# Rapport de Projet

## 2I006

XIANG YANGFAN

3300401

Mercredi 15 Avril 2015

***Partie 1***

I) Structure general de code

L'ensemble de mon code est séparé en plusieurs fichiers différents pour un meilleur vu de la structure général et organisés par leur fonctionnalité.

-*structure.c* : contient toutes la structure de base de circuit et les fonctions primaires sur la structure de base.

-*fonction.c* : les fonctions primaires d'opération sur la structure.

-*fonctionFile.c* : les fonctions permettant de lire ou écrire des données dans un fichier et les fonctions permettant de dessiner la solution du problème de Via avec un tableau de solution.

-*structure_AVL.c* : la structure de AVL avec les fonctions de manipulation primaires.

-*graphe.c* : la structure de graphe non orienté et les fonctions de manipulation et de création.

-*resolution.c* : les fonctions permettant de résoudre le problème d'intersection entre les segments et stocké la solution dans un tableau.

-*Makefile* : l'automatise la tache de la compilation de l'ensemble des codes.

-*mainJeuxTest* : permet d'effectuer une test de jeux pour vérifier la validité des fonctions qu'on peut choisir sur quel circuit on veut tester et avec quel méthode de recherche d'intersection. Il suffit simplement de l'exécuter et suive les étapes dans le terminal.

II) Analyse synthétique des fonctions

*Intersect_naif* :

Soit la fonction *intersect_naif* prend en argument un *Netlist* et pour chaque segment qui est dans cette Netlist on cherche tous les autres segments qui s'intersecte avec cette segment. Notons n le nombre de segment. On commence d'abord a récupéré tous les segments et le stocké dans un tableau de pointeur sur le segment. Cette opération coute $\theta\,(n)$.

Puis pour tous les segments dans le tableau on recherche les segments qui s'intersecte dans le même tableau, on a donc deux boucles for imbriqué. La complexité finale de la fonction est de $\theta\,(n^2) + \theta\,(n) = \theta\,(n^2)$ .

*Intersection_balayage* :

La fonction *intersection_balayage* prend aussi en argument un *Netlist* et cherche pour chaque segment les segments qui s'intersecte par la methode de balayage. Notons aussi n le nombre de segment. On construit d'abord un tableau de pointeur sur la structure *Extremite*. Cette opération coute $\theta\,(n)$ et le tableau d' *Extremite* contient au plus $2n$ extrémités dans le cas ou tous les segments sont horizontaux donc 2 points d'extrémité pour chaque segment. Pour pouvoir trier le tableau d'Extremite j'utilise un tri rapide qui à une complexité en $\mathrm{O}\,(taille \log taille)$, taille étant la taille du tableau.

Supposons que pour chaque extrémité on a dans T n segments à vérifier, on a une complexité dans le pire de cas $\mathrm{O}\,(n^2)$.
Si on introduit $\alpha$ ne borne maximale sur nombre de segments horizontaux traversé par la droite quelque soit l'abscisse et m le nombre de segments verticaux. Dans le pire de cas la complexité en introduisant $\alpha$ est $\mathrm{O}\,(m \times \alpha)$ avec $m + \alpha \leq n$.

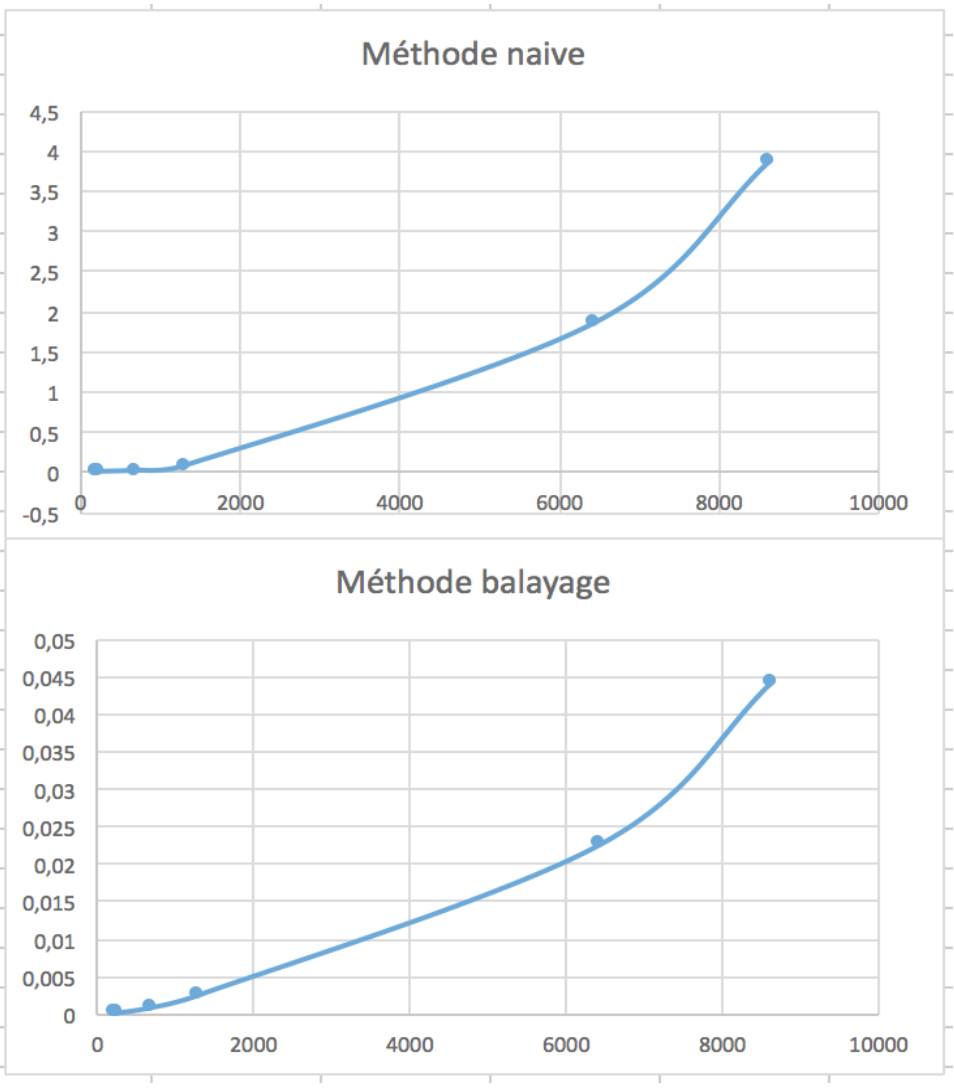
III) Choix des structures

Pour stocker les points d'extrémité j'ai choisi d'utiliser un tableau de pointeur sur point d'extrémité. Il y a plusieurs raisons à cela, d'abord on veut parcourir les points d'extrémités dans l'ordre croissant des abscisses donc le mieux est de trier. Un tableau est très adapté pour un tri avec des accès et de modification rapides, puisque se sont des pointeurs on économise de la mémoire et du temps si on compare avec une liste ou il faut faire des malloc pour les cellules.

Dans la méthode de balayage la structure T est une liste chainé, ainsi la fonction insertion d'un élément dans l'ordre et la suppression d'un élément ont une complexité de $\theta\,(n)$ si n est la longueur de la chaine. Si on opte pour une structure AVL l'insertion et la suppression ont alors une complexité $\mathrm{O}\,(\log n)$. En revanche une liste chainé est plus facile à coder comparé au AVL plus particulièrement pour la fonction *AuDessus* qui dans le cas de la chaine pendra simplement celui qui suit alors que pour un AVL si le fils droite est vide il faut monté dans l'arbre et de chercher dans les pères.

IV) Statistique

| Nom | Nombre de segment | Méthode naïve Temps(s) | Méthode balayage Temps(s) |
|---|---|---|---|
| alea0030_030_10_088.net | 209 | 0.002338 | 0.000247 |
| alea0030_030_90_007.net | 241 | 0.001957 | 0.000286 |
| alea0100_050_10_097.net | 693 | 0.016206 | 0.001037 |
| alea0100_080_90_024.net | 1293 | 0.073703 | 0.002669 |
| alea0300_300_10_044.net | 6441 | 1.855673 | 0.022720 |
| alea0300_300_90_099.net | 8639 | 3.875221 | 0.044321 |



On observe que dans la réalité la silhouette des deux courbes se ressemble, c'est sans doute le fait de manipuler en plus la structure de chaine (insertion, suppression, recherche) dans la fonction de balayage.

***Partie 2***

Nous avons deux méthodes pour résoudre le problème.

Le première est celui de deux faces, les segments horizontaux en face A et les segments verticaux en face B. cette méthode permet de résoudre le problème mais avec un nombre de via très élever car pour chaque point qui est incident de segment horizontal et vertical alors ce point est un point de Via. Implémentation de cette fonction a une complexité de $\theta\,(n)$ n étant le nombre de sommet du graphe.

La seconde méthode qui résout le problème avec un nombre de Via minimum. Se basant sur la recherche des cycles impairs du graphe qui représente le problème, la fonction est donc NP-difficile. La complexité de cette méthode est déterminée à partir de nombre de cycle impair. Néanmoins si on considère n le nombre de cycle impair et m la taille du chemin maximum du graphe, on peut en déduire que la complexité est de $O\,(n \times m)$.