Projet: Le problème de Via Minimization

Ce projet s'intéresse au problème de "Via Minimization" (minimisation du nombre de vias) qui est une étape du processus de conception de circuits électroniques.

Déroulé du projet

L'énoncé de ce projet est divisé en 2 parties et une séance finale :
- **Partie A :** qui se déroule de la séance 4 à la séance 7 avec rendu **AVANT** la séance 8 (semaine du 25/03/19). Il s'agit d'un rendu intermédiaire comportant votre code et un rapport très synthétique d'analyse de performance. Il n'y a pas de soutenance à prévoir et vous devez attaquer la partie B lors de la séance 8.
- **Partie B :** qui se déroule de la séance 8 à 10 avec rendu final lors de la séance 11 (semaine du 15/04/19). Il s'agit du rendu final de l'ensemble du projet et comportant un rapport complet reprenant le rapport de la partie A.
- **Séance 11 :** (semaine du 13/04/19) :
Ce projet est à rendre lors de la semaine 11 lors de la séance de TD/TME à votre chargé de TD/TME. A cette occasion, vous rendrez votre rapport et soutiendrez une mini-soutenance devant vos chargés de TD/TME. Lors de cette séance 11, un dernier exercice vous permettra de poursuivre le projet et d'avoir des points bonus.

Chaque partie est subdivisée en exercices qui vont vous permettre de concevoir progressivement le programme final. Il est conseillé de suivre les étapes données par ces différents exercices. Chaque exercice contient des notions qui seront introduites en cours et en TD en parallèle. Il est impératif que vous travailliez régulièrement afin de ne pas prendre de retard : chaque exercice correspond ainsi à une ou deux séances de TME précises où les chargés de TME peuvent vous aider sur l'exercice en cours.

Attention : Cet énoncé peut connaître des petites évolutions ou un apporter des précision : consulter fréquemment la page du module :
https ://www.licence.info.upmc.fr/lmd/licence/2018/ue/2I006-2019fev/

Définitions et cadre du sujet

Cadre du projet : les circuits VLSI

On appelle *circuit VLSI* (Very Large Scale Integrated) un dispositif électronique permettant la réalisation d'une fonction logique connue (stockage mémoire, calcul numérique, pilotage de robot,...). Un circuit est entièrement décrit par un schéma électronique donnant les liens logiques entre les différents éléments du circuit. Dans ce projet, nous considérons des circuits dont on ignore la fonction et l'utilisation : ceci nous permet ici de donner une définition schématique d'un circuit.

Notons qu'il existe principalement deux types de technologies : les cartes de circuits imprimés (Printed Circuit Board ou PCB) et les circuits intégrés.
Les cartes de circuits imprimés servent en général à rassembler les gros composants électroniques de manière à former par exemple les cartes mères des micro-ordinateurs, les systèmes de pilotage de robots ou les appareils de mesure. Cette technologie consiste à souder les composants sur la face supérieure et/ou inférieure d'une carte en époxy dans laquelle circulent des pistes de cuivre correspondant aux réseaux. Parmi ces pistes de cuivre, certaines permettent de véhiculer l'alimentation des composants.
Les circuits intégrés nécessitent une technologie plus récente qui a permis la miniaturisation des processeurs, ainsi appelés *puce* (chip, en anglais). La plupart des circuits intégrés sont créés sur un support en silicium dans lequel sont gravés ou superposés des pistes et des composants. Cette technologie permet de concevoir des circuits de tailles et de complexités si importantes que certains observateurs les présentent comme les objets les plus complexes créés par l'homme. Cela induit pour leur conception la nécessité d'utiliser des procédés entièrement automatisés et optimisés pour leur conception et leur fabrication.

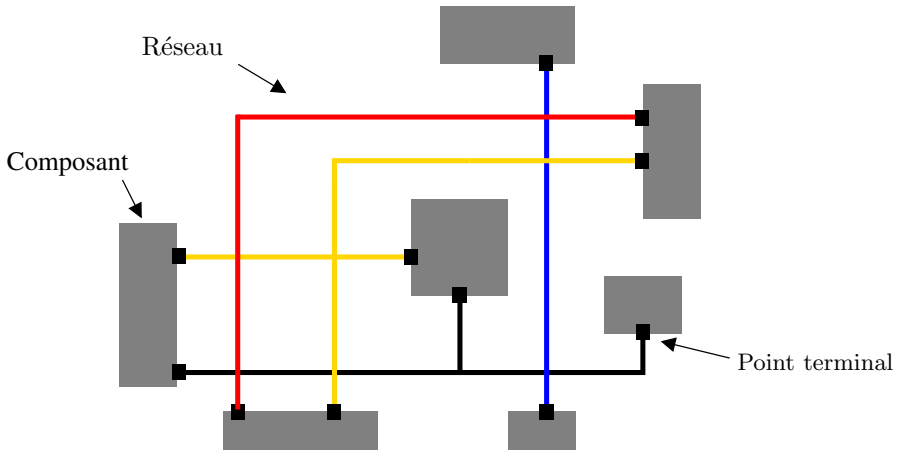

FIGURE 1 – Un exemple de circuit

Dans ce projet, un *circuit* est simplement défini par un ensemble de *composants*, un ensemble de *points terminaux* et un ensemble de *réseaux* :

- Les composants seront considérés comme des boîtes noires pouvant schématiser les composants de base (transistors, condensateurs,...) ou des macro-composants (processeurs, sous-circuits,...).
- A chaque composant correspond un ensemble de *points terminaux*. Un point terminal est une zone (assimilé à un point) du composant qui connecte le composant au reste du circuit.
- Un *réseau* est un ensemble de points terminaux reliés électriquement par des *pistes* conductrices : c'est-à-dire un ensemble de lignes droites ou courbes reliant les points.

La figure 1 illustre les définitions précédentes. Les composants (donnés par les boîtes grises) sont reliés par quatre réseaux. Chacun des 4 réseaux est donné par des lignes de couleurs différentes. Certains réseaux ne sont que des lignes droites mais certains se ramifient en plusieurs segments de droite. Par exemple, le réseau rouge est constitué de 2 segments de droites reliés au même point en formant un coude. Le réseau jaune comporte 5 segments de lignes droites dont 4 d'entre eux sont reliés au même point.

Les étapes de la conception de circuits VLSI

On appelle *problème de conception de circuits VLSI* (VLSI design problem, en anglais) l'ensemble du processus qui permet de positionner le circuit VLSI à sur un support, en prenant en compte toutes les caractéristiques de la technologie choisie concernant les composants et les réseaux.
Le problème de conception de circuits VLSI se découpe traditionnellement en plusieurs étapes. En effet, ce processus est très complexe et il serait difficile de tout concevoir en une seule étape. De plus, cette division permet à l'opérateur qui le conçoit d'intervenir manuellement au cours du développement pour orienter la conception. On distingue principalement trois étapes : le placement des composants, le routage des réseaux et l'affectation des réseaux aux faces :

- *L'étape de placement* (placement, en anglais) consiste à positionner les composants sur le support. Cette étape est réalisée en essayant de disposer les composants au mieux les uns à côté des autres en fonction du fait qu'ils doivent être connectés entre eux.
- Etant données les coordonnées des points terminaux des composants et ayant connaissance de ceux qui doivent être reliés par un réseau, *l'étape de routage* (routing, en anglais) consiste à donner des emplacements pour des pistes reliant tous les points terminaux d'un même réseau. En fait, la figure 1 indique un circuit à l'issue de l'étape de routage.
- L'étape *d'affectation des réseaux aux faces* (layer assignment en anglais) consiste à éviter de faire chevaucher deux pistes de réseaux différents. En effet, chaque réseau véhicule une information qui lui est propre et ne doit pas être connecté à un autre réseau. Heureusement, le support d'un circuit VLSI possèdent une face supérieure et une face inférieure. En fait, il est souvent impossible de faire circuler tous les réseaux d'un circuit sur une seule surface sans que ceux-ci ne se chevauchent. Les points terminaux des composants sont accessibles sur les faces (en traversant le support), mais un segment de pistes n'est présent que sur une des faces.

Problème de Via Minimization

Dans ce projet, nous allons nous intéresser à cette dernière étape de conception appelée "affectation des réseaux sur deux faces".

Lorsque deux segments de pistes appartenant à un même réseau sont sur deux faces différentes, le réseau doit néanmoins rester connecté. Concrètement, un changement de face est rendu possible par le placement d'un *via* qui permet de connecter électriquement les différents segments de piste d'un réseau entre des faces différentes. Pour un circuit imprimé, un via est un trou gainé de cuivre, percé dans le support en époxy. En circuit intégré, un via est un point de contact entre les pistes de deux couches adjacentes.

La figure 2 a) présente un circuit logique comportant trois réseaux reliant chacun deux points terminaux. Ce circuit ne peut être dessiné sur une seule couche sans que deux pistes ne se chevauchent. Par contre, sur deux faces, une solution est rendue possible par le percement d'un via. La figure 2 b) donne une solution possible sur deux faces : chaque face est alors indiquée par un type de traits différents (A : trait plein et B : train pointillé). On peut remarquer que l'un des réseaux est découpé en deux segments qui sont affectés sur deux couches faces. Un via percé entre les deux faces permet de relier les deux sous-réseaux.

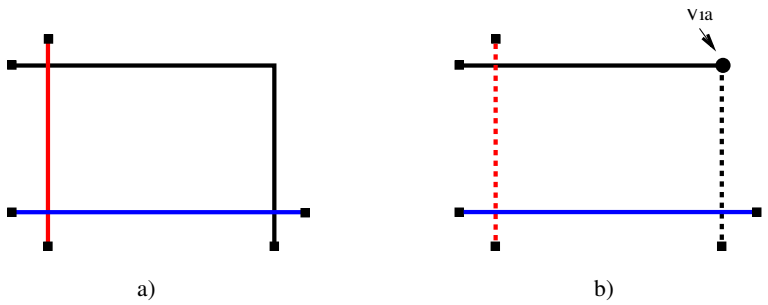

FIGURE 2 – a) Un exemple de circuit et b) son dessin sur deux couches avec un via

Comme on peut le voir sur cet exemple, les vias sont utiles et fréquemment nécessaires, mais ils dégradent la performance du circuit et peuvent être la cause de contraintes électrostatiques. De plus, les vias ont un coût de production non négligeable. On désire donc placer un nombre minimum de vias.

On considère dans ce projet le problème consistant à répartir les segments de pistes des réseaux d'un circuit VLSI entre deux faces, sans que deux segments appartenant à des réseaux différents ne soient connectés. On désire trouver une telle affectation en plaçant un nombre minimum de vias : c'est *le problème de Via Minimization.*

PARTIE A : Circuits VLSI et recherche d'intersections

Cette première partie a pour objectif de lire des instances du problème de via minimization et de rechercher les intersections éventuelles entre segments de réseaux différents.

Exercice 1 – Manipulation d'une instance du problème (Séance 4 - sem. 18/02/19)

On désire tout d'abord manipuler et afficher une instance du problème de Via Minimization. Un telle instance est aussi appelée une *netlist* c'est-à-dire la liste des réseaux.

Vous pouvez récupérer l'archive contenant les instances de la partie A sur le site du module : `Instance_Netlist.tgz`. Comme les pistes ont déjà été dessinées sur un plan, un *réseau* est un ensemble de points reliés par des *segments* de pistes. On appellera ici *segments* une ligne droite délimitée par deux points et on dira que ces deux points sont alors *adjacents*. Comme on l'a vu sur la figure 1, un réseau peut être un simple segment reliant deux points terminaux ou un réseau plus complexe ou des segments sont incidents à un point d'embranchement. On appellera *degré d'un point* le nombre de segments **d'un même réseau** qui part de ce point. Dans toutes les instances rencontrées dans ce projet, les segments seront soit horizontaux (lettre H), soit verticaux (lettre V).

Un instance sera donnée par un fichier d'extension `.net` correspondant au format donné par l'exemple suivant :

```
3             // Nb de réseau dans l'instance.
0 2 1         // Numero du réseau / Nb de pts du réseau / Nb de segments du réseau
  0 20 40     // Point 0 de coordonnées (20,40)
  1 120 40    // Point 1 de coordonnées (120,40)
  0 1         // segment reliant les points 0 et 1 du reseau
1 3 2
  0 60 0
  1 60 150
  2 20 150
  0 1
  1 2
2 4 3
  0 80 0
  1 80 200
  2 20 200
  3 150 200
  0 1
  1 2
  1 3
```

Dans l'exemple précédent, la netlist de l'instance possède 3 réseaux tels que le réseau 0 est un segment,

le réseau 1 est composé de deux segments "en coude", le réseau 3 est composé de trois segments "en T".

Vous trouverez sur le site du module un ensemble d'instances (benchmark en anglais) pour ce projet. Certaines instances ont été générées aléatoirement et d'autres proviennent de circuits VLSI réels.

Pour charger et manipuler en mémoire un réseau, on va utiliser la structure de données suivantes.

```
struct segment;

typedef struct cell_segment{
  struct segment* seg;
  struct cell_segment *suiv;
} Cell_segment;

typedef struct segment{

  int NumRes;  /* Numero du reseau auquel appartient ce segment*/

  int p1, p2;  /* Numero des points aux extremites du segment */
               /* En utilisant la numerotation de T_Pt */
               /* p1 est le point en bas a gauche par rapport a p2*/

  int HouV;    /* 0 si Horizontal et 1 si Vertical */

  struct cell_segment *Lintersec;  /* Liste des segments en intersection */

} Segment;

typedef struct point{
  double x,y; /* Coordonnees du point */

  int num_res; /* Numero du reseau contenant ce point = Index du tableau T_res*/

  Cell_segment  *Lincid; /* Liste des segments incidents a ce point */

} Point;

typedef struct reseau{

  int NumRes; /* Numero du reseau = Indice dans le tableau T_Res */

  int NbPt;  /* Nombre de points de ce reseau */

  Point* *T_Pt; /* Tableau de pointeurs sur chaque point de ce reseau */

} Reseau;
```

En utilisant cette structure, un réseau est donné comme un ensemble indicé de `NbPt` points (numérotés de 0 à `NbPt` -1) dans le tableau `T_pt`. On retrouve les segments en regardant, pour chaque point, les segments incidents à ces points. Pour un segment donné, on donne le numéros de ces points extrémités dans le tableau `T_pt`, son orientation H ou V et la liste de ses intersections avec d'autres segments de réseau différents. **Attention,** cette liste des intersections Lintersec est initialisée comme une liste vide. Nous la remplirons dans les exercices suivants.

Ainsi, la structure de données suivante code une instance netlist complète.

```
typedef struct netlist{
   int NbRes;  /* Nombre de reseaux */
```

```
3
4     Reseau* *T_Res; /* Tableau  pointant sur chaque reseau        */
5
6  } Netlist;
```

On peut remarquer que l'on numérote les `NnRes` réseaux d'une netlist de 0 à `NbRes` -1 et ce numéro correspond à l'index du tableau `T_res`. Ainsi un point du circuit peut être donné par son numéro de réseau suivi de son numéro de point.

Q 1.1 Implémentez un ensemble de méthodes `Netlist` qui permettent de créer et d'allouer une instance de notre structure à partir d'un fichier d'entrée. (Vous pouvez utiliser la bibliothèque d'Entrée/Sortie qui vous a été fournie dans les TME de début de semestre).

Q 1.2 Dans le but de valider votre code de lecture d'instance, construisez une fonction qui affiche dans un fichier le contenu d'une `Netlist` en respectant le même format que celui contenu dans le fichier (cela revient à re-créer le fichier d'entrée). Tester votre code sur plusieurs instances.

Q 1.3 On désirer afficher les instances. Pour cela, il y a plusieurs possibilités mais nous voulons un outil facile d'accès et permettant d'afficher des instances de grandes tailles avec ces outils. Nous allons utiliser le format d'images SVG (Scalable Vector Graphics) qui est très employé pour décrire des graphiques simples et qui est très utilisé pour internet. Votre code va créer un fichier au format SVG pour html qui sera ainsi lu directement par votre explorateur internet préféré.
Nous vous proposons sur le site du module une petite librairie C très très simple qui crée un fichier SVG avec extension html. Il s'agit d'un struct `SVGwriter` qui est manipulé par des méthodes permettant de créer le fichier, ajouter des lignes et des points et changer de couleurs. Il y a également une génération aléatoire de couleur de segments (pensez à initialiser la génération aléatoire si vous désirez obtenir des couleurs diverses). Vous pouvez librement reprendre, modifier, adapter ou intégrer ce code au vôtre.
Créer un programme `VisuNetlist` qui lit une instance et écrit sur disque un fichier au format svg (donc d'extension `.svg`.) reprenant le nom de l'instance et permettant de visualiser l'instance en utilisant le browser web de votre choix.

Exercice 2 – Recherche des intersections (Séance 5 - Sem. 25/02/2019)

Nous allons, dans les deux exercices suivants, rechercher toutes les intersections entre segments appartenant à des réseaux différents. A l'issue de cette étape, la structure de données permettra de connaître, pour un segment donné, la liste des segments qu'il croise (grâce au champs `Lintersec`) afin de pouvoir placer ce segment sur une face ou l'autre du support du circuit.
Les intersections dans nos instances n'ont lieu qu'entre un segment horizontal et un segment vertical appartenant à des réseaux différents : il n'y a pas non plus deux segments qui seraient "l'un sous l'autre" : c'est-à-dire deux segments verticaux (respectivement deux segments horizontaux) se chevauchant. Ainsi, nous pouvons vérifier simplement si deux segments s_v et s_h s'intersectent : il suffit de vérifier si l'abscisse du segment vertical s_v est comprise dans l'intervalle défini par les abscisses du segment horizontal s_h et que l'ordonnée du segment s_h est comprise dans l'intervalle défini par les ordonnées du segment s_v. D'autre part, les coordonnées des points sont des entiers et on suppose que deux points ou deux segments ont même abscisse (respectivement même ordonnée) si leur abscisse (resp. coordonnée) coïncide exactement (ce qui revient à considérer des pistes d'épaisseur inférieur strictement à 0,5).

Q 2.1 Implémentez une fonction `int intersection(Netlist *N, Segment *s1, Segment *s2)` qui

vérifie si le segment s_1 intersecte le segment s_2.

La complexité de ce problème de recherche d'intersections a pour paramètre le nombre n de segments de l'instance. Remarquons tout d'abord qu'il existe un algorithme évident en $O(n^2)$ pour cette recherche d'intersections : cet "algorithme naïf" consiste à comparer deux à deux tous les segments.

Q 2.2 Implémenter une fonction `nb_segment` qui retourne le nombre de segments d'une instance netlist.

Q 2.3 Implémenter une fonction retournant un tableau dont chaque case est un pointeur sur l'un des segments. Implémenter la fonction `intersect_naif` pour rechercher ces intersections. Elle consiste à comparer deux à deux tous les segments et à remplir les listes `Lintersec` de chacun des segments.

Q 2.4 Après utilisation d'un fonction de recherche d'intersection, on désire sauvegarder sur disque les listes d'intersections pour effectuer l'affectation des segments aux couches dans la partie B du projet. Pour cela, implémenter une fonction `Sauvegarde_intersection` prenant en paramètre une Netlist et qui écrit sur disque la liste des intersections dans un fichier au format .int. Un fichier au format .int possède le même nom que l'instance à laquelle il correspond et se compose d'une liste de sextuplets $(r_1\ p_1^1\ p_2^1\ r_2\ p_1^2\ p_2^2)$ indiquant que le segment reliant les points p_1^1 et p_2^1 du réseau r_1 intersecte le segment reliant les points p_1^2 et p_2^2 du réseau r_2 (une intersection correspond à un unique sextuplet du fichier).

Exercice 3 – Méthode par balayage (Séance 5-6 - Sem. 25/02/19 et 04/03/19)

Dans le cas où tous les segments se coupent les uns les autres, le nombre d'intersections est également de l'ordre de $O(n^2)$: on peut donc en déduire que, dans ce pire des cas, l'algorithme naïf est de complexité minimale pour le problème.

Dans notre cas d'utilisation, pour les instances que l'on manipule, tous les segments sont dessinés dans un plan et sont assez courts : ainsi le nombre d'intersections par segment est très limité. Il est donc intéressant de rechercher un algorithme plus efficace que l'algorithme naïf. Nous allons utiliser une procédure par balayage pour réaliser la même tâche avec une complexité moindre. L'idée est la suivante : on considère une droite de balayage verticale imaginaire qui se déplace à travers l'ensemble de segments, de la gauche vers la droite. A une abscisse x donnée de son balayage, notons k_x le nombre de segments horizontaux coupant la droite. On considère aussi l'ensemble T de taille k_x de tous ces segments horizontaux, triés par leurs ordonnées : voir figure 3. Un segment vertical d'abscisse x ne pourra alors intersecter que les segments de T.
Pour mettre en œuvre ce balayage, on considère *l'échéancier des points d'événement* E qui est défini comme une séquence d'abscisses, triées de la gauche vers la droite, qui vont définir les positions d'arrêt de la droite de balayage. Dans notre cas, les points d'événements seront les extrémités gauche et droite des segments horizontaux et les abscisses des segments verticaux. **Attention :** dans ce tri, pour deux points de même abscisse, un point gauche de segment vertical est **avant** un segment vertical et un segment vertical est **avant** un point droit de segment horizontal : en effet, l'évènement menant à la recherche d'intersection est le fait de balayer un segment vertical et il ne faut pas qu'à ce moment, les segments horizontaux pouvant le croiser soit hors de la structure T.

On considère la structure suivante pour stocker une extrémité dans l'échéancier E.

```
1 typedef struct extremite{
```

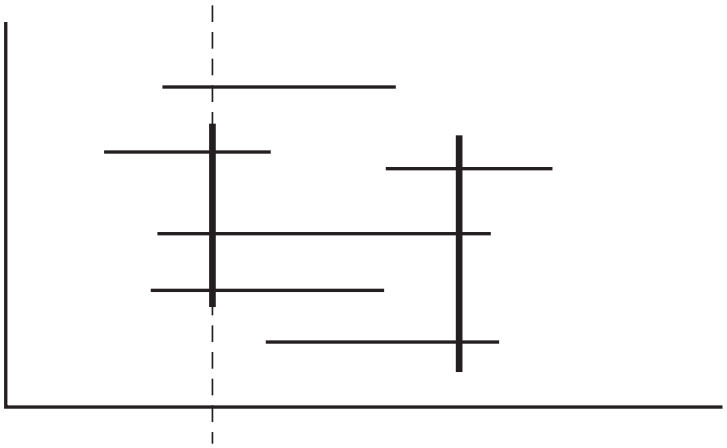

FIGURE 3 – Un exemple de balayage

```
2
3   double x;  /* Ordonnee du point */
4
5   int VouGouD; /* 0 si segment V / 1 si pt gauche d'un segment H / 2 si pt droit d'un
        segment H */
6
7   Segment * PtrSeg; /* Pointeur sur le segment correspondant a l'extremite */
8
9   int NumPt;  /* si segment H: numero du point correspondant */
10
11 } Extremite;
```

Q 3.1 Choisir une structure adaptée pour manipuler une séquence triée `E` de pointeurs sur `Extremite` correspondant à l'échéancier. Remarquons que, pour un point du plan, il peut y avoir plusieurs points-extrémités de segments. Implémenter la fonctions `creer_echeancier` qui, à partir d'une `Netlist`, crée la structure et, si besoin, `tri_échéancier` qui trie l'échéancier en fonction de l'abscisse croissante. Effectuer l'une après l'autre, ces deux fonctions doivent posséder une complexité en $O(n \log n)$ avec une consommation mémoire minimale.

L'échéancier E va être lu dans l'ordre croissant des abscisses. On commence avec une structure vide pour T, puis on opère le comportement adapté en fonction des points d'évènement :

- si le point d'événement correspond à l'extrémité gauche d'un segment horizontal, on insère ce segment dans T,
- si le point d'événement correspond à l'extrémité droite d'un segment horizontal, on supprime ce segment de T,
- si le point d'événement correspond à l'abscisse d'un segment vertical, on génère la liste des segments de T qu'il intersecte.

La structure T qui contient des segments horizontaux est donc triée selon une clef égale à leurs ordonnées. On considère les fonctions `Insérer(s,T)` et `Supprimer(s,T)` qui permettent d'insérer un segment ou de supprimer un segment de T ; la fonction `Prem_segment_apres(y,T)` qui retourne un

pointeur sur le premier segment de T d'ordonnée juste au-dessus de l'ordonnée y (et retourne NULL s'il n'y en a pas) ; et la fonction `AuDessus(h,T)` qui retourner un pointeur sur le segment qui est juste au-dessus du segment de pointeur h (et retourne NULL s'il n'y en as pas).

L'algorithme global s'écrit alors comme suit :

```
Créer l'échéancier E trié selon les abscisses
T ← ∅
pour chaque point r de E dans l'ordre croissant faire
    si r est extrémité gauche d'un segment horizontal h alors
        Insérer(h,T)
    fin
    si r est extrémité droite d'un segment horizontal h alors
        Supprimer(h,T)
    fin
    si r est l'abscisse d'un segment vertical v alors
        y1 ← ordonnée du point le plus bas de v
        y2 ← ordonnée du point le plus haut de v
        h ← Prem_segment_apres(y,T)
        tant que h ≠NULL et ordonnee de h ≤ y2 faire
            si h et v de réseaux différents alors
                Ajouter h à Lintersec de v
                Ajouter v à Lintersec de h
            fin
            h ← AuDessus(h,T)
        fin
    fin
fin
```

Algorithme 1 : Recherche des intersections

Q 3.2 On veut utiliser une structure de liste triée de pointeurs sur segment pour la structure T. Donner les fonctions et le code correspondant à la méthode de balayage.

Q 3.3 Donner la complexité (pire-cas) en fonction de n de cette méthode lorsque la structure T est implémentée par une liste chaînée triée. On note α une borne maximale sur nombre de segments horizontaux traversés par la droite quelque soit l'abscisse du plan, c'est-à-dire $\alpha \geq k_x$ pour tout x dans l'échéancier E. Si l'on suppose que cette donnée α est un paramètre de l'algorithme indépendant de n, donner une autre mesure de la complexité de l'algorithme.

Q 3.4 On désire comparer les performance des algorithmes `intersect_naif` et `intersect_balayage`. Pour cela, nous allons tracer les courbes des temps d'exécution des deux algorithmes pour les instances de la benchmark. Noter que les instances appelées `ibm` sont en fait des instances dont tous les réseaux de même échelle, situés sur des surfaces identiques : on peut donc considérer qu'elles possèdent toute une borne α commune malgré un nombre n de segments bien différents. En utilisant ces différentes instances, justifier si les courbes théoriques respectent ou non vos estimations de complexité.

Exercice 4 – Une structure d'AVL pour stocker les segments de pistes (sem. 6-7 - 04/03 et 18/03)

Afin d'améliorer la complexité de la méthode par balayage de l'exercice précédent, on propose d'implémenter la structure T par un *arbre binaire équilibré*.

Q 4.1 En adaptant la structure d'AVL vu en cours et TD, implémenter une structure d'arbre AVL dont chaque sommet contient un pointeur sur un segment horizontal. La clef de la structure est l'ordonnée des segments.

Q 4.2 Implémenter les méthodes nécessaires à la fonction de balayage. Il est important de tester au fur et à mesure vos méthodes. Pour cela, implémenter une fonction d'affichage de l'AVL dans le but de pouvoir visualiser de petites instances du problème. Tester vos méthodes "à la main".

Q 4.3 Utiliser la structure d'AVL et vos méthodes pour la méthode par balayage.

Q 4.4 Reprenez le cadre de la question 3.3 pour donner les complexités des méthodes de manipulation de l'AVL et de la méthode par balayage.

Q 4.5 Reprenez la question 3.4 pour l'amélioration du balayage en utilisant votre structure d'AVL. Concluez quant à la meilleure structure et complexité pour la recherche d'intersections.

PARTIE B : Graphe de modélisation et minimisation des vias

Dans cette partie, nous allons étudier plusieurs méthodes pour positionner les segments sur les faces en minimisant le nombre de vias correspondant à cette affectation.

Exercice 5 – Graphe de modélisation (Séance 8 - Sem. 25/03/2019)

Nous allons tout d'abord modéliser le problème par un graphe afin de mettre en œuvre facilement ces méthodes.
On considère une `Netlist` pour lequel on connaît toutes les intersections. On appelle *graphe de modélisation* de la `Netlist` un graphe non-orienté $G = (V_S \cup V_P, C \cup X)$ tel que

- V_S est un ensemble de sommets associés aux segments ;
- V_P est un ensemble de sommets associés aux extrémités des segments ;
- C est un ensemble d'arêtes dites de *continuité* tel qu'il y a une arête reliant un sommet v_s de V_S à un sommet v_p de V_P si le point correspondant à v_p est une extrémité du segment correspondant à v_s ;
- X est un ensemble d'arêtes dites de *conflit* tel qu'il y a une arête reliant deux sommets v_s et $v_{s'}$ de V_P si les deux segments correspondant à ces sommets s'intersectent.

On remarque que ce graphe peut contenir un grand nombre de sommets mais peu d'arêtes par rapport aux nombres de sommets. On choisit d'utiliser une structure de liste d'adjacence pour implémenter un tel graphe.
Attention : il faut adapter légèrement la structure vue en cours de manière à conserver un lien entre les sommets du graphe et les segments et points de la Netlist : ainsi on stockera dans chaque sommet si ce sommet correspond à un segment de la Netlist ainsi qu'un pointeur sur ce segment ; ou si ce sommet correspond à un point de la Netlist ainsi qu'un pointeur sur ce point. On peut remarquer que, même s'il y a deux types logiques d'arêtes, il n'y a pas besoin de coder le type de l'arête.

Q 5.1 Implémentez une fonction qui crée un `Graphe` à partir du fichier de l'instance `Netlist` et du fichier .int des intersections correspondant.

Q 5.2 On désire visualiser le graphe obtenu en le dessinant sur un plan. Les sommets v_p de V_P correspondant à des points seront placés aux coordonnées de ces points. Les sommets v_s de V_S correspondant à des segments seront positionnés aux coordonnées du milieu de ce segment. Les arêtes de X et de C seront représentées par des segments de droite. Vous pouvez donc utiliser le même procédé postscript que pour la partie A. Noter que les arêtes de C reprendront approximativement les tracés des pistes de l'instance. Ainsi, en comparant la représentation d'une `netlist` et du graphe de modélisation, vous pouvez visualiser si votre création du graphe correspond bien au résultat attendu.

Exercice 6 – Une première méthode (Séance 9 - Sem. 01/04/2019)

Le problème de Via Minimization consiste à répartir les segments de pistes de tous les réseaux d'un circuit VLSI entre une face A et une face B. Une méthode simple (et encore utilisée en pratique)

consiste à placer tous les segments horizontaux sur la face A et tous les segments horizontaux sur la face B. Cette méthode implique que tous les points d'un réseau incident à au moins un segment horizontal et au moins un segment vertical vont devenir des vias.

Pour coder une solution du problème de Via Minimization, nous allons utiliser un tableau d'entiers S indicé sur les numéros de tous les sommets du graphe G (y compris donc les sommets correspondant à des points) : pour un sommet i correspondant à un segment, $S[i]$ contiendra 1 ou 2 suivant que le segment correspondant est positionné sur la face 1 ou 2 ; pour un sommet i correspondant à un point extrémité, $[i]$ contiendra 0 si ce point correspond à un via ou sinon un nombre positif quelconque.

Q 6.1 Implémenter cette méthode pour remplir le tableau S. Pour cela, attribuer d'abord les valeurs des segments puis déterminer quel point est un via ou non.

Q 6.2 Etant donné un tableau S, donner une fonction qui affiche le nombre de vias et la solution. Pour cela, vous pouvez utiliser le format postscript. Utiliser des couleurs bien différenciées pour les faces 1 et 2 et un point noir pour les vias (sans représenter les points non vias).

Exercice 7 – Méthode par recherche de cycles impairs (semaine 9-10 - 01/04 et 08/04)

On désire implémenter des méthodes permettant de placer moins de vias parmi les points extrémités. Ce problème a été démontré "NP-difficile", c'est-à-dire pour lequel on ne connaît pas d'algorithme polynomial pour le résoudre. On va néanmoins s'intéresser à déterminer des "bonnes solutions".

L'algorithme est basé sur la notion de cycle impair. On appelle *cycle impair* (resp. *pair*) un cycle d'un graphe contenant un nombre impair (resp. pair) de sommets. Dans le graphe G de modélisation, remarquons que l'on peut colorier alternativement les sommets adjacents "face 1"/"face 2" : ainsi les sommets-segments correspondant à ces sommets seront positionnés en face 1 ou face 2 en respectant les intersections sans utiliser aucun via. En revanche, les segments correspondant à un cycle impair ne peuvent pas être coloriés alternativement "face 1"/"face 2" : en fait, on ne peut pas positionner ces segments sans utiliser au moins un via. On doit alors choisir pour via n'importe lequel des sommets du cycle impair correspondant à une extrémité.

Q 7.1 Implémenter une fonction `detecte_cycle_impair` qui, à partir d'un graphe et d'un tableau S détecte si les sommets i de G ayant $S[i] \neq 0$ forment un cycle impair. Pour cela, vous pouvez utiliser le code de détection de cycle vu en cours en l'adaptant de la manière suivante : on utilise un tableau de marquage M qui donne une affectation 0, 1 ou 2 aux sommets pendant le parcours. Au départ, tous les sommets sont marqués dans M à 0 si $S[i] = 0$ et -1 sinon. Le sommet de départ du parcours est marqué à 1. Puis au cours du parcours en profondeur : on considère les cas suivants :

- pour un sommet avec $M[i] = 0$ pas d'exploration des sommets adjacents ;
- pour un sommet marqué à -1 et rencontré à partir d'un père à 1 (resp. à 2) : le marquer à 2 (resp. à 1) puis explorer ses sommets adjacents ;
- pour un sommet marqué à 1 (resp. à 2) et rencontré à partir d'un père marqué à 1 (resp. à 2), on détecte un cycle impair : stop.

En cas de détection d'un cycle impair, la fonction retourne une liste chaînée des sommets formant le cycle (ou liste vide si pas de cycle impair).

Q 7.2 Implémenter la fonction `Ajout_vias_cycle_impair` d'ajouts de via suivante : on part d'un

tableau S où tous les sommets sont mis à -1 : cela correspond à l'absence de tout via. On va utiliser autant de fois que nécessaire la fonction `detecte_cycle_impair` : tant qu'elle ne retourne pas NULL, on marque à 0 dans S un sommet-point du cycle impair : cela correspond à ajouter un via.

Q 7.3 On veut implémenter alors une fonction `bicolore` qui, à partir du tableau S obtenu par la fonction `Ajout_vias_cycle_impair`, attribue à tous les sommets i ayant $S[i] = -1$ la valeur $S[i] = 1$ ou $S[i] = 2$ de façon à ce que, pour chaque arête, ses deux sommets extrémités aient soit $S[i] = 0$ pour au moins un des deux, soit deux valeurs différentes. Pour cela, utiliser une fonction de parcours qui ignore les sommets marqués à 0.

Q 7.4 Tester la méthode en utilisant la fonction vue à la question 6.2.

Q 7.5 Donner la complexité de cette méthode.

Q 7.6 IMPORTANT : Dans votre rapport final, n'oubliez pas de proposer des jeux d'essais pour déterminer l'efficacité de la méthode par rapport à la méthode de l'exercice précédent.

RENDU final : Le rapport et le code final, sont à rendre à votre chargé de TD/TME, en début de séance 11 (semaine du 15/04/19). Votre rapport décrivant votre code et votre travail de réflexion, ainsi qu'une analyse statistique et théorique de sa qualité (voir sur le site document "Rendu de projets").
Séance 11 (semaine du 15/04/19) :
A cette séance, vous lui montrerez votre code, son fonctionnement et ses performances et les deux membres du binôme répondront à des questions.
Remarque : Lors de cette séance 11, un dernier exercice ci-dessous vous permettra de poursuivre le projet et d'avoir des points bonus.

Exercice 8 – Bonus (Séance 11 - Sem. 15/04/19)

D'autres idées peuvent mener à des algorithmes plus rapides ou donnant de meilleurs solutions qu'avec les exercices précédents. Nous vous invitons dans cet exercice "bonus" à proposer vos propres algorithmes.
Quelque piste d'idées :
- pour donner des solutions rapides : il faut remplir rapidement S de manière à obtenir une réponse NULL à la fonction `detecte_cycle_impair`. Proposer des façons de remplir S en utilisant des parcours bien choisis.
- pour donner des solutions contenant moins de vias, on peut construire une méthode qui, partant d'une solution S réalisable, tente de l'améliorer itérativement en déplaçant les vias un à un de manière à en utiliser moins.

Q 8.1 Proposer votre (vos) propre(s) algorithme(s) et tester-les comparativement aux méthodes rencontrées jusqu'ici.

L'exercice proposé ici est prévu **après** le rendu du rapport final. Il s'agit donc d'un exercice bonus dont vous pouvez rendre les résultats jusqu'au 4 avril. A cette date, nous comparerons les meilleures méthodes suivant un classement rapidité/qualité (c'est-à-dire qu'une méthode donnant des résultats peu intéressants doit être très rapide ; et que l'on attend d'une méthode lente d'être efficace). Les meilleurs résultats de chaque groupe et de tout le module doteront leurs créateurs de points supplémentaires.