Projet: **Le problème de Via Minimization**

1) Structures de données utilisées

Outre les structures fournies, on utilisera pour le balayage, trois structures :

- Un tas sous forme de tableau pour le tri de l'échéancier
- La structure `T` étant une liste chainée de segments horizontaux contenant l'ordonnée de ces segments pour un accès plus rapide
- Une structure d'AVL de segments horizontaux : `AVLseg`, contenant également l'ordonnée de ces segments et la hauteur du sous arbre dont le nœud est la racine

Pour le graphe on utilise trois structures :

- Le graphe contenant un tableau de sommets : `sommets` et sa taille : `taille`
- Les sommets contenant un pointeur vers un Point ou un Segment : `data`, un caractère indiquant lequel des deux il s'agit : `PouS`, un entier étant l'indice du sommet dans le tableau du graphe : `ind` et une liste chainée de sommets avec lesquels il y a un arc : `liste`
- Une liste chainée de sommets : `Agregation`, servant à représenter les arcs dans une structure sommet

2) Organisation des fichiers du code

Le fichier Netlist.h contient l'intégralité des structures de donnée manipulées dans le projet. On trouve ensuite 5 répertoires délimitant les fichiers pour les fonctions de chacun des exercices. Ils contiennent tous un fichier .c jeu de test ayant pour nom Test<nom du répertoire> ainsi que un fichier .h pour chaque fichier source hormis le jeu de test. On trouve donc l'exercice 1 dans `Init`, l'exercice 2 dans `Intersec`, les exercices 3 et 4 dans Balayage et les exercices 5, 6 et 7 dans Graphe. L'ensemble des dépendances et des raccourcis de constructions sont visualisables dans le `Makefile`. Enfin, le répertoire `Instance_Intlist` sert à stocker les fichiers de donnée d'intersections produits par `TestIntersec`.

3) Fonctions clés

Généralités : L'insertion dans les listes chainées dans toutes les structures se fait en tête en O(1). Dans toutes les fonctions accédant aux segments de la Netlist, on les récupère via un parcourt des listes de segments incidents aux points (`Lincid`) de la Netlist, en restant sûre qu'on y accède pas deux fois par la vérification qu'il s'agisse bien de leur `p1`.

`creer_echeancier` : Cette fonction alloue et tri l'échéancier tout en transmettant sa taille par pointeur. Elle parcourt d'abord les points et segment pour les compter et allouer le tableau puis alloue, initialise et insère les extrémités

dans le tableau sous forme de tas trié par leur abscisse. Enfin, elle supprime le minimum en le plaçant à la fin du tableau dans la place libérée par sa suppression et cela jusqu'à ce que le tas n'existe plus réalisant ainsi un tri par tas en O(n log(n)) tout en allouant le moins d'espace possible cf 4).

AVL : On insère dans l'AVL de hauteur h comme dans un ABR O(h) avant d'effectuer une double rotation sur la racine si nécessaire O(1).

Pour la suppression d'un nœud, on échange sa valeur avec le minimum (maximum) de son fils droit (resp. gauche) selon le déséquilibre de hauteur. On peut ensuite libérer le nœud étant l'ancien min (resp. max) du fils droit (gauche) après avoir remis son fils droit (resp. gauche) comme fils gauche (droit) de son père. Le tout en un seul parcourt descendant à chaque changement de nœud soit O(h).

`Prem_segment_apres_AVL` : On parcourt de la racine l'arbre en cherchant un nœud supérieur à l'ordonnée ord passée en paramètre. Si on le trouve, on cherche si il y un nœud dans son fils gauche dont l'ordonnée est supérieur à ord, si c'est le cas on le remplace et on recommence jusqu'à ce que ne soit plus le cas. Au pire des cas, on arrive à une feuille, cette fonction est donc aussi en O(h).

`AuDessus_AVL` : Trouver le nœud d'ordonnée minimum supérieur au nœud h passé en argument revient à faire `Prem_segment_apres_AVL` mais avec une supériorité stricte. Comme les cordonnées manipulées ne dépassent pas une précision de 0.01 on appelle `Prem_segment_apres_AVL` avec l'ordonnée de h +0.001

`detecte_cycle_impair` : On parcourt les sommets du graphe et on lance la fonction récursive de détection de cycle impair `ajout_cycle` sur chacun d'eux. Si elle aboutit, on renvoie le cycle obtenu. Cette fonction sert notamment pour effectuer des régénérations au cas où un ou des sommets forment une partie isolée dans le graphe. Si le graphe a ns sommets et na arcs, comme on parcourt au plus une fois chaque sommet dans `ajout_cycle` et qu'on fait ça pour chaque sommet, cette fonction est donc en $O(ns^2)$ même si dans les fait c'est bien inférieur à cela.

`ajout_cycle` : Fonction récursive de détection de cycle impair. Renvoie 0 si ce n'est pas un sommet faisant partie d'un cycle, 1 si il fait partit d'un parcours contenant un cycle impair, 2 ou plus si il fait partie d'un cycle impair. Cette dernière valeur est en fait l'indice du premier sommet du cycle rencontré dans le parcourt et permet ainsi d'ajouter seulement les sommets du cycle à la liste. Comme expliqué au-dessus cette fonction est en O(ns) en pire cas.

`bicolore` et `colorier` : On colorie chaque sommet en parcourant tous les sommets du graphe au plus une fois par des parcours en profondeur par la fonction récursive `colorier.` Cette fonction est don en O(ns).

4) Jeux de test

TestInit : Prend en argument un fichier .net correspondant à une Netlist, la créée en mémoire, en fait un dessin SVG puis la libère.

TestIntersec : Prend en argument un fichier .net correspondant à une Netlist, la créée en mémoire, applique la méthode naïve de détection des intersections qu'elle initialise en mémoire avant de créer un fichier de données d'intersection .int correspondant dans le répertoire Instances_Intlist. Ajoute également une ligne dans un fichier TestIntersec.data avec le temps d'exécution et le nombre de segments correspondant.

TestBalayage : Prend en argument un fichier .net correspondant à une Netlist, la créée en mémoire. Demande à l'utilisateur quelle structure il souhaite utiliser entre la liste ou l'AVL avant d'initialiser les intersections de la Netlist selon la méthode choisie. Ajoute également une ligne dans un fichier TestBalayage.data avec le temps d'exécution et le nombre de segments correspondant.

TestGrapheInit : Prend en argument un fichier .net correspondant à une Netlist et un .int contenant les intersections de cette Netlist, la créée en mémoire puis crée le graphe correspondant. Prend en 3^{e} argument 1 ou 0 selon la méthode que l'on souhaite utiliser pour l'affectation des vias : 0 pour la méthode naïve et 1 par detection de cycle impaire. Affiche le graphe ainsi que la Netlist avec les vias positionnés selon la méthode choisie.

5) Performances