersections entre les segments (0,1) du réseau 0 et (0,1) du réseau et entre les segments (0,1) du réseau 0 et (0,1) du réseau 1

Le résultat :

```
0 0 1 2 0 1
0 0 1 1 0 1
```

Fichier des intersections de intersect_naïf

```
0 0 1 2 0 1
0 0 1 1 0 1
```

Fichier des intersections de intersect_balayage

```
0 0 1 2 0 1
0 0 1 1 0 1
```

Fichier des intersections de intersect_balayage_AVL

Interprétation :

Il y'a deux intersections dans tous les fichiers :

- Entre les segments (0,1) du réseau 0 et (0,1) du réseau 2
- Entre les segments (0,1) du réseau 0 et (0,1) du réseau 1

Conclusion :

Les trois algorithmes ont bien trouvé les bonnes intersections.

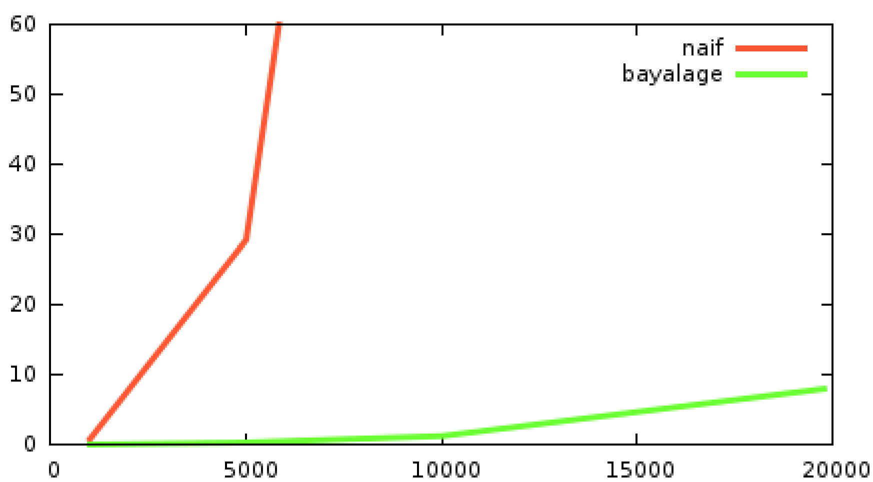
ii) Evaluation des performances

Pour évaluer les performances des algorithmes on fait le test sur tous les fichiers de InstanceA qui contient des netlists de différente taille qui ont tous une borne de segments horizontaux traversés à un moment donné, donc : intersect_naïf est en $O(n^2)$, intersect_balayage et intersect_balayge_AVL sont en $O(n.\log(n))$.

a) Comparaison entre intersect_naïf et intersect_balayge

Exception : intersect_naif doit être quadratique et intersect_balayage linéarithmique, et intersect_balayage doit être plus rapide que intersect_naïf

Résultat :

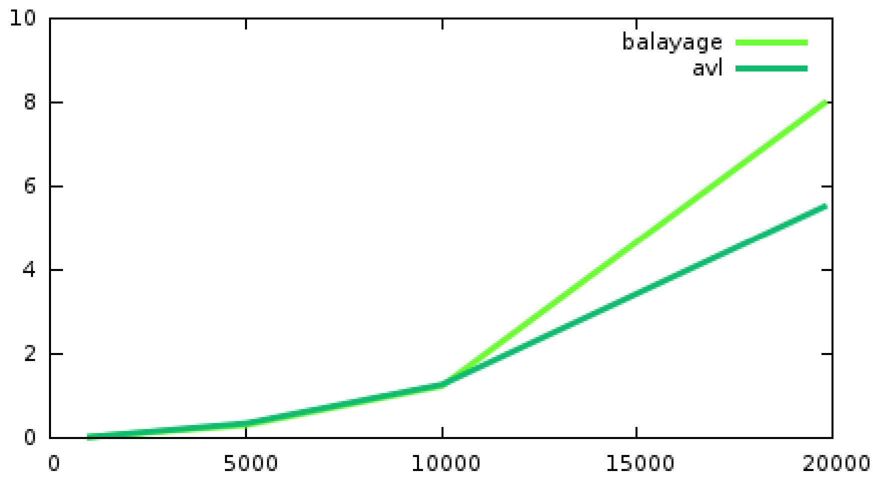


Interprétation : intersect_naif est bien quadratique et intersect _balayage linéarithmique, et intersect _balayage est beaucoup plus perfomant que intersect _naif.

b) Comparaison entre intersect_balayage et intersect_balayge_AVL

Exception : *intersect_balayage* et *intersect_balayage*_AVL doivent être linéarithmique et plus ou moins de même rapidité.

Résultat :



Interprétation : *intersect_balayage* et *intersect_balayage*_AVL sont bien linéarithmique et intersect_balayage_AVL est légèrement plus rapide que intersect_balayage.

Partie B

1) Description des structures implémentées

Dans cette partie, nous allons étudier des méthodes pour positionner les segments sur les faces en minimisant le nombre de vias correspondant à cette affectation, on implémentera donc deux méthodes différentes afin d'obtenir celle avec la meilleure complexité.

Pour cela, nous avons besoin de manipuler les structures suivantes :

- Une structure nommée Graphe qui contiendra les données de nos netlist dans des sommets.
- Une structure nommée Sommet qui contiendra des segments ou des extrémités.

Toutes les méthodes pour traiter la partie B ont été implémenté dans le fichier netlistG.c et les structures de données dans netlist.h et netlistG.h.

2) Complexité des algorithmes

i) Complexité getVia

getVia est une fonction itérative qui parcourt tous les sommets du graphe de taille n, et pour chaque sommet deux cas de figure sont possible :

- Si c'est un sommet segment, on ne fait que des instructions simples.
- Si c'est un sommet extrémité, on parcourt toute sa liste d'adjacence qui aura α sommet avec α << n.

Donc getVia fera n fois α instructions ce qui fait une complexité de O(n).

ii) Complexité de bicolore

Cette fonction fait un simple appel à Ajout_vias_cycle_impair puis parcourt les n sommet du graphe, et pour chaque sommet on parcourt sa liste d'adjacence, qui dans le pire cas vaudra m le nombre d'arrêtes du graphe.

Ce qui nous donne une complexité de O(n.m) au sein de la boucle

Donc la complexité de bicolore est égale au max entre celle de Ajout_vias_cycle_impair et O(n.m).

Calculons la complexité de Ajout_vias_cycle_impair :

Ajout_vias_cycle_impair fait appel à detect_cycle_impair qui fait lui appel detect_cycle_impair_rec.

Complexité de detect_cycle_impair_rec :

Soit C(m) le nombre d'appels récursif à detect_cycle_impair_rec avec m le nombre d'arrêtes du graphe, donc on a :

$C(m) = C(m-1) + \alpha = C(m-2) + 2.\alpha$, avec α le nombre d'instructions exécutés à chaque appel

Et par substitution on arrive à :

$C(m) = C(0) + m.\ \alpha$

Avec : $C(0) = 0$ et $1 < \alpha << m$

Donc : $C(m) = \alpha.m$

Donc detect_cycle_impair_rec en O(m)

Complexité de detect_cycle_impair:

Soit n la taille du graphe

detect_cycle_impair initialise un tableau de marquage avec une boucle de n itération, puis fait appel à detect_cycle_impair_rec sur tous sommet du graphe et ce jusqu'à trouver un cycle impair ou atteindre n itération, mais à chaque parcourt il ne repasse pas sur les sommets déjà visités dans les appels précédent, donc dans le pire cas on parcourt tous les sommets et les arrêtes ce qui nous fait une complexité de O(n+m) .

Donc detect_cycle_impair est en O(n+m)

Ajout_vias_cycle_impair fait un simple appel à detect_cycle_impair, puis tant qu'elle trouve des cycles impairs dans le graphe, elle place des vias et refait appel à detect_cycle_impair.

Soit β le nombre de cycle impair dans le graphe, donc Ajout_vias_cycle_impair fait β appels à detect_cycle_impair

Donc Ajout_vias_cycle_impair est en $O(\beta(n+m))$.

Ce qui signifie que bicolore est en $O(\max(n.m, \beta(n+m)))$.

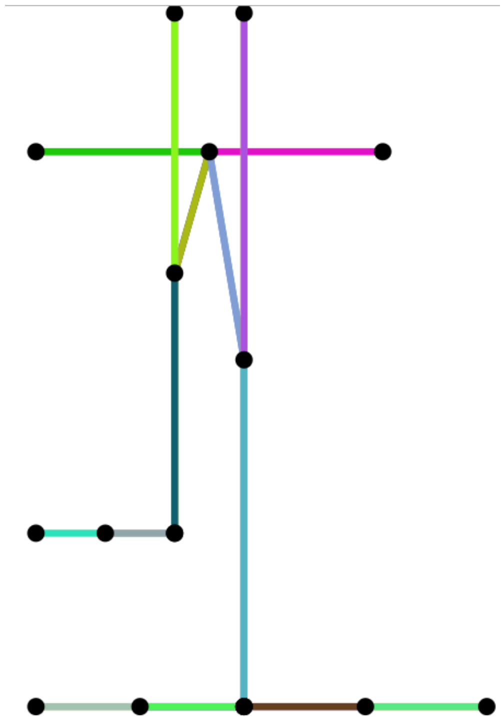
3) Description des jeux d'essais

i) Test de validation du code

Pour tester la validité des trois algorithmes qui recherchent les intersections, on fait le test sur les fichiers test.net et c.net qui contiennent les netlists suivantes :

```
3
0 2 1
  0 20 40
  1 120 40
  0 1
1 3 2
  0 60 0
  1 60 150
  2 20 150
  1 2
  0 1
2 4 3
  0 80 0
  1 80 200
  2 20 200
  3 150 200
  1 2
  1 3
  0 1
```

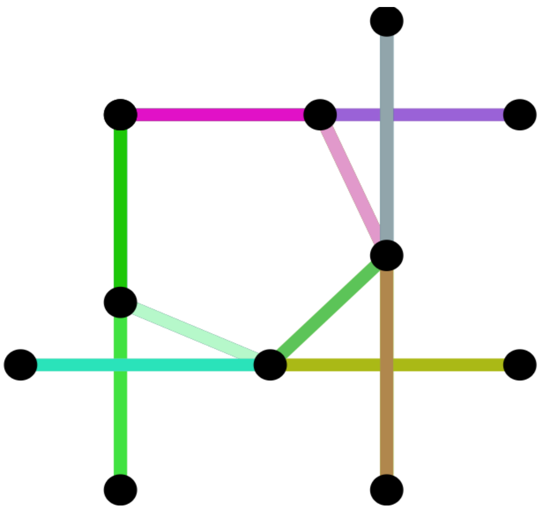
Fichier test.net



Visualisation de test.net

```
3
0 3 2
  0 20 80
  1 20 20
  2 80 20
  0 1
  1 2
1 2 1
  0 5 60
  1 80 60
  0 1
2 2 1
  0 60 5
  1 60 80
  0 1
```

Fichier c.net



Visualisation de c.net

Résultat attendu :

Pour test.net :

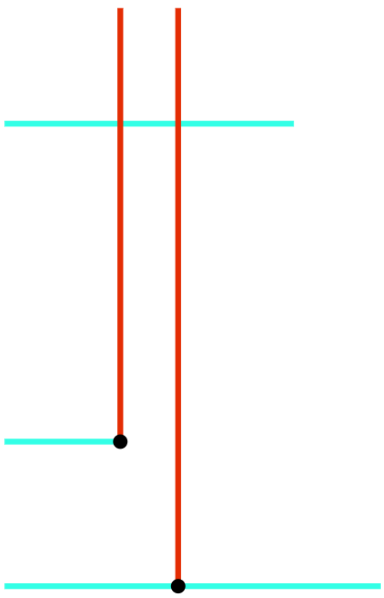
- Avec getVia : on doit trouver 2 vias car il y a deux points qui sont extrémités de segments verticaux et horizontaux à la fois.
- Avec bicolore : on doit trouver 0 via car il n'y a pas de cycle impair dans le graphe.

Pour c.net :

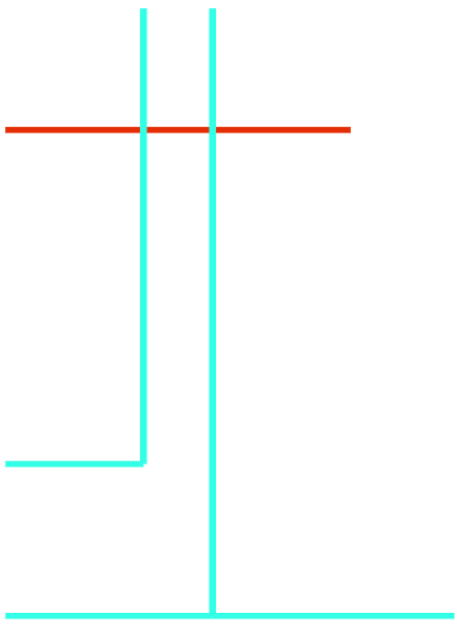
- Avec getVia : on doit trouver 1 via car il n'y a qu'un seul point qui est extrémité d'un segment vertical et horizontal à la fois.
- Avec bicolore : on doit trouver 1 via car il n'y a qu'un seul cycle impair dans le graphe et doit être placé sur un sommet extrémité du cycle.

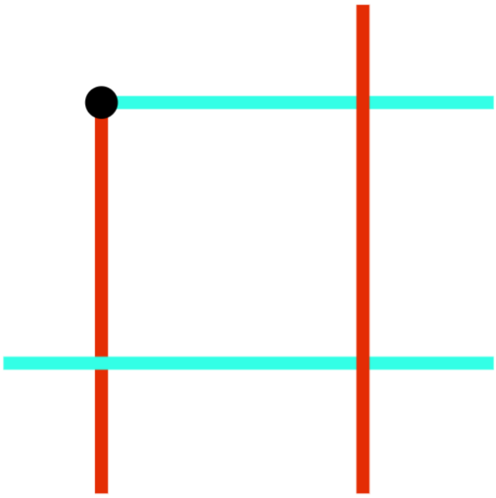
Le résultat :

Visualisation des vias du fichier test.net avec getVia

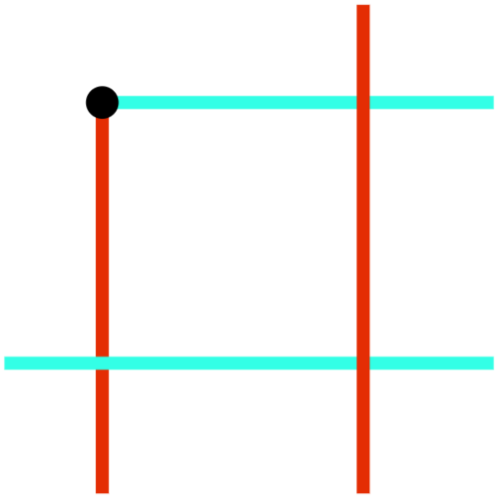


Visualisation des vias du fichier test.net avec bicolore





Visualisation des vias du fichier c.net avec getVia



Visualisation des vias du fichier c.net avec bicolore

Interprétation :

Fichier test.net :

- Avec getVia on obtient deux vias, qui sont les deux des extrémités de segments verticaux et horizontaux à la fois.
- Avec bicolore on n'a aucun via car le graphe ne présente pas cycle impair.

Fichier c.net :

- Avec getVia on a un seul via qui correspond bien à l'unique point qui est extrémité d'un segment vertical et d'un autre horizontal.
- Avec bicolore on a un seul via qui correspond bien à une extrémité dans le cycle impair.

Conclusion :

Les deux algorithmes ont bien trouvé les bons résultats.

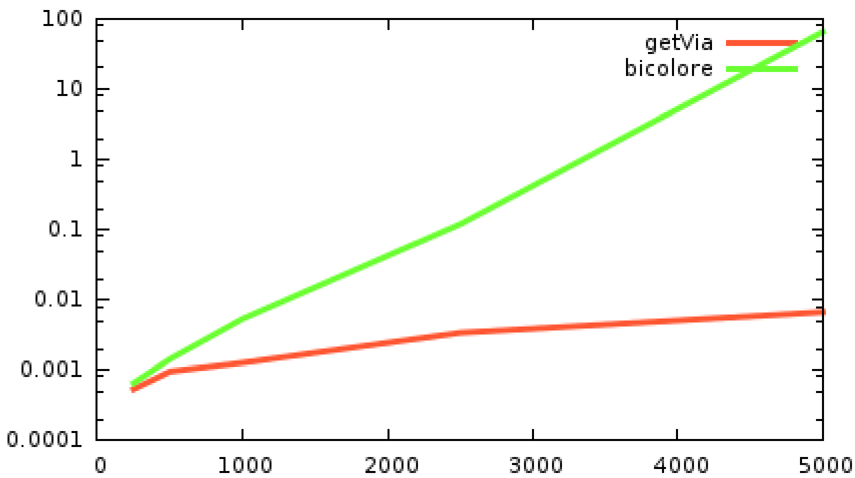
ii) Evaluation des performances

Pour évaluer les performances des deux algorithmes on fait le test sur tous les fichiers de InstanceA qui contient des netlists de différente taille, on s'intéresse ici à la rapidité et le nomre de vias généré par les deux algorithmes.

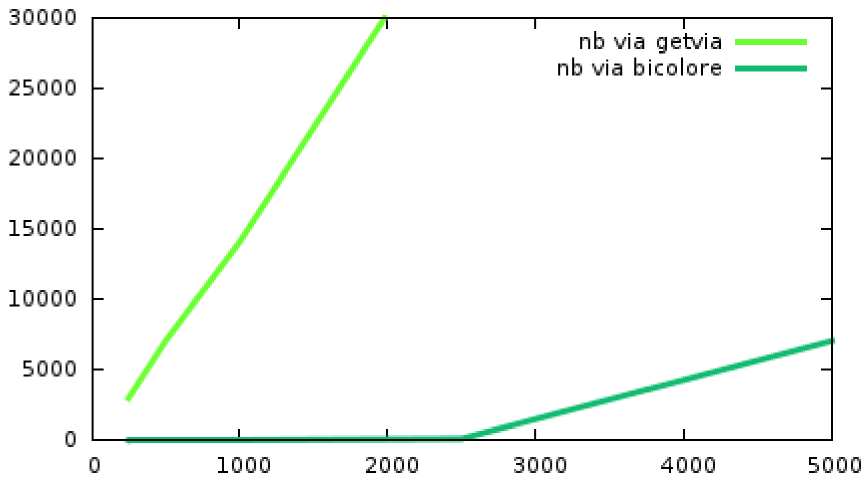
getVia est en O(n) et bicolore est en O(max (n.m, β(n+m))).

Exception : getVia doit être linéaire et bicolore quadratique donc getVia doit être plus rapide que bicolore.

Résultat :



Courbes représentant les performances des algorithmes



Courbes représentant le nombre de via généré par les deux algorithmes

Interprétation : getVia plus performante que bicolore comme prévue, mais on remarque qu'elle génère plus de via que ce dernier.

Conclusion : en dessous d'une vingtaine de milliers de segments on préférera largement utiliser bicolore que getVia car elle est plus performante sur tous les plans, au-delà getVia devient moins chronophage que bicolore mais génère un nombre assez conséquent de vias, il faudra donc adapter le choix de la méthode selon les besoins.